# 应用笔记AN3523

# 基于CMX的飞思卡尔USB_Lite的高级应用

文件编号：AN3523

版本：0，09/2007

苏州大学飞思卡尔嵌入式系统研发中心翻译

http://sumcu.suda.edu.cn

2009年11月

# 目录

# 基于CMX的飞思卡尔USB_Lite的高级应用

Eric Gregori
产品专家－嵌入式固件

## 1 引言

本文档包含了高级CMX USB应用。其中CMX USB协议栈的详细资料包含在AN3492（“USB和CMX协议栈的使用”）中。AN3492介绍了USB的相关内容、枚举及主机和设备端协议栈的API。在本文（即AN3523）中讨论的固件就在该协议栈的应用层中。

CMX USB协议栈包括了主机和设备端的协议栈。目前该协议栈可以运行在ColdFire和8位的微处理器JM60上。该协议栈也可以运行在将来的低功耗V1版的核上。

本文中讨论的应用建立在USB协议栈固件的顶端。该协议栈的源代码可以从www.freescale.com上下载。和该文档一起下载的二进制代码，可以写入到开发板上以演示本文讨论的应用。

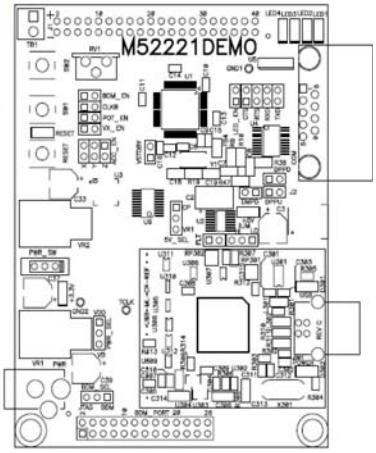
目前有3种开发板支持CMX USB协议栈：M52221DEMO，M52223EVB，M52211EVB及8位的DEMO9S08JM60。JM60目前只支持设备方示例。

## 2 基于加速度传感器的USB鼠标

基于加速度传感器的USB鼠标固件演示了使用CMX USB栈开发HID鼠标。加速度传感器用来检测关于演示板相对于的重力的方向。板上的SW1和SW2被用作鼠标的左键和右键。

倾斜板左边或者右边即移动鼠标指针左或右。
倾斜板前面或者后面即移动鼠标指针上或下。
倾斜程度越大，鼠标指针移动的越快。

倾斜板以移动鼠标指针（注意接口在右边）




鼠标按钮

加速度传感器为MMA7260Q，它是一3轴加速度传感器、灵敏度从1.5G到6.0G，可通过编程调整。倾斜的信息以电压的形式输出，因此MCU需要一A/D转换器。


## 2.1 使用A/D转换器读取加速度传感器

ColdFire ADC的参数如下：

- 12位的精度
- ADC的最大时钟频率为5.0MHZ，周期200ns
- 采样率高达1.66兆/秒
- 单次转换时间为8.5个ADC时钟周期(8.5*200ns=1.7us)
- 附加转换（Additional conversion）时间是6个ADC时钟周期(6*200ns=1.2us)

- 同步模式下，8路转换时间为26.5个ADC时钟(26.5*200ns=5.3us)
- 支持同步采样且能保持2次
- 支持高达8路的序列采样存储
- 内部复用选择2到8个输入
- 节能模式允许自动关闭/打开ADC模块的所有或部分
- 未被选择的输入可以容纳/源电流而不影响ADC性能,且支持在嘈杂的工业环境中的操作
- 如果超出输出范围的限制(过高或过低),或者在零交点,在扫描结束时会产生一个可选的中断
- 可以减去一预先写入的补偿值来进行采样纠错
- 有符号或无符号的结果
- 为了支持复杂输入类型的所有的输入引脚而产生单结果或者不同的输入

ADC的功能模块如下图所示，包括两个四通道输入选择模块，这两个模块和两个独立的S/H电路相连，将信息馈给两个12位的ADC。这两个转换器将转换结构存储在一缓冲区中，以待进一步处理。

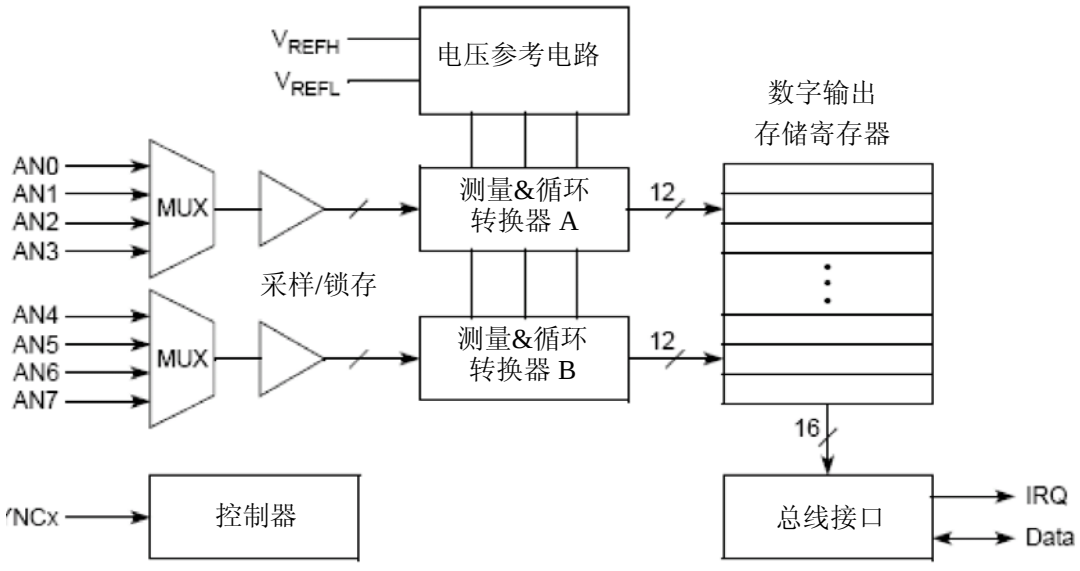



图1 ClodFire A/D模块框图

该鼠标使用两路A/D通道（AN4,AN5）读取来自加速度传感器MMA7260的X（AN4）和Y（AN5）的信号。A/D设置为连续转换。

### 2.1.1 A/D初始化函数

```
/*********************************************************************
* init_adc - Analog-to-Digital Converter (ADC)                       *
*********************************************************************/
void init_adc (void)
{
    // Scan mode = Loop parallel, converters A and B run simultaneously
    // ADC clock frequency = 10.67 MHz
    // Voltage reference supplied by VDDA and VSSA
    // All ADC interrupts disabled
    //
    // Sample list for converter A:
    // Sample 0 : ANA0
    //          Low limit = $000, High limit = $FFF, Offset = $000
    // Sample 1 : ANA1
    //          Low limit = $000, High limit = $FFF, Offset = $000
    // Sample 2 : ANA2
    //          Low limit = $000, High limit = $FFF, Offset = $000
    // Sample 3 : ANA3
    //          Low limit = $000, High limit = $FFF, Offset = $000
    //
    // Sample list for converter B:
    // Sample 4 : ANB0
    //          Low limit = $000, High limit = $FFF, Offset = $000
    // Sample 5 : ANB1
    //          Low limit = $000, High limit = $FFF, Offset = $000
    // Sample 6 : ANB2
    //          Low limit = $000, High limit = $FFF, Offset = $000
    // Sample 7 : ANB3
    //          Low limit = $000, High limit = $FFF, Offset = $000
    // Initialise ADC
    //    CLST1[SAMPLE3] = %011
    //    CLST1[SAMPLE2] = %010
    //    CLST1[SAMPLE1] = %001
    //    CLST1[SAMPLE0] = 0
    MCF_ADC_ADLST1 = MCF_ADC_ADLST1_SAMPLE3(0x3) |
                     MCF_ADC_ADLST1_SAMPLE2(0x2) |
                     MCF_ADC_ADLST1_SAMPLE1(0x1);
    //    CLST2[SAMPLE7] = %111
    //    CLST2[SAMPLE6] = %110
    //    CLST2[SAMPLE5] = %101
    //    CLST2[SAMPLE4] = %100
    MCF_ADC_ADLST2 = MCF_ADC_ADLST2_SAMPLE7(0x7) |
```

```
                                MCF_ADC_ADLST2_SAMPLE6(0x6) |
                                MCF_ADC_ADLST2_SAMPLE5(0x5) |
                                MCF_ADC_ADLST2_SAMPLE4(0x4);
    //      CTRL2[STOP1] = 1
    //      CTRL2[START1] = 0
    //      CTRL2[SYNC1] = 0
    //      CTRL2[EOSIE1] = 0
    //      CTRL2[SIMULT] = 1
    //      CTRL2[DIV] = $2
    MCF_ADC_CTRL2 = MCF_ADC_CTRL2_DIV(3);
    // Power up ADC converter(s) in use
    //      PWR[ASB] = 0
    //      PWR[PUDELAY] = $d
    //      PWR[APD] = 0
    //      PWR[PD2] = 1
    //      PWR[PD1] = 0
    //      PWR[PD0] = 0
    MCF_ADC_POWER = MCF_ADC_POWER_PUDELAY(4);
    //      CTRL1[STOP0] = 1
    //      CTRL1[START0] = 0
    //      CTRL1[SYNC0] = 0
    //      CTRL1[EOSIE0] = 0
    //      CTRL1[ZCIE] = 0
    //      CTRL1[LLMTIE] = 0
    //      CTRL1[HLMTIE] = 0
    //      CTRL1[CHNCFG3] = 0
    //      CTRL1[CHNCFG2] = 0
    //      CTRL1[CHNCFG1] = 0
    //      CTRL1[CHNCFG0] = 0
    //      CTRL1[SMODE] = %010
    MCF_ADC_CTRL1 = MCF_ADC_CTRL1_SMODE(2);
    /* Pin assignments for port AN
          Pin AN7 : Analog input AN7
          Pin AN6 : Analog input AN6
          Pin AN5 : Analog input AN5
          Pin AN4 : Analog input AN4
          Pin AN3 : Analog input AN3
          Pin AN2 : Analog input AN2
          Pin AN1 : Analog input AN1
          Pin AN0 : Analog input AN0
    */
    MCF_GPIO_DDRAN = 0;
    MCF_GPIO_PANPAR = MCF_GPIO_PANPAR_PANPAR7 |
                                  MCF_GPIO_PANPAR_PANPAR6 |
```

```
                MCF_GPIO_PANPAR_PANPAR5 |
                MCF_GPIO_PANPAR_PANPAR4 |
                MCF_GPIO_PANPAR_PANPAR3 |
                MCF_GPIO_PANPAR_PANPAR2 |
                MCF_GPIO_PANPAR_PANPAR1 |
                MCF_GPIO_PANPAR_PANPAR0;
```

### 2.1.2 A/D转换的启动

函数start_AD()用来启动连续转换。函数start_AD()被调用后，A/D转换器将连续地扫描模拟信号通道0-7，并且将结果存储在寄存器0-7中。

必须注意，函数start_AD()启动A转换器（CTRL1寄存器中的START0）的转换。当CTRL2寄存器中的SIMULT位被置位后，通过使用START0或STOP0位同时启动或停止转换器A和B。

表1 CTRL2寄存器

| 字段 | 描述 |
|---|---|
| 5<br>SIMULT | A/D转换器的同步模式。此位仅仅影响并行扫描模式。<br><br>当SIMULT=1（缺省值）时并行扫描工作在同步模式下。在转换器A和同时扫描操作并且将常常导致转换器A和B的变换。START0, STOP0, SYNC0和EOSIE0控制位和SYNC0输入被用作在转换器间同时地启动和停止扫描。一次扫描在当两个扫描器之一遇到一个终止信号时停止。当并行扫描完成时，如果EOSIE0被置位那么EOSI0将被引发。CIP0状态位指示一个并行扫描在进行中。<br><br>当SIMULT=0，在转换器A和B上的扫描独立的进行。转换器B有它自己独立的一套控制系统（START1,STOP1,SYNC1,EOSIE1, SYNC1）来控制和报告它的状态。每个转换器持续的工作直到它的周期被耗尽（4个）或者一个遇到一个终止信号。在循环的并行扫描模式下，每一个转换器在前一个重复完成时继续下一个重复的工作直到转化器的STOP位被确定。<br><br>0=并行扫描独立完成<br>1=并行扫描同时完成（缺省） |

#### 2.1.2.1 tart_AD() 函数

```
void start_AD( void )
{
    /* Clear stop bits */
    MCF_ADC_CTRL1 &= ~MCF_ADC_CTRL1_STOP0;
    MCF_ADC_CTRL2 &= ~MCF_ADC_CTRL2_STOP1;
    /* Set start bit */
    MCF_ADC_CTRL1 |= MCF_ADC_CTRL1_START0;
}
```

### 2.1.3 读取A/D转换

A/D转换器在连续模式时，A/D结果将在ADRSLT寄存器中不断的被更新。这些寄存器可以被应用程序随时读取。函数read_AD(channel)返回一个16位的值来表示channel指定的通道的转换结果。

```
short read_AD( int channel )
{
    switch( channel )
    {
      case 0:
          return( ((MCF_ADC_ADRSLT0&0x7FF8)>>3) );
      case 1:
          return( ((MCF_ADC_ADRSLT1&0x7FF8)>>3) );
      case 2:
          return( ((MCF_ADC_ADRSLT2&0x7FF8)>>3) );
      case 3:
          return( ((MCF_ADC_ADRSLT3&0x7FF8)>>3) );
      case 4:
          return( ((MCF_ADC_ADRSLT4&0x7FF8)>>3) );
      case 5:
          return( ((MCF_ADC_ADRSLT5&0x7FF8)>>3) );
      case 6:
          return( ((MCF_ADC_ADRSLT6&0x7FF8)>>3) );
      case 7:
          return( ((MCF_ADC_ADRSLT7&0x7FF8)>>3) );
      }
      return(0);
}
```

## 2.2 HID鼠标报告概述

鼠标使用HID类与PC进行通信。AN3492（“USB和CMX协议栈的使用”）中详细讨论了HID类。PC将根据报告描述符将该设备自动识别为鼠标。

HID报告描述符用来识别一个目标以及目标中数据的存储方法。HID报告结构体定义在文档“Device Class Definition for Human Interface Devices（HID）Version 1.11”中，在 www.usb.org 上可以找到。

报告描述符是一项目组（groups of items），每一个项目定义了一块信息。每一块信息（项）由一组带有值的标识符组成。标识符决定了信息的类型，值包含了实际的配置数据。例如，一个鼠标描述符包含了一个USAGE标记，此标记定义了一个鼠标的报告。它同时也包含另一个USAGE标记，定义了报告的一段

是是键1的，另一段是键2的。最后的两个其他的USAGE定义X和Y坐标。

INPUT标记定义了从设备到PC的数据，同时OUTPUT标记定义了从PC到设备的数据。且此数据可为变量或常量。

把标记和数据组合在一表中，主机（PC）方的报告解释器自始至终读取该表。数据的说明定义在上文提到的规范中。在usb.org网站上的一个工具可以用来帮助创建报告描述符(http://www.usb.org/developers/hidpage/dt2_4.zip)。还包含很多例子。在usb.org上规范“Universal Serial Bus HID Usage Tables version 1.12”详细定义了鼠标的使用标记。

## 2.3 鼠标报告描述符举例

| Tag, value | description |
|---|---|
| 0x05, 0x01, | /* USAGE_PAGE (Generic Desktop) */ |
| 0x09, 0x02, | /* USAGE (Mouse) */ |
| 0xa1, 0x01, | /* COLLECTION (Application) */ |
| 0x09, 0x01, | /* USAGE (Pointer) */ |
| 0xa1, 0x00, | /* COLLECTION (Physical) */ |
| 0x05, 0x09, | /* USAGE_PAGE (Button) */ |
| 0x19, 0x01, | /* USAGE_MINIMUM (Button 1) */ |
| 0x29, 0x03, | /* USAGE_MAXIMUM (Button 3) */ |
| 0x15, 0x00, | /* LOGICAL_MINIMUM (0) */ |
| 0x25, 0x01, | /* LOGICAL_MAXIMUM (1) */ |
| 0x95, 0x03, | /* REPORT_COUNT (3) */ |
| 0x75, 0x01, | /* REPORT_SIZE (1) */ |
| 0x81, 0x02, | /* INPUT (Data,Var,Abs) */ |
| 0x95, 0x01, | /* REPORT_COUNT (1) */ |
| 0x75, 0x05, | /* REPORT_SIZE (5) */ |
| 0x81, 0x01, | /* INPUT (Cnst,Ary,Abs) */ |
| 0x05, 0x01, | /* USAGE_PAGE (Generic Desktop) */ |
| 0x09, 0x30, | /* USAGE (X) */ |
| 0x09, 0x31, | /* USAGE (Y) */ |
| 0x15, 0x81, | /* LOGICAL_MINIMUM (-127) */ |
| 0x25, 0x7f, | /* LOGICAL_MAXIMUM (127) */ |
| 0x75, 0x08, | /* REPORT_SIZE (8) */ |
| 0x95, 0x02, | /* REPORT_COUNT (2) */ |
| 0x81, 0x06, | /* INPUT (Data,Var,Rel) */ |
| 0xc0, | /* END_COLLECTION */ |
| 0xc0 | /* END_COLLECTION */ |

### 2.3.1 鼠标报告描述符分解（breakdown）

USAGE_PAGE(Generic Desktop)

“Generic Desktop”usage包含了对下列设备的支持：指示器（pointer）、鼠标、操纵杆、游戏手柄、键盘、小键盘、多轴控制器。该usage定义在规范“Universal Serial Bus HID Usage Tables version 1.12”的4.1节。

USAGE(Mouse)

鼠标（Mouse）是上文选择的“Generic Desktop”设备类中的一个设备。该行信息通知PC（主机）设备是一鼠标。

COLLECTION(Application)

COLLECTION是一组标记的集合。在其中，这组标记定义了通用设备的应用，如：鼠标，键盘。。。。。。

USAGE(Pointer)

下列数据定义了一个指示器（pointer）设备

COLLECTION(Physical)

6.2.2.6节，“Device Class Definition for Human Interface Device（HID）Version 1.11”，“将测量和感应数据集和单点结合起来的传感设备”。

USAGE_PAGE(Button)

USAGE_MINIMUM(Button 1)

USAGE_MAXIMUM(Button 3)

下列标记描述了3个按键（鼠标上的3个按键）

LOGICAL_MINIMUM(0)

LOGICAL_MAXMUM(1)

按键只能有2个值，0或1

REPORT_COUNT(3)

REPORT_SIZE(1)

按键值包含在一个单字节（SIZE）中的3位（COUNT）中

INPUT(Data,Var,Abs)

上文描述的按键的USAGE信息从设备发给主机。数据可以为可变的且绝对的。

REPORT_COUNT(1)

REPORT_SIZE(5)

INPUT(Cnst,Ary,Abs)

定义了5位的微调电容器（paddr），因此按键信息包含在一个单字节中。

填充数据来由设备发送给PC，可以为任意常量

USAGE_PAGE(Generic Desktop)

USAGE(X)

USAGE(Y)

X和Y数据是“Generic Desktop“usage的子集。该usage定义在规范“Universal Serial Bus HID Usage Tables version 1.12”4.2节中。

LOGICAL_MINIMUM(-127)

LOGICAL_MAXMUM(127)

REPORT_COUNT8()

REPORT_SIZE(2)

INPUT(Data,Var,Abs)

X和Y数据由设备发送给PC。该数据为2字节，最大值为127最小值为-127。该数据应当作为可变的和相对的值读取（be read as Variale and Relative）。

根据描述描述符PC将把X和Y数据看作相对数据，用一个基值来加或减。指针向右移，设X为一个正数。X越大，指针移动的越快。相同的规格同样用于Y。微软要求X和Y同为绝对或者相对的，而不是二者的组合。

END_COLLECTION

物理集合的结束

END_COLLECTION

应用集合结束

## 2.4 固件

CMX USB协议栈的设备API在AN3492(“USB及CMX协议栈的使用”)中详细描述了。在USB设备固件的顶端是main()函数。通过报告（report）队列来发送和接受数据。鼠标使用单独的IN报告队列（方向通常相对主机PC而言）。

PC期望X和Y数据为相对的，所以通过参考A/D读取值减去一新A/D读取值得到一个差值。参考A/D读取值在复位后获得。在复位后板应平放于桌面，这样参考A/D读取值才能被正确的设置。

```
int hid_mouse(void)
{
    int x=0;
    hcc_u8 in_report;
    unsigned char oldx, oldy;
```

```
char delta;
// init HID state machine and USB driver
HID_init(0, 0);
// Take a reference snapshot of the current A/D values
// for the X and Y position. Assume that this is base reading
// with the board flat on a table.
oldx = (unsigned char)(read_AD( 4 )>>4);
oldy = (unsigned char)(read_AD( 5 )>>4);
// Create a report queue of 3 bytes ( see report descriptor )
in_report=hid_add_report(rpt_in, 0, 3);
while(1)
{
    // Process HID report queues
    hid_process();
    // Only send new report if queue is empty
    if (!hid_report_pending(in_report))
    {
        // Assume no motion or button pushes
        DIR_REP_BUTTONS(hid_report) = 0;
        DIR_REP_X(hid_report) = 0;
        DIR_REP_Y(hid_report) = 0;
        // Calculate delta from reference position ( oldx )
        delta = (char)(oldx - (read_AD( 4 )>>4));
        // Hysterisys for us older folks
        if ( (delta>THRESHOLD) || (delta<-THRESHOLD) )
        {
            // insert delta into X position in report
            DIR_REP_X(hid_report) = delta;
        }
        // Calculate delta from reference position ( oldy )
        delta = (char)(oldy - (read_AD( 5 )>>4));
        // Hysterisys
        if ((delta>THRESHOLD) || (delta<-THRESHOLD) )
        {
            // insert delta for Y position in report
            DIR_REP_Y(hid_report) = delta;
        }
        // If SW1 pushed, set button 1 bit in report
        if (SW1_ACTIVE())
            DIR_REP_BUTTONS(hid_report) = 0x01;
        // If SW2 pushed, set button 2 bit in report
        if (SW2_ACTIVE())
            DIR_REP_BUTTONS(hid_report) = 0x02;
        // Insert report into queue
```

```
                hid_write_report(in_report, (hcc_u8*)hid_report);
            }
        }
        return(0);
}
```

## 2.5 报告（reports）中数据定位

DIR_REP_X， DIR_REP_Y 和 DIR_REP_BUTTONS 宏将数据插入函数 hid_add_report()声明的3字节报告（report）中。这些位位置定义在HID报告描述符中。

REPORT_COUNT(3)

REPORT_SIZE(1)

REPORT_COUNT(1)

REPORT_SIZE(5)

按键信息存储在报告字节0的第0，1和2位中，按键1就是第0位。微软将按键1定义为左键，按键2为右键，按键3则为中（center）键。

USAGE(X)

USAGE(Y)

LOGICAL_MINIMUM(-127)

LOGICAL_MAXIMUM(127)

REPORT_SIZE(8)

REPORT_COUNT(2)

X坐标存储在报告字节1中，相应的Y坐标则在字节2中。

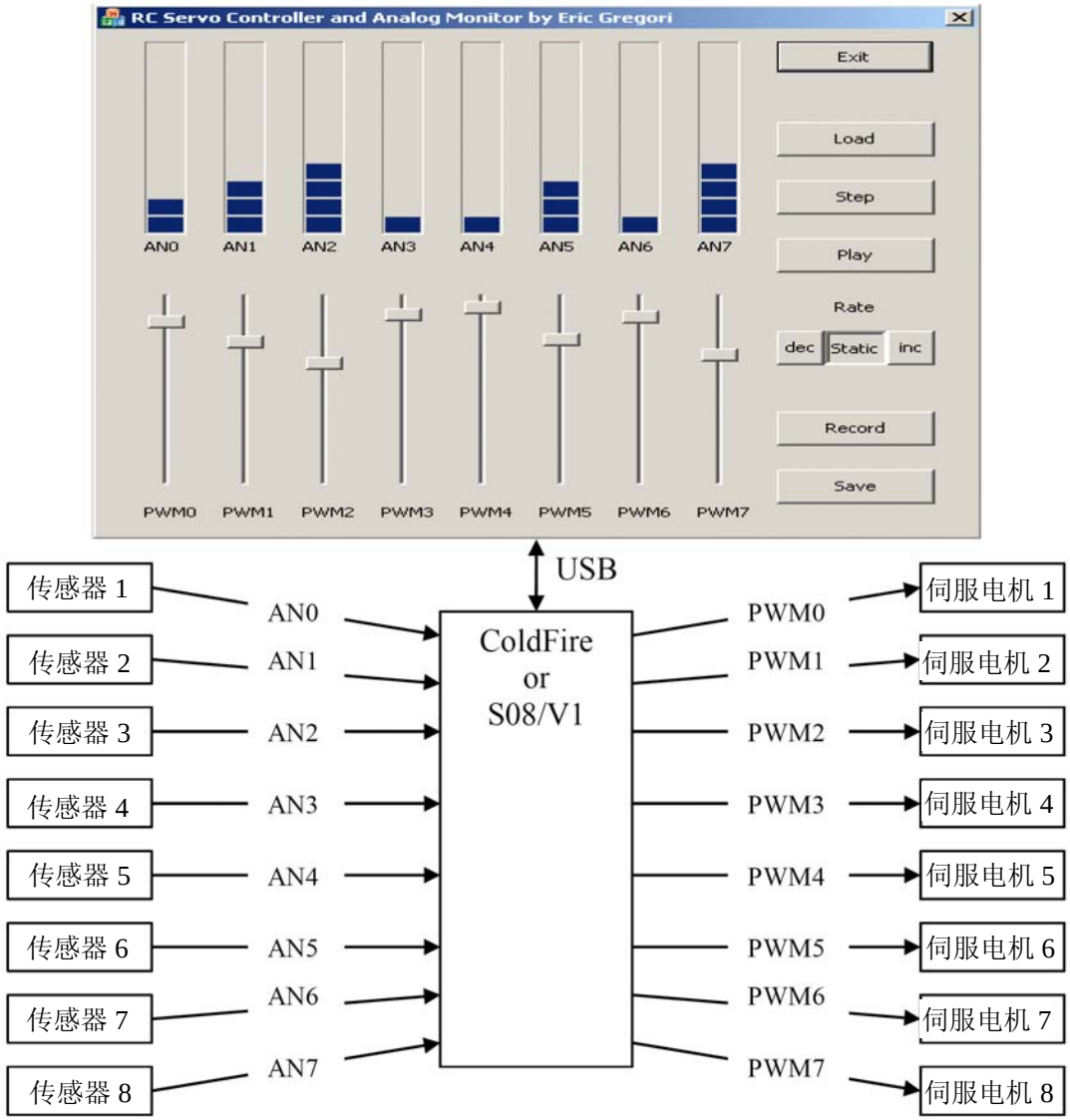
| 3 按键鼠标报告<br>3 字节（5 个填充位） | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 字节 2<br>76543210 | 字节 1<br>76543210 | 字节 0<br>7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Y<br>数据 | X<br>数据 | | | | | | B3<br>中 | B2<br>右 | B1<br>左 |

图2 鼠标数据结构

# 3 USB RC Servo （PWM）控制器和模拟监控器

RC Servo 控制器演示了自定义ID设备报告描述符的使用，并且PC（主机）

方软件需要与自定义设备通信。通过USB最多能可以控制8个RC 伺服电机。设备方固件也要支持返回8字节的模拟数据给PC方。

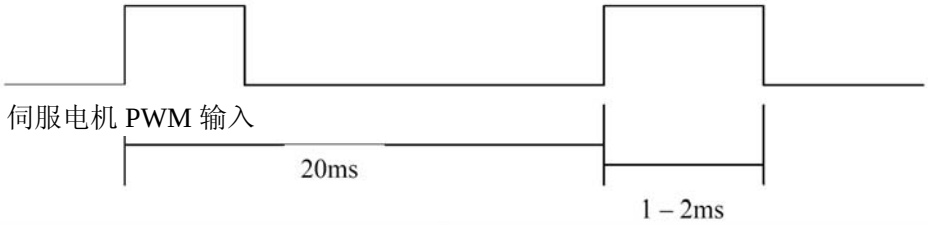



## 3.1 使用PWM控制器控制一个RC Servo

RC Servo（远程控制 Servo）设备用来控制非专业领域的机械设备。远程控制汽车，卡车，船和机器人等案例中都是用RC Servos来控制舵、油门、l腿和胳膊。RC Servo将一个PWM信号转换成一个机械动作。正如名字所示，它是一个有反馈和闭环控制的伺服装置。PWM信号设置一个介于-90和90度之间的位置，机械上伺服装置锁定位置值在-90到90之间。

控制RC Servo的PWM信号最小电压峰值为3.3伏（有些伺服装置要求高达5伏），周期为20ms（不需要那么精确，但因该大于15ms和小于50ms）和一个介

于1ms和2ms之间中心为1.5ms的高电平时间（high time）。




| PWM 高电平时间 | 伺服电机位置（角度） |
|---|---|
| ~1ms | -90 degrees |
| 1.5ms | 0 degrees |
| ~2ms | 90 degrees |

图3 伺服电机位置与PWM高电平的关系

对于PWM高电平时间范围为1ms和180度的机械旋转，伺服装置（servo）分辨大约为5.6微妙每度。RC Servos实际的分辨度依赖于伺服设备。RC 伺服装置为主要的模拟设备（也有数字Servos可用）所以具体规格有一些偏差。

在这种设计下要支持8个Servos。此需求要求ColdFire PWM控制器能够用于8位的8通道模式下。仅仅使用一个8位计数器，20ms的周期能产生78us（14度）的分辨度。在大多数应用中这不可以接受，所以需要设计一方法在20ms周期内的提供更好分辨度（低于5度）。

### 3.1.1 在“one shot”模式中使用ColdFire PWM控制器

ColdFire PWM控制器使用双缓冲机制来消除PWM流中的假信号（glitches）。这种机制用来产生“one shot”模式。在这种模式下，PWM控制器仅仅产生单脉冲。这允许八位PWM计数器在脉冲内使用而不是跨越整个周期。

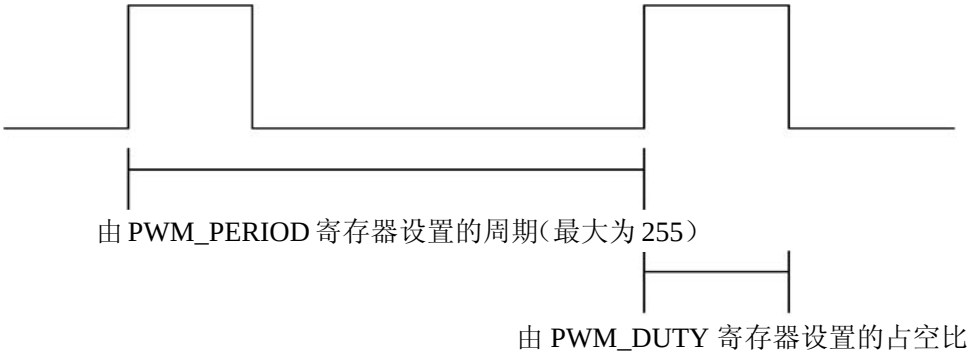



图4 标准PWM配置-极限分辨度为整个周期内255次计数

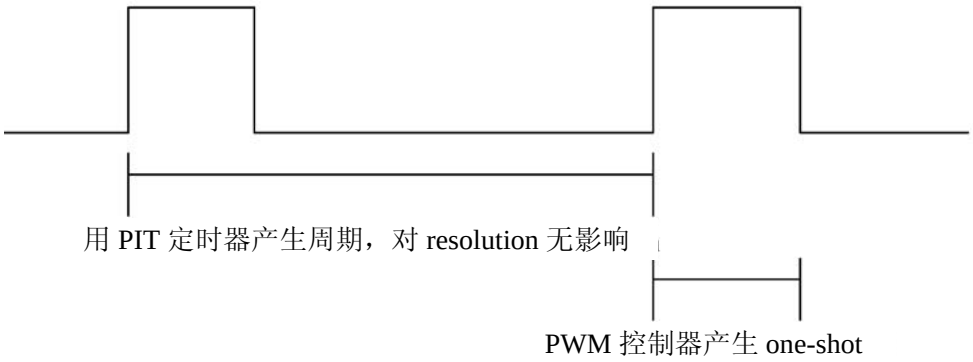
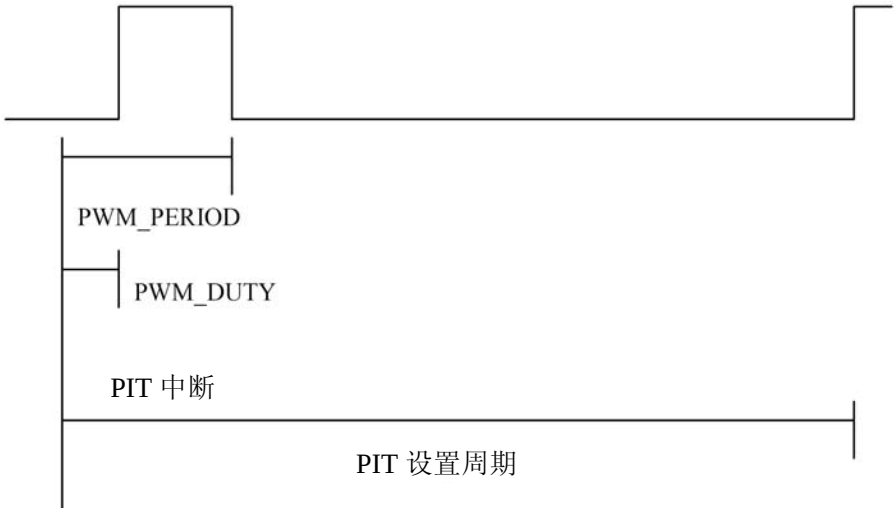


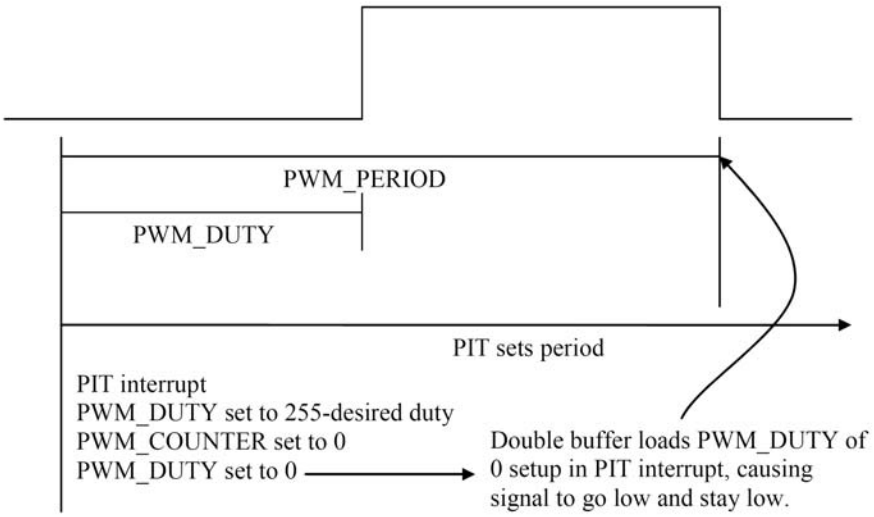

图5 one-shot模式使用PIT定时器来产生PWM周期，PWM模块产生脉冲

使用one-shot模式，高电平时间显著地增加了。8位的PWM计数器完全用来产生高脉冲，同时使用一个单独的PIT定时器来产生PWM周期。在每一个PWM周期中PIT定时器产生一个中断，此时PWM控制器则产生一个one-shot脉冲。通过设置PWM_DUTY寄存器设置占空比、复位PWM_CONTER，然后立即将PWM_DUTY设为0来产生“one-shot”脉冲。由于双缓冲PWM_DUTY=0的设置并不能马上起效，相反当PWM_RERIOD寄存器时间到时才有效。




PWM_PERIOD=PWM 控制器中的 Period 寄存器
PWM_DUTY=PWM 控制器中的 DUTY cycle 寄存器
PIT=可编程中断定时器（定时器中断）

图6 PWM的配置




事实上PWM占空比寄存器设置为255（想要的占空比）。通过配置PWM控制器来产生一个低的持续的脉冲。这将导致每一PWM_PWRIOD后PWM控制器将信号设置为0。

### 3.1.2 PIT中断处理

```
__declspec(interrupt:0) void PIT1_isr(void)
{
    // disable PWM channels
    // This avoids any possible glitches since we do not
    // know the value of the PWM counter
    MCF_PWM_PWME = 0;
    // Write duty cycle
    // Set PWM_DUTY to 255-desired duty cycle
    MCF_PWM_PWMDTY4 = MCF_PWM_PWMDTY_DUTY(255-pwm_duty[0]);
    MCF_PWM_PWMDTY6 = MCF_PWM_PWMDTY_DUTY(255-pwm_duty[1]);
    // Reset counter, loads duty cycle
    MCF_PWM_PWMCNT4 = 0;
    MCF_PWM_PWMCNT6 = 0;
    // enable output
    MCF_PWM_PWME = 0x50;
    // Force to 0 after PWM_PERIOD – creating one-shot
    MCF_PWM_PWMDTY4 = MCF_PWM_PWMDTY_DUTY(0);
    MCF_PWM_PWMDTY6 = MCF_PWM_PWMDTY_DUTY(0);
    /* Clear interrupt at CSR */
    MCF_PIT_PCSR(1) |= MCF_PIT_PCSR_PIF;
}
```

## 3.2 读取A/D转换

该工程中使用的A/D驱动与2.1节（“使用A/D转换来读取加速度传感器”）描述的相同。

## 3.3 HID报告概述

在2.2节，“HID鼠标报告概述”中，有完整的报告描述符的描述。在该工程中我们需要一自定义的报告描述符。使用报告描述符的工具创建了一支持8字节IN和OUT传输的报告描述符。http://www.usb.org/developers/hidpage/dt2_4.zip上可以找到该工具。

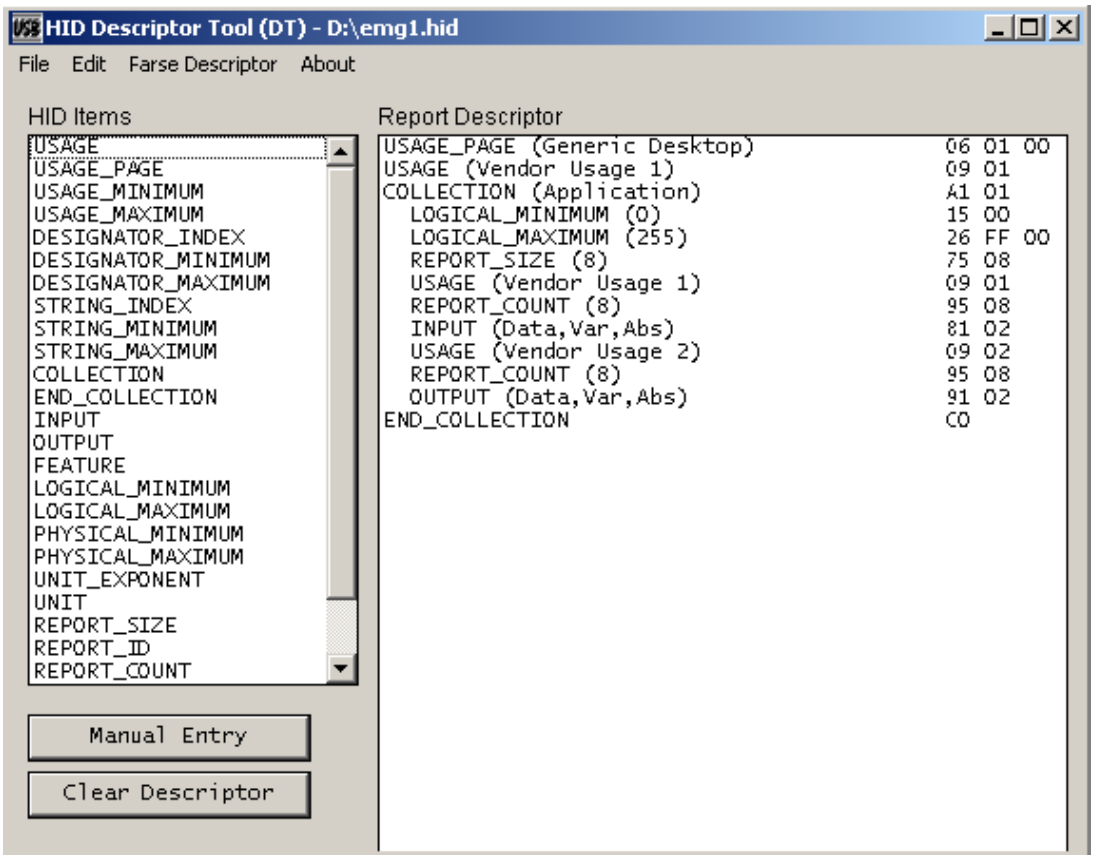



图7 8字节的IN/OUT自定义报告

### 3.3.1 自定义报告描述符概述

LOGICAL_MINMUM(0)

LOGICAL_MAXIMUM(255)-数据范围

REPORT_SIZE(8)-8位/数据块

REPORT_COUNT(8)- 输入的8数据块和输出数据的8个

此报告将传输8字节的IN数据和8字节的OUT数据。发送给PC的8字节数据用于传输模拟转换结果，8字节的OUT数据用于调整PWM占空比。

## 3.4 固件

```
void hid_generic(void)
{
    hcc_u8 out_report;
    hcc_u8 in_report;
    pwm_duty[0] = 78;
    pwm_duty[1] = 156;
    pwm_duty[2] = 156;
    pwm_duty[3] = 78;
    PIT_mode = 'P';
    init_PWM();
    init_PIT1();
    HID_init(500, 0);
    out_report=hid_add_report(rpt_out, 0, 8);
    in_report=hid_add_report(rpt_in, 0, 8);
    LED4_ON;
    while(!device_stp)
    {
      hid_process();
      /* Send switch status. */
      if (!hid_report_pending(in_report))
      {
        hcc_u8 tmp[8];
        if( period_counter > 1 )
        {
          period_counter = 0;
          tmp[0] = (unsigned char)((read_AD( 0 )>>4)&0x00ff);
          tmp[1] = (unsigned char)((read_AD( 1 )>>4)&0x00ff);
          tmp[2] = (unsigned char)((read_AD( 2 )>>4)&0x00ff);
          tmp[3] = (unsigned char)((read_AD( 3 )>>4)&0x00ff);
          tmp[4] = (unsigned char)((read_AD( 4 )>>4)&0x00ff);
          tmp[5] = (unsigned char)((read_AD( 5 )>>4)&0x00ff);
          tmp[6] = (unsigned char)((read_AD( 6 )>>4)&0x00ff);
          tmp[7] = (unsigned char)((read_AD( 7 )>>4)&0x00ff);
          hid_write_report(in_report, (unsigned char *)&tmp);
        }
      }
      /* Set status leds if needed. */
      if (hid_report_pending(out_report))
      {
```

```
            hcc_u8 data[9];
            hid_read_report(out_report, (unsigned char *)&data);
            pwm_duty[0] = data[0];
            pwm_duty[1] = data[1];
            pwm_duty[2] = data[2];
            pwm_duty[3] = data[3];
        }
        busy_wait();
    }
}
```

## 3.5 PC主机应用

```
// Send slider data to USB driver
static void send_pwm(void)
{
    unsigned char lstate[8];
    lstate[0] = theApp.dlg->pwm0.GetPos();
    lstate[1] = theApp.dlg->pwm1.GetPos();
    lstate[2] = theApp.dlg->pwm2.GetPos();
    lstate[3] = theApp.dlg->pwm3.GetPos();
    lstate[4] = theApp.dlg->pwm4.GetPos();
    lstate[5] = theApp.dlg->pwm5.GetPos();
    lstate[6] = theApp.dlg->pwm6.GetPos();
    lstate[7] = theApp.dlg->pwm7.GetPos();
    HIDWrite(&lstate);
}
//Update bargraphs with analog data
static void get_analog(void)
{
    unsigned char lstate[32];
    if (HIDRead(&lstate))
    {
        theApp.dlg->an0.SetPos( lstate[0] );
        theApp.dlg->an1.SetPos( lstate[1] );
        theApp.dlg->an2.SetPos( lstate[2] );
        theApp.dlg->an3.SetPos( lstate[3] );
        theApp.dlg->an4.SetPos( lstate[4] );
        theApp.dlg->an5.SetPos( lstate[5] );
        theApp.dlg->an6.SetPos( lstate[6] );
        theApp.dlg->an7.SetPos( lstate[7] );
    }
}
```

# 4 USB主控枚举嗅探器(主机驱动示例)

CMX协议栈同时支持主机和设备方。通过展示设备枚举的过程演示了协议栈的主机方。枚举是从设备到主机的数据结构（描述符）的传输。此过程发生在设备接入主机时。www.usb.org上的USB2.0规范定义了枚举和不同的描述符。AN3492（“USB及使用CMX协议栈”）中也彻底的描述枚举以及描述符，其中还有主机固件API的说明。

使用此固件，可以在开发板上连接不同的设备，并且显示设备发给主机的描述符。使用USB2.0规范中定义的描述符，可以理解该描述符的具体含义。

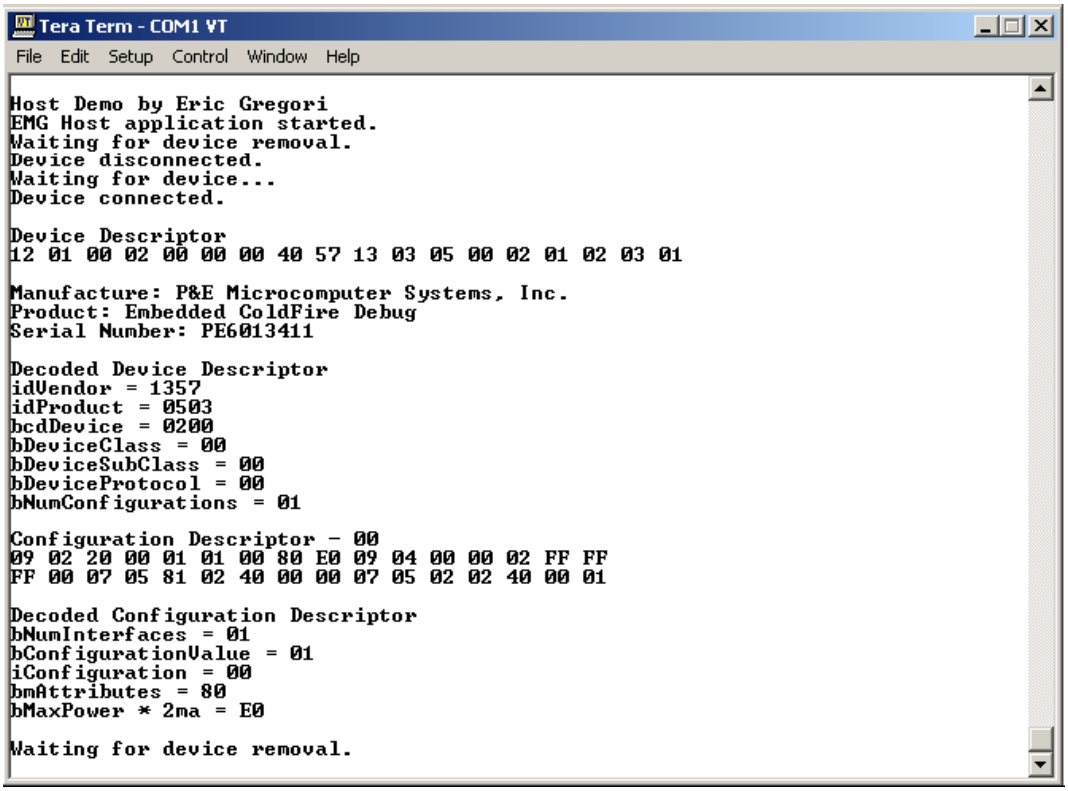



图8 从插入ColdFire调试器（M52221DEMO）得到的探测结果

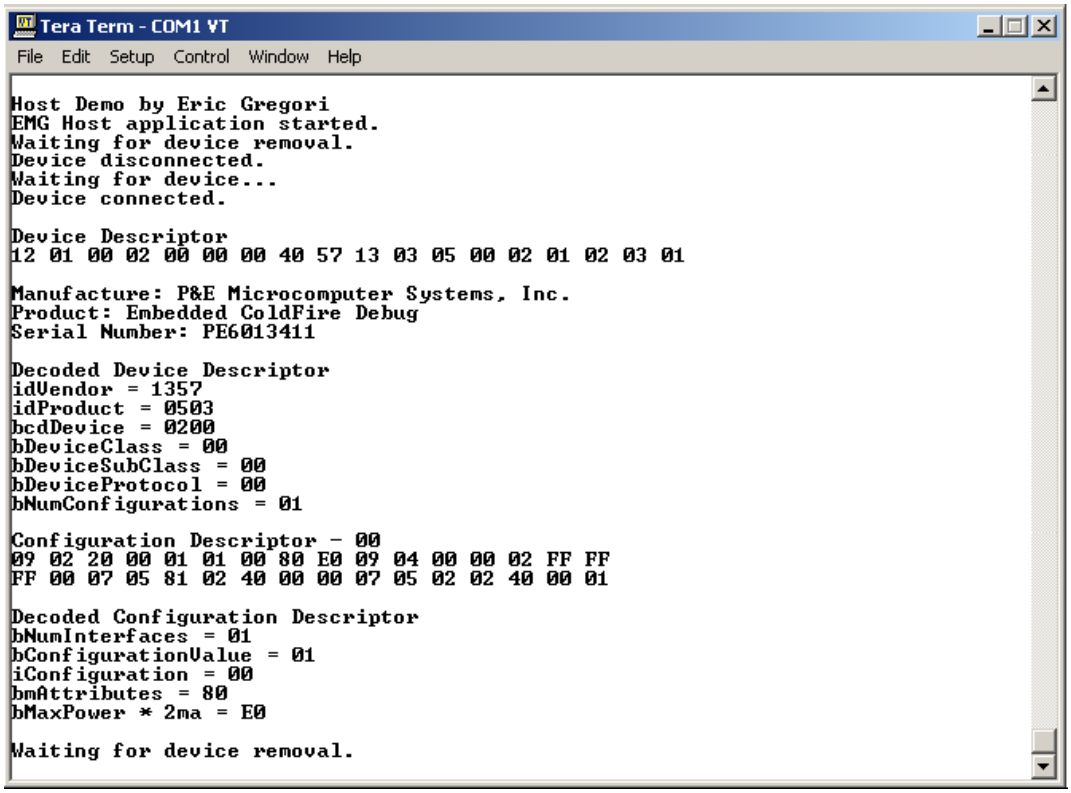



图9 从增加的一个USB摄像机得到的探测结果

## 4.1 固件

### 4.1.1 emg_host_demo()

下列代码是一个怎样使用主机API来枚举设备的例子。在这个例子中，此固件通过串口（38400，8，n，1）打印出设备和配置描述符。一个新函数（不是标准栈的一部分）用来请求设备上的字符串描述符。

```
//
// Enumerate device, and output device / configuration descriptors to the serial port in hex
// Serial descripors are also printed to the serial port
//
// Written by Eric Gregori(847) 651 - 1971
//
int main(void)
{
    hcc_u8 cfg, str1, str2, str3;
    hcc_u16 length, i;
    device_info_t dev_inf;
    cfg_info_t cfg_inf;
    hw_init();
```

```
uart_init(38400, 1, 'n', 8);
host_init();
print( "\r\nHost Demo by Eric Gregori\n\r" );
print("EMG Host application started.\r\n");
while(1)
{
    busy_wait();
    /* a device is already connected, wait till it is disconnected */
    print("Waiting for device removal.\r\n");
    while(host_has_device()); // Spin waiting for !ATTACH
    print("Device disconnected.\r\n");
    /* At this point no device is attached. Wait till attachment. */
    print("Waiting for device...\r\n");
    while(!host_scan_for_device()); // Spin waiting for ATTACH
    print("Device connected.\r\n");
    // Read and parse device descriptor
    // get_device_info() calls get_dev_desc()
    if( !get_device_info(&dev_inf) )
    {
        print( "\n\rDevice Descriptor\n\r" );
        for( i=0; i<18; i++ )
        {
            emg_printbytehex( dbuffer[i] );
            print( " " );
        }
        print( "\n\r" );
        str1 = dbuffer[14];
        str2 = dbuffer[15];
        str3 = dbuffer[16];
        if( str1 )
        {
            print( "\r\nManufacture: " );
            emg_print_str_desc( str1 );
        }
        if( str2 )
        {
            print( "\r\nProduct: " );
            emg_print_str_desc( str2 );
        }
        if( str3 )
        {
            print( "\r\nSerial Number: " );
            emg_print_str_desc( str3 );
        }
```

```
        print( "\r\n\r\nDecoded Device Descriptor" );
        print( "\n\ridVendor = " );
        emg_printwordhex( dev_inf.vid );
        print( "\n\ridProduct = " );
        emg_printwordhex( dev_inf.pid );
        print( "\n\rbcdDevice = " );
        emg_printwordhex( dev_inf.rev );
        print( "\n\rbDeviceClass = " );
        emg_printbytehex( dev_inf.clas );
        print( "\n\rbDeviceSubClass = " );
        emg_printbytehex( dev_inf.sclas );
        print( "\n\rbDeviceProtocol = " );
        emg_printbytehex( dev_inf.protocol );
        print( "\n\rbNumConfigurations = " );
        emg_printbytehex( dev_inf.ncfg );
        print( "\n\r" );
    }
    else
        print( "\r\nFailure Reading Device Descriptor" );
        // Read all configuration descriptors
        for(cfg=0; cfg < dev_inf.ncfg; cfg++)
        {
            // get the configuration descriptor
            if (get_cfg_desc(cfg))
                continue; // Descriptor cfg not found
            else
            {
                print( "\n\rConfiguration Descriptor - " );
                emg_printbytehex( cfg );
                length=RD_LE16(dbuffer+2);
                for( i=0; i<length; i++ )
                {
                    if( (i%16) == 0 )
                    print( "\n\r" );
                    emg_printbytehex( dbuffer[i] );
                    print( " " );
                }
                str1 = dbuffer[6];
                print( "\n\r\n\rDecoded Configuration Descriptor" );
                // Call get_cfg_info() to parse configuration descriptor
                get_cfg_info( &cfg_inf );
                print( "\n\rbNumInterfaces = " );
                emg_printbytehex( cfg_inf.nifc );
                print( "\n\rbConfigurationValue = " );
```

```
                        emg_printbytehex( cfg_inf.ndx );
                        print( "\n\riConfiguration = " );
                        emg_printbytehex( cfg_inf.str );
                        print( "\n\rbmAttributes = " );
                        emg_printbytehex( cfg_inf.attrib );
                        print( "\n\rbMaxPower * 2ma = " );
                        emg_printbytehex( cfg_inf.max_power );
                        if( str1 )
                        {
                            print( "\r\nManufacture: " );
                            emg_print_str_desc( str1 );
                        }
                    }
                    print( "\n\r\n\r" );
                } // end of config descriptor read
                //
    } // while(1)
}
```

### 4.1.2 显示一个字符串描述符-emg_print_str_desc()

```
void emg_print_str_desc( unsigned char desc )
{
    unsigned chari;
    if( !emg_get_str_descriptor( desc ) )
    {
      // Unicoded string is in dbuffer starting at 2
      // strlen = (dbuffer[0] - 2)*2
      for( i=2; i<=(dbuffer[0]-2); i+=2 )
      uart_putch( dbuffer[i] );
    }
}
```

### 4.1.3 emg_get_str_descriptor()

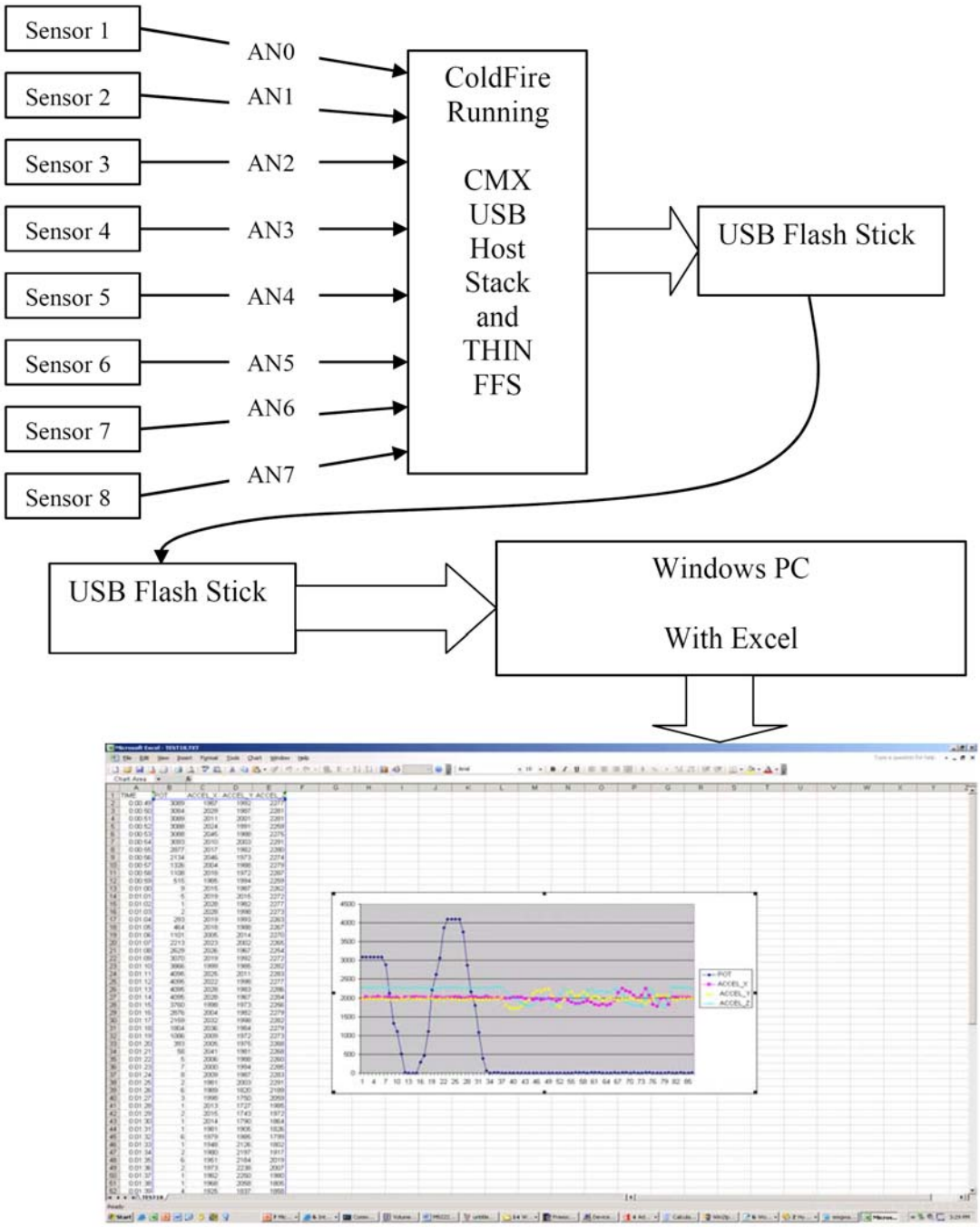
```
int emg_get_str_descriptor( unsigned char desc )
{
    hcc_u8 setup[8];
    hcc_u16 length=3;
    hcc_u8 retry=3;
    std_error=stderr_none;
    do
    {
      // Build SETUP data packet
      fill_setup_packet(setup,STP_DIR_IN,STP_TYPE_STD,STP_RECIPIENT_DEVICE,
```

```
                        STDRQ_GET_DESCRIPTOR,
                        (hcc_u16)((STDDTYPE_STRING<<8)|desc), 0, length);
        if (length == host_receive_control(setup, dbuffer, 0))
        {
          /* Check returned descriptor type and length (ignore extra bytes) */
          if ((USBDSC_TYPE(dbuffer) == STDDTYPE_STRING))
          {
            length=dbuffer[0];
            if( length >= DBUFFER_SIZE )
                length = DBUFFER_SIZE-1;
            // Rebuild SETUP data packet with new length
            fill_setup_packet(setup, STP_DIR_IN,
STP_TYPE_STD,STP_RECIPIENT_DEVICE,
                              STDRQ_GET_DESCRIPTOR,
                              (hcc_u16)((STDDTYPE_STRING<<8)|desc),
                              0, length);
            if (length == host_receive_control(setup, dbuffer, 0))
                return(0);
          }
        }
      }while(retry--);
      std_error=stderr_host;
      return(1);
}
```

# 5 基于USB闪存棒的模拟数据记录器（Logger）(大容量存储类)

CMX USB主机协议栈有读取和写闪存棒的能力。CMX USB协议栈写入到闪存棒的数据能够在不需要额外软件安装在PC上的情况下从PC方读取到。PC写入到闪存棒的数据，可以被CMX USB协议栈读取到。

固件以标准ASCII码的格式将模拟数据记录到闪存棒中。可以被 Microsoft Excel导出。固件读取4路模拟通道（很容易扩展到8通道）和为每个数据集增加时间戳。




## 5.1 大容量存储类

USB大容量存储类由“Universal Serial Bus Mass Storage Class Specification Overview（revision 1.2）”指定。该规范可以在www.usb.org找到。大容量存储类把USB包组包在ATAPI(Advanced Technology Attachment Packet Interface)和SCSI(Small Computer System Interface)命令集中。

## 5.2 命令集

表2 子类码/命令集

| 子类码 | 命令块规范 | 典型应用 |
|---|---|---|
| 1 | Reduced Block Commands(RBC) | 闪存设备 |
| 2 | SFF-8020i,MMC-2(ATAPI) | CD/DVD |
| 3 | QIC-157 | 磁带设备 |
| 4 | UFI | 阵列设备 |
| 5 | SFF-8070i | 阵列设备 |
| 6 | SCSI | |

## 5.3 USB传输机制（BULK）

大容量存储类使用BULK传输传输数据到大容量存储设备或从设备读取数据。BULK传输是最大有效数据为64字节的握手的传输。每帧中传输可以大于1，最大为19，最大传输速度为1216字节/豪秒或者1216000字节/秒。

BULK传输特征：

1．“可利用的带宽”访问USB

2．传输重发（握手）

3．数据传输保证但不保证延时

对于带宽分配BULK传输是最低优先级的传输。

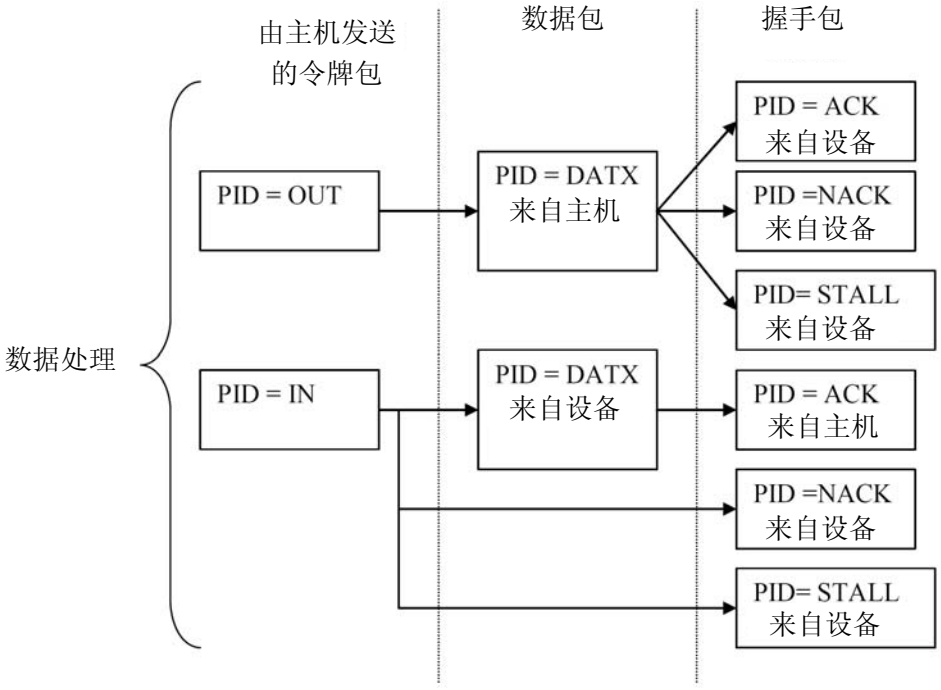



图10 BULK传输事务

## 5.4 CMX THIN 闪存文件系统

CMX THIN闪存文件系统为资源有限的嵌入式系统设计的。THIN文件系统支持FAT12，FAT16，和FAT32文件系统。THIN文件系统是一位于大容量存储类之上的软件层，为用户应用提供标准的API。CMX_FFS_THIN_5222x.pdf详细描述了CMX THIN flash 文件系统及其API，该文件位于USB协议栈的docs目录下。

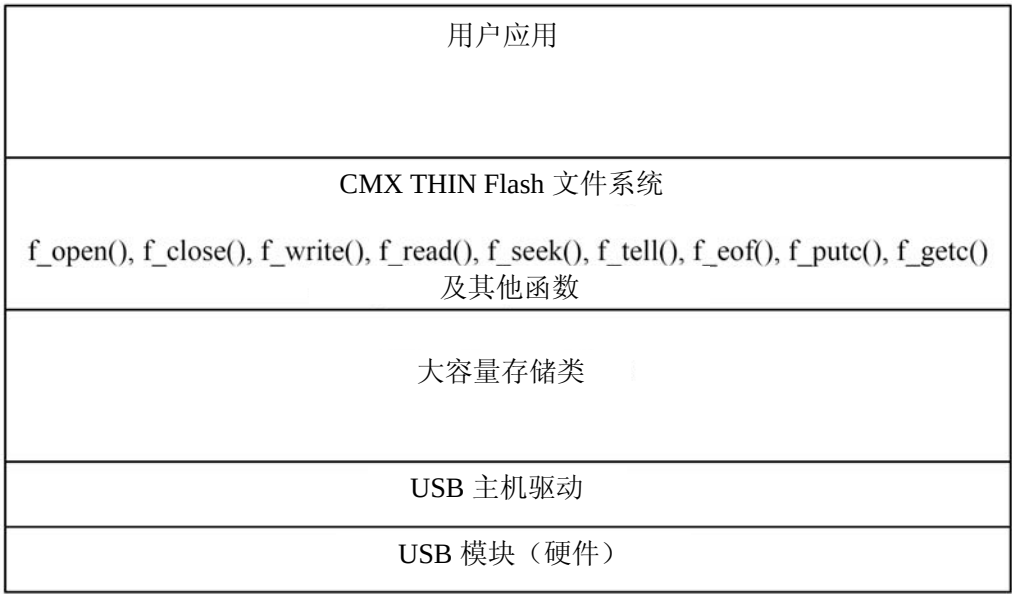



图11 大容量存储类USB协议栈

## 5.5 使用A/D转换

A/D转换被用来为数据记录器（logger）采样模拟通道。它设置为连续模式。在每一个采样周期后数据记录器通过简易驱动来简单地读取A/D数据。不尝试与A/D通道同步。实际上，固件某一时刻读取每个A/D通道一次，在两次读取之间将二进制转换为十进制并写入到flash。这对于实际使用的数据记录器来说是不可接收的，但该采样工程可以很容易地修改以支持必需的同步。A/D转换内容包含在2.1节中。

## 5.6 使用RTC

实时时钟（RTC）用于为每次采样提供“时间戳”。下列代码为MCF5222X(M52223EVB和M52221DEMO)中RTC模块的代码。RTC时钟由CPU的主时钟分频而来。RTC以二进制的方式提供时，分和秒。

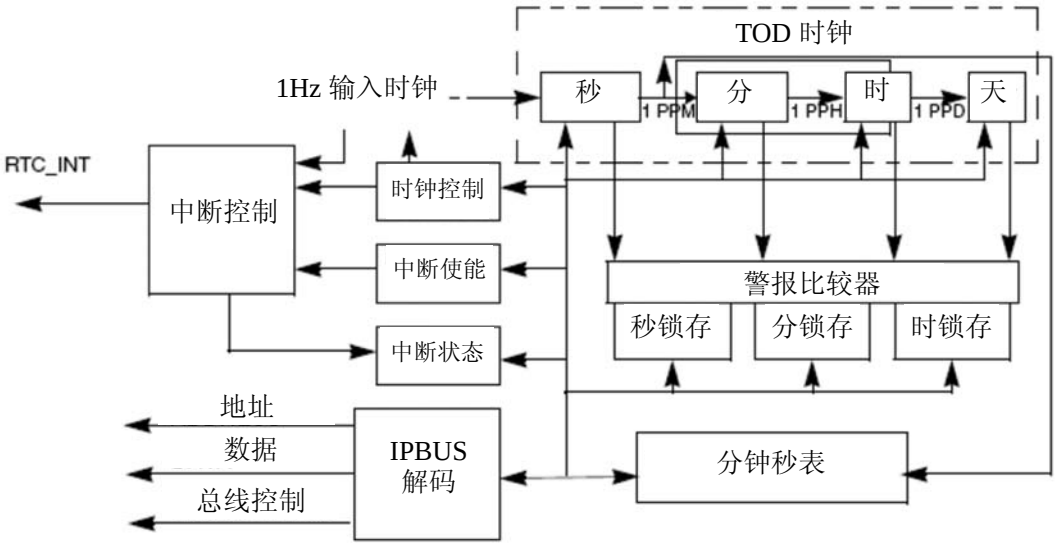



图11 大容量存储类USB协议栈

## 5.6.1 配置RTC

RTC需要一1HZ的时钟输入。该时钟从系统时钟分频而来。该分频系数由RTCDR寄存器来配置。

表3 MCF_CLOCK_RTCDR寄存器

| 字段 | 描述 |
|---|---|
| 31—0<br>RTCDF | 实时时钟分频因子。此字段通过一个RTCDR+1（从1到4294967296）的因子来从系统时钟为实时时钟模块分频。 |

用于输入分频的“系统时钟”实际上是PLL的输入（INPUT），分频器（divider）的输入为晶振的频率。

/* Set real time clock freq-Oscillator clock(crystal frequency) = 48MHz */

MCF_CLOCK_RTCDR = 48000000-1;

## 5.6.2 读取RTC

读取RTC需要读取时，分和秒寄存器。

```
//
// Author: Eric Gregori (847) 651 - 1971
//
// Read time from ColdFIre RTC
//
unsigned char get_time( unsigned char type )
{
    switch( type )
```

```
    {
        case 'H':
            return( (unsigned char)((MCF_RTC_HOURMIN & 0x00001F00)>>8 ));
        case 'M':
            return( (unsigned char)(MCF_RTC_HOURMIN & 0x0000003F ));
        case 'S':
            return( (unsigned char)(MCF_RTC_SECONDS & 0x0000003F ));
      }
      return( 0 );
}
```

## 5.7 数据记录器（logger）固件

### 5.7.1 unsigned char write_log(F_FILE *file, unsigned char c)

```
//
// Author: Eric Gregori (847) 651 - 1971
//
// Write raw byte to file, and output to screen
//
unsigned char write_log( F_FILE*file, unsigned char c )
{
    uart_putch( c );
    return( (unsigned char)(c != (unsigned char)f_putc( (int)c, file ) ));
}
```

### 5.7.2 unsigned char write_log_dec(F_FILE *file, unsigned short d)

```
//
// Author: Eric Gregori (847) 651 - 1971
//
// Write decimal value to file, and to screen
//
unsigned char write_log_dec( F_FILE*file, unsigned short d )
{
    unsigned charth, h, t, o, ret;
    unsigned short c;
    c = d;
    for( th=0; th<9;)
    {
        if( c >= 1000 )
        {
            c -= 1000;
            th++;
        } else
```

```
                break;
        }
        for( h=0; h<9; )
        {
                if( c >= 100 )
                {
                        c -= 100;
                        h++;
                } else
                        break;
        }
        for( t=0; t<9; )
        {
                if( c >= 10 )
                {
                        c -= 10;
                        t++;
                } else
                        break;
        }
        o = (unsigned char)c;
        ret = 0;
        if( th )
                ret = write_log( file, (unsigned char)(th+0x30));
        if( !ret && (th || h ) )
                ret = write_log( file, (unsigned char)(h+0x30));
        if( !ret && (th || h || t) )
                ret = write_log( file, (unsigned char)(t+0x30));
        if( !ret )
                ret = write_log( file, (unsigned char)(o+0x30));
        return( ret );
}
```

## 5.7.3 unsigned char write_log_string(F_FILE *file, unsigned char *data)

```
//
// Author: Eric Gregori (847) 651 - 1971
//
unsigned char write_log_string( F_FILE *file, unsigned char *data )
{
        unsigned char i, ret;
        ret = 0;
        for( i=0; (!ret && data[i]); i++ )
                ret = write_log( file, data[i] );
return( ret );
```

```
}
```

## 5.7.4 void cmd_emglog(char *param)

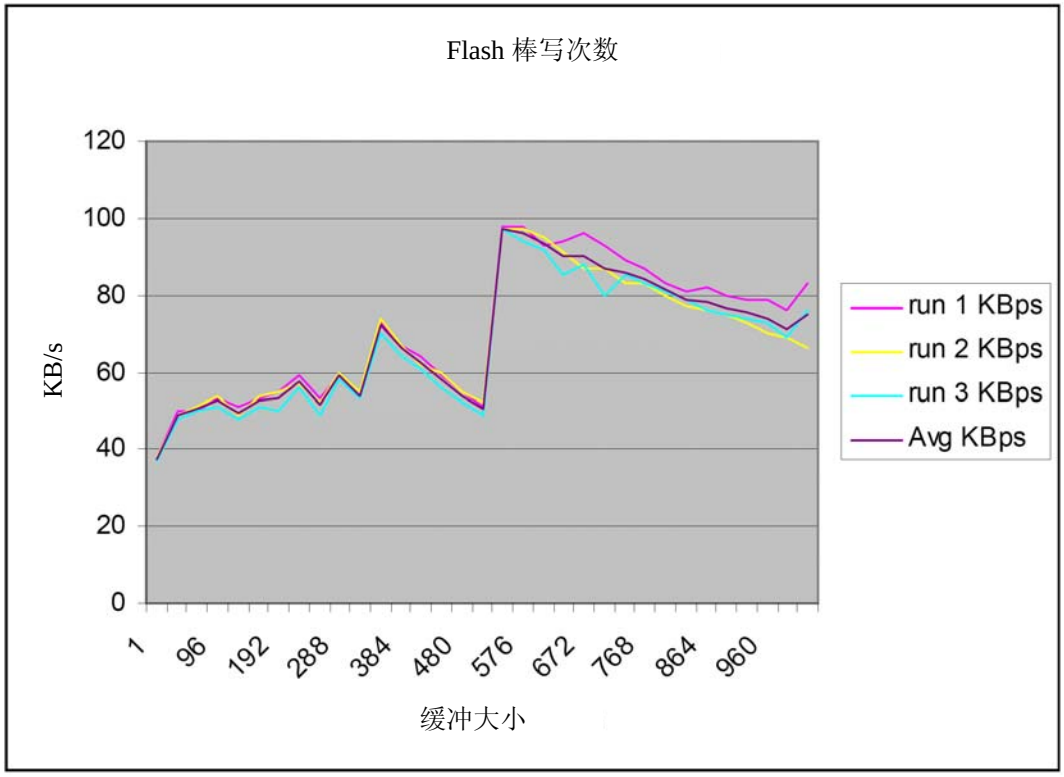
```
//
// Author: Eric Gregori (847) 651 - 1971
//
// Log analog data to file in comma delimited format
//
void cmd_emglog( char *param)
{
	F_FILE *file;
	hcc_u8 c, osec;
	hcc_u16 ad;
	print( "\n\nHit enter to quit\n\n" );
	file=f_open(param, "w");
	if (file == 0)
	{
		print("Failed to open ");
		print(param);
		print(".\r\n");
		return;
	}
	print(".\r\n");
	(void)write_log_string( file,
			(unsigned char *)("TIME,POT,ACCEL_X, ACCEL_Y, ACCEL_Z\r\n" ));
	while(1)
	{
		if( osec == get_time('S') )
		   continue;
		osec = get_time('S');
		c=get_time('H');
		if( write_log_dec(file, (unsigned short)c) ) break;
		if( write_log(file, ':' )) break;
		c=get_time('M');
		if( write_log_dec(file, (unsigned short)c) ) break;
		if( write_log(file, ':' )) break;
		c=get_time('S');
		if( write_log_dec(file, (unsigned short)c) ) break;
		if( write_log(file, ',' )) break;
		ad = (unsigned short)read_AD( 0 );
		if( write_log_dec(file, ad) ) break;
		if( write_log(file, ',' )) break;
		ad = (unsigned short)read_AD( 4 );
		if( write_log_dec(file, ad) ) break;
```

```
            if( write_log(file, ',' )) break;
            ad = (unsigned short)read_AD( 5 );
            if( write_log_dec(file, ad) ) break;
            if( write_log(file, ',' )) break;
            ad = (unsigned short)read_AD( 6 );
            if( write_log_dec(file, ad) ) break;
            if( write_log(file, '\r' )) break;
            if( write_log(file, '\n' )) break;
            if (uart_input_ready())
            {
                  break;
            }
      }
      f_close( file );
      print( "\nFile Closed" );
      return;
}
```

## 5.8 数据写入的速度

为了测试闪存棒的写入时最大数据传输速度，写了一个简单的固件。此固件的功能是在1秒内以尽可能快的速度将数据项写到闪存棒中。其结果将通过串口输出。

使用许多大小不同的数据项以测试其性能。正如所料，数据项越大，效果越好。假定一个包大小为64字节，BULK传输的USB2.0规格表明最大的理论数据传输速度为1187.5Kb/S。每一包仅为64字节的数据，最关键的变化是转移是一帧内所能完成的事务数量。每个事务携带了64字节的数据。



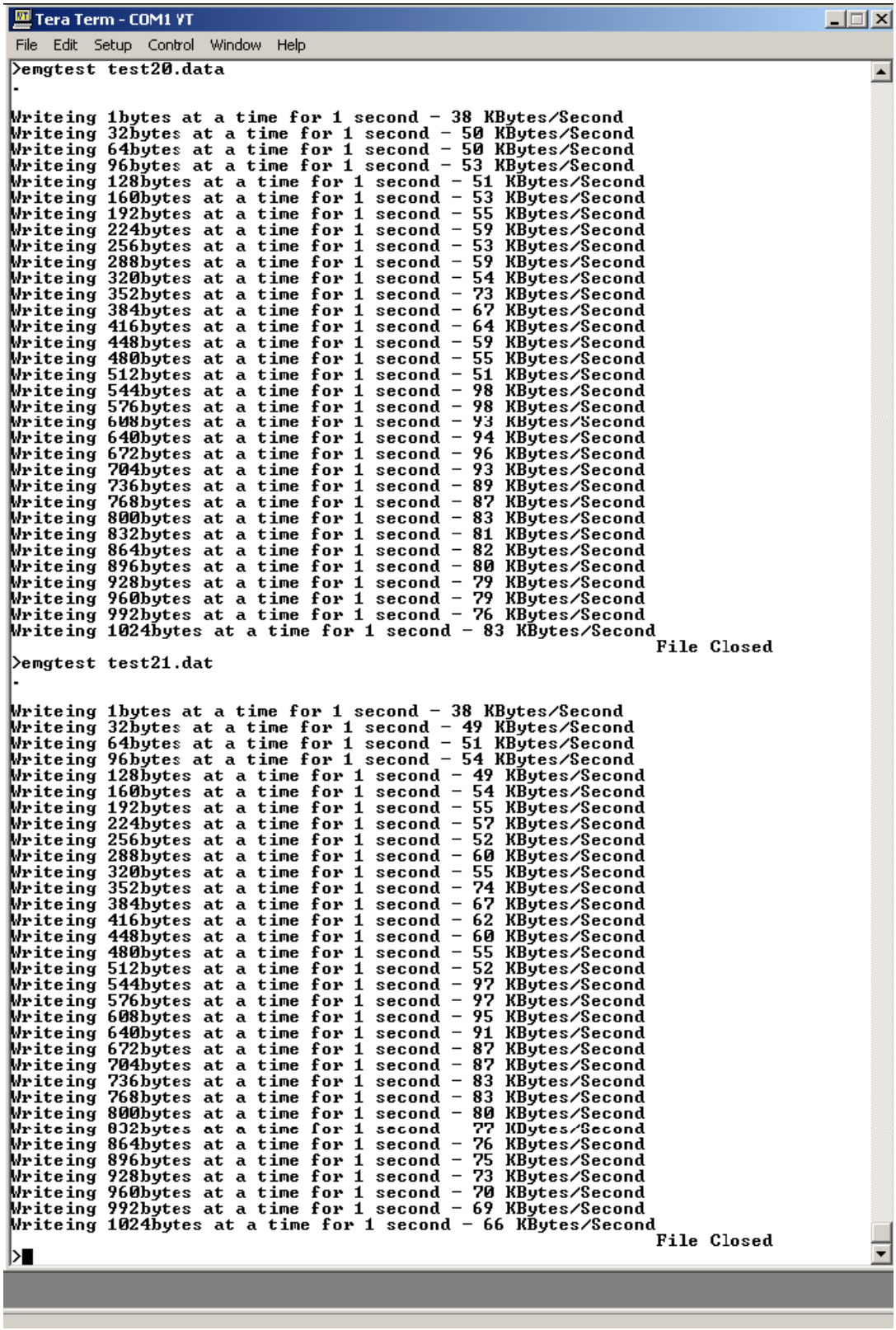

### 5.8.1 emgtest<文件名>命令

向写文件<文件名>中尽可能快的写不同大小的项。

使用 f_write(buff, size, 1, file)来写文件。

其中：buff是一个未初始化的内存区域（写到闪存棒的数据）。

size是要写的数据字节数/调用 f_write()-“数据项大小”。


Tera Term - COM1 VT
File Edit Setup Control Window Help
>emgtest test20.data
Writeing 1bytes at a time for 1 second - 38 KBytes/Second
Writeing 32bytes at a time for 1 second - 50 KBytes/Second
Writeing 64bytes at a time for 1 second - 50 KBytes/Second
Writeing 96bytes at a time for 1 second - 53 KBytes/Second
Writeing 128bytes at a time for 1 second - 51 KBytes/Second
Writeing 160bytes at a time for 1 second - 53 KBytes/Second
Writeing 192bytes at a time for 1 second - 55 KBytes/Second
Writeing 224bytes at a time for 1 second - 59 KBytes/Second
Writeing 256bytes at a time for 1 second - 53 KBytes/Second
Writeing 288bytes at a time for 1 second - 59 KBytes/Second
Writeing 320bytes at a time for 1 second - 54 KBytes/Second
Writeing 352bytes at a time for 1 second - 73 KBytes/Second
Writeing 384bytes at a time for 1 second - 67 KBytes/Second
Writeing 416bytes at a time for 1 second - 64 KBytes/Second
Writeing 448bytes at a time for 1 second - 59 KBytes/Second
Writeing 480bytes at a time for 1 second - 55 KBytes/Second
Writeing 512bytes at a time for 1 second - 51 KBytes/Second
Writeing 544bytes at a time for 1 second - 98 KBytes/Second
Writeing 576bytes at a time for 1 second - 98 KBytes/Second
Writeing 608bytes at a time for 1 second - 93 KBytes/Second
Writeing 640bytes at a time for 1 second - 94 KBytes/Second
Writeing 672bytes at a time for 1 second - 96 KBytes/Second
Writeing 704bytes at a time for 1 second - 93 KBytes/Second
Writeing 736bytes at a time for 1 second - 89 KBytes/Second
Writeing 768bytes at a time for 1 second - 87 KBytes/Second
Writeing 800bytes at a time for 1 second - 83 KBytes/Second
Writeing 832bytes at a time for 1 second - 81 KBytes/Second
Writeing 864bytes at a time for 1 second - 82 KBytes/Second
Writeing 896bytes at a time for 1 second - 80 KBytes/Second
Writeing 928bytes at a time for 1 second - 79 KBytes/Second
Writeing 960bytes at a time for 1 second - 79 KBytes/Second
Writeing 992bytes at a time for 1 second - 76 KBytes/Second
Writeing 1024bytes at a time for 1 second - 83 KBytes/Second
File Closed
>emgtest test21.dat
Writeing 1bytes at a time for 1 second - 38 KBytes/Second
Writeing 32bytes at a time for 1 second - 49 KBytes/Second
Writeing 64bytes at a time for 1 second - 51 KBytes/Second
Writeing 96bytes at a time for 1 second - 54 KBytes/Second
Writeing 128bytes at a time for 1 second - 49 KBytes/Second
Writeing 160bytes at a time for 1 second - 54 KBytes/Second
Writeing 192bytes at a time for 1 second - 55 KBytes/Second
Writeing 224bytes at a time for 1 second - 57 KBytes/Second
Writeing 256bytes at a time for 1 second - 52 KBytes/Second
Writeing 288bytes at a time for 1 second - 60 KBytes/Second
Writeing 320bytes at a time for 1 second - 55 KBytes/Second
Writeing 352bytes at a time for 1 second - 74 KBytes/Second
Writeing 384bytes at a time for 1 second - 67 KBytes/Second
Writeing 416bytes at a time for 1 second - 62 KBytes/Second
Writeing 448bytes at a time for 1 second - 60 KBytes/Second
Writeing 480bytes at a time for 1 second - 55 KBytes/Second
Writeing 512bytes at a time for 1 second - 52 KBytes/Second
Writeing 544bytes at a time for 1 second - 97 KBytes/Second
Writeing 576bytes at a time for 1 second - 97 KBytes/Second
Writeing 608bytes at a time for 1 second - 95 KBytes/Second
Writeing 640bytes at a time for 1 second - 91 KBytes/Second
Writeing 672bytes at a time for 1 second - 87 KBytes/Second
Writeing 704bytes at a time for 1 second - 87 KBytes/Second
Writeing 736bytes at a time for 1 second - 83 KBytes/Second
Writeing 768bytes at a time for 1 second - 83 KBytes/Second
Writeing 800bytes at a time for 1 second - 80 KBytes/Second
Writeing 832bytes at a time for 1 second - 77 KBytes/Second
Writeing 864bytes at a time for 1 second - 76 KBytes/Second
Writeing 896bytes at a time for 1 second - 75 KBytes/Second
Writeing 928bytes at a time for 1 second - 73 KBytes/Second
Writeing 960bytes at a time for 1 second - 70 KBytes/Second
Writeing 992bytes at a time for 1 second - 69 KBytes/Second
Writeing 1024bytes at a time for 1 second - 66 KBytes/Second
File Closed
>

## 5.9实时使用

包括文件系统的大容量存储固件是单线程的。固件不使用中断，但使用一个定时器PIT0来跟踪硬件超时。固件自旋（spin）仅当等待USB OTG模块变为可用时。USB主机驱动的核心是文件usb_host.c中的usb_host_start_transaction()函数。

当主机自旋等待硬件完成事务时，USB主机在测试引脚输出高电平。使用如下代码来完成下面的图像：

```
LED3_OFF;
    F_write(buff, 64, 1, file);
    LED3_ON;
```

在下列图像中LED3信号被标为“Start”。最活跃的信号是usb_host_start_transaction()函数中的指示。每一次函数自旋等待一次事务的结，信号变为高电平。usb_host_start_transaction()函数自旋时间为总时间的76%。

**注意**

写USB闪存仅仅使用24%的时间，剩下的时间自旋。

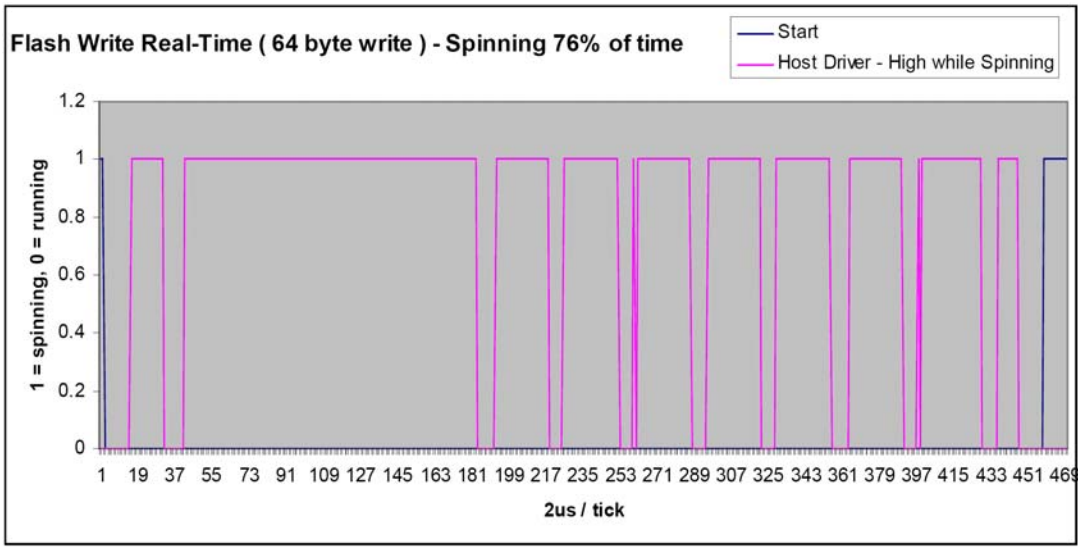


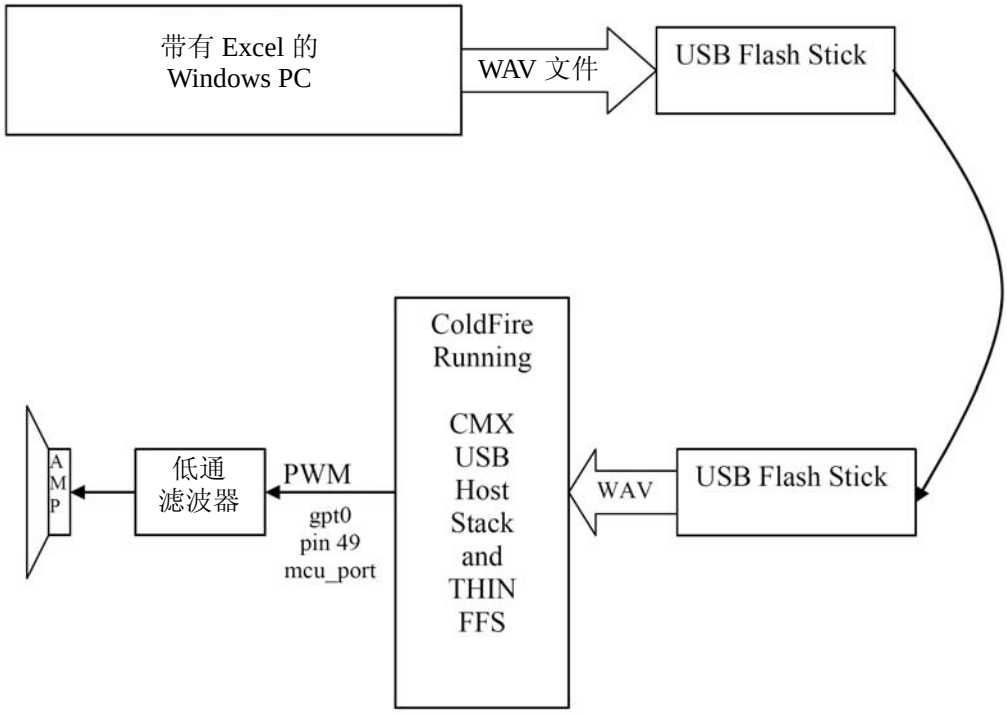

图13 实时分析图

# 6 基于USB闪存棒的WAV播放器和简易文本语音（text to speech）引擎

使用PWM模块作为一简易的数模转换器，音频能够从闪存棒中播放。使用PC将音频转换成WAV格式或者使用录音工具在PC上建立一个WAV文件。将文

件保存到闪存棒中。将闪存棒插入到DEMO和EVB板上，然后使用emgplay<文件名>命令来播放此文件。

目前固件设置使用8位的采样率为8KHZ mono PCM文件来工作。采样速率是基于支持一些旧的音频样本（一个8位的工程（音频超过802.15.4））的决定。USB协议栈可以使用另外的固件轻松地支持更高的播放率，立体声，和压缩。




## 6.1 使用PWM通道的音频应用

部分ColdFire有一个8通道8位的PWM模块，且能够配置为4通道16位的模式。S08(JM60)或者部分基于V1核的mcu通道数依赖于可用的定时器数和每个定时器的通道数。

对于部分ColdFire，我们使用8通道8位模式下PWM模块。PWM控制器配置来产生一个25.5us的周期。ColdFire PWM模块允许8通道中每一个通道被配置一不同的周期。在12.75us周期下PWM频率是78.431KHZ。当通过一个低通滤波器时，占空比可以调制为从0到100%以提供0到3.3伏的电压。更高的PWM频率被用来简化低通滤波器设计。在高频情况下，大多数放大电路有足够的输入阻抗来消除独立低通滤波器需求。

### 6.1.1 初始化PWM控制器

ColdFire初始化工具是一个很有用的工具，在freescale.com上可以找到。它是一个基于Windows的GUI，允许你用图形的方式来配置ColdFire外设，并自动创建初始化代码。

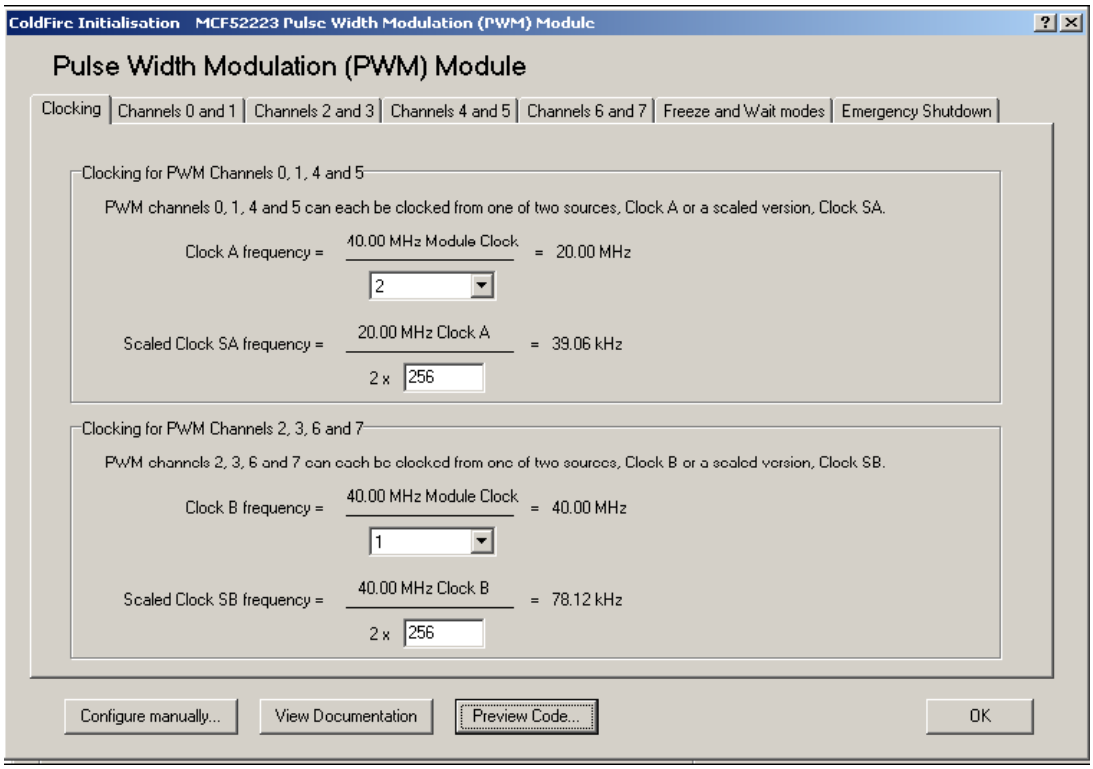



图14 设置PWM时钟

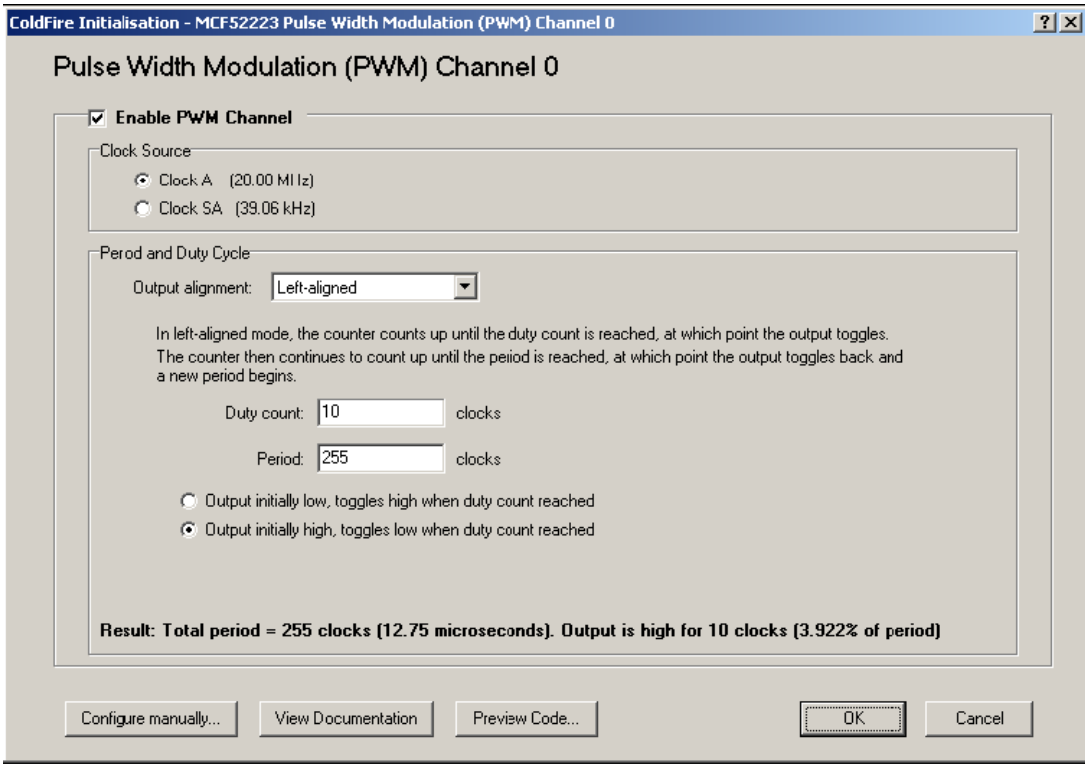



图15 配置PWM通道0

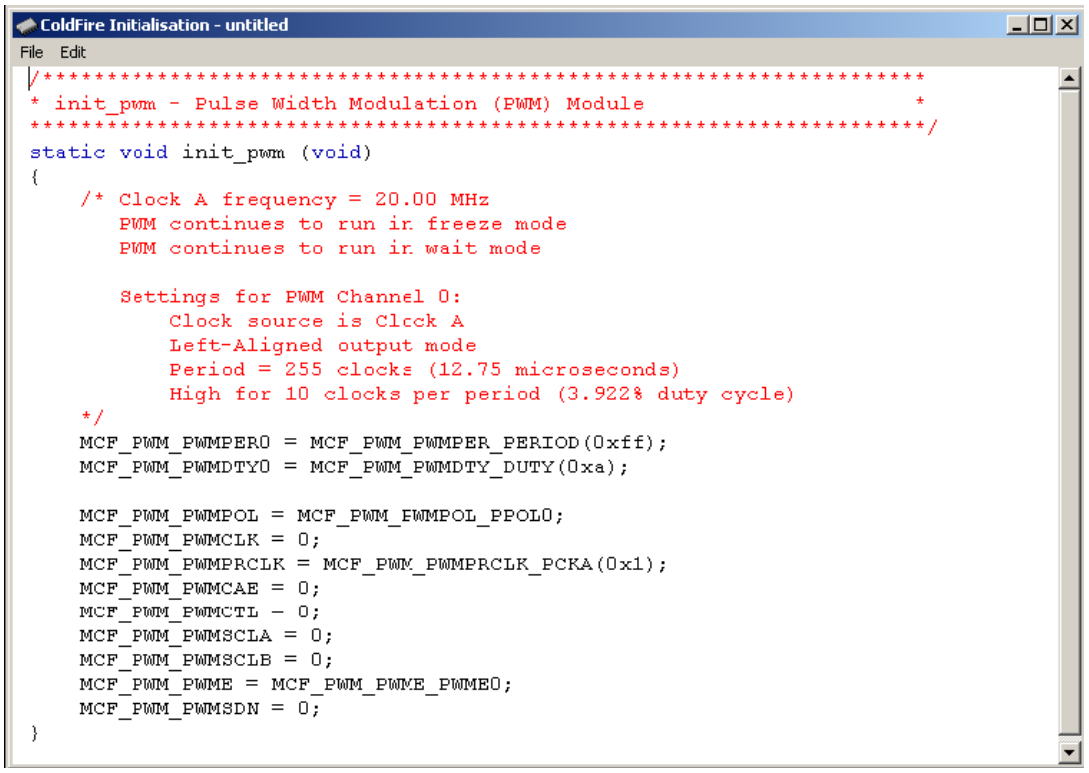



图16 点击预览代码键后的结果

### 6.1.2 调制PWM的占空比

PWM占空比必须每125us（8KHZ的采样率）更新一次。ColdFire使用一个PIT（programmable interval timer）来完成。PIT设置成每125us中断一次。在中断过程中从环形缓冲（下节将描述）中复制一字节到PWM占空比寄存器中。

#### 6.1.2.1 初始化PIT

使用PIT1是因为CMX USB协议栈为了处理内部超时使用了PIT0。

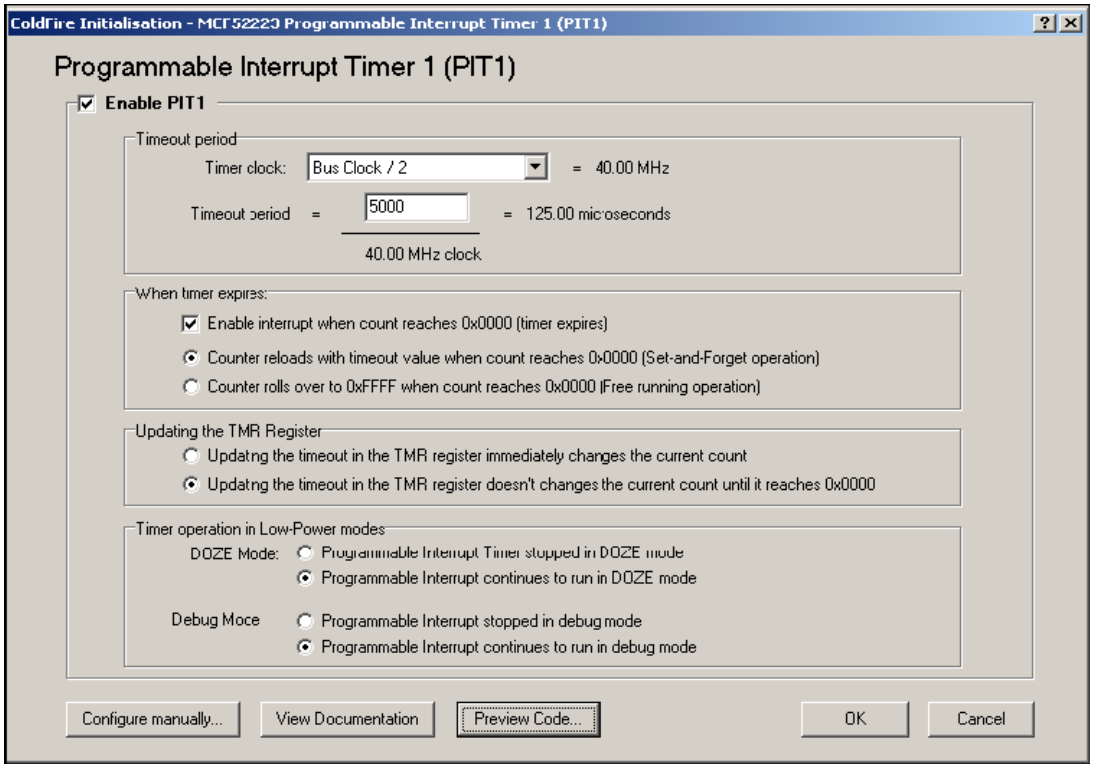



图17 使用CFInit配置PIT

点击预览代码将产生用于剪切和复制到源码中的初始化代码。

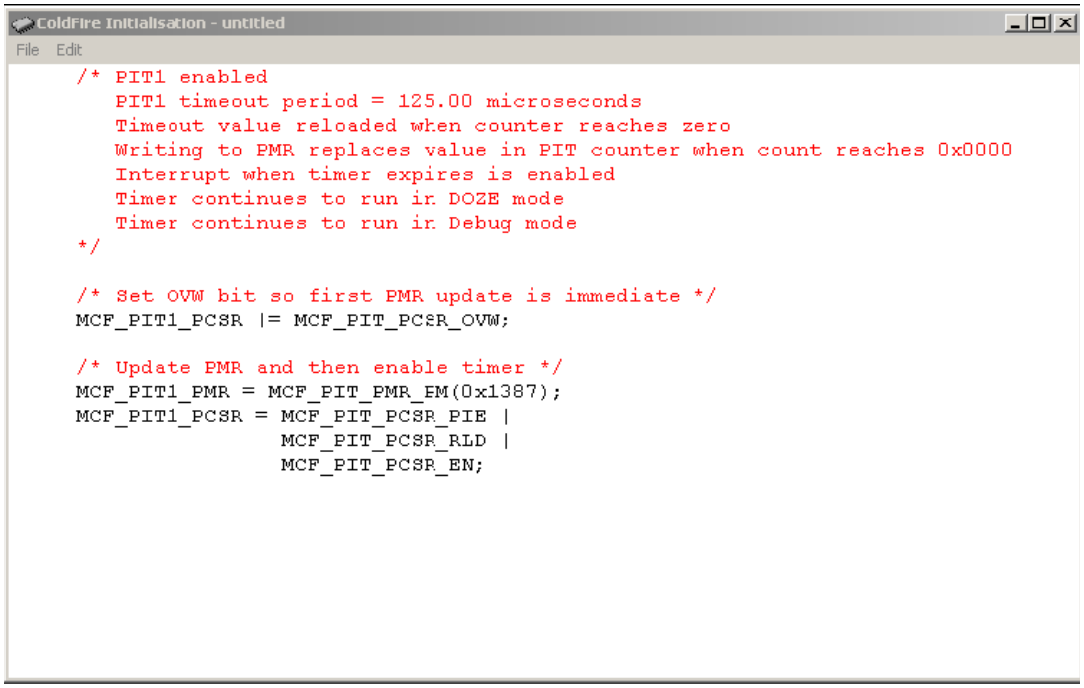



图17 使用CFInit配置PIT

### 6.1.2.2 PIT中断处理

PIT定时器每125us轮询一次。当它轮询后产生一中断。中断控制器必须被设置为使能PIT1中断，并且设置它的优先级。中断向量表也必须由一指向新中断处理函数的指针来初始化。

```
//
// Author: Eric Gregori (847) 651 - 1971
//
//
__declspec(interrupt:0)
void PIT1_it_handler(void)
{
    if( !mute && (data_out != data_in ))
    {
        MCF_PWM_PWMDTY1 = data_buffer[data_out++];
        LED1_TGL;
    }
    /* Clear interrupt at CSR */
    MCF_PIT_PCSR(1) |= MCF_PIT_PCSR_PIF;
}
/*********************************************************************
* init_interrupt_controller - Interrupt Controller *
*********************************************************************/
```

```
static void init_interrupt_controller (void)
{
    #ifdef AUDIO
    /* Configured interrupt sources in order of priority...
    Level 7: External interrupt /IRQ7, (initially masked)
    Level 6: External interrupt /IRQ6, (initially masked)
    Level 5: External interrupt /IRQ5, (initially masked)
    Level 4: External interrupt /IRQ4, (initially masked)
    PIT 0 interrupt
    Level 3: External interrupt /IRQ3, (initially masked)
    Level 2: External interrupt /IRQ2, (initially masked)
    Level 1: External interrupt /IRQ1, (initially masked)
    */
    MCF_INTC0_ICR56 = MCF_INTC_ICR_IL(0x4);
    MCF_INTC0_IMRH &= ~MCF_INTC_IMRH_MASK56;
    #endif
}
```

在中断控制器中PIT1的中断号为56，该信息可以从参考手册或者CFInit中得到。

中断控制器向量号不同于向量表中的向量号。USB协议栈的向量表包括中断控制器向量号来作为下一个未使用的向量。

```
    vector77: .long _irq_handler77 /* PIT0 PIF V55*/
    vector78: .long _PIT1_it_handler /* PIT1 PIF V56*/
    vector79: .long _irq_handler79 /* reserved */
```

PIT中断处理程序简单的从环形缓冲（下节将描述）中读取一字节写入到PWM占空比寄存器。然后增加环形缓冲的输出指针。

## 6.2 从闪存棒中读取WMV文件

数据从闪存块中以200字节/块的形式读出。然后这些数据每次一字节复制到一个环形缓冲区中。这样做以允许PIT中断及闪存棒读取代码完全与时间无关（不需要同步）。

使用f_read()函数来将数据从闪存棒中读出。

使用emgplay<文件名>命令来播放WAV文件。也可以用一指针来调用该函数，该指针指向字符串文件名，文件名以NULL结束。

```
volatile unsigned char data_in;
volatile unsigned char data_out;
volatile unsigned char data_buffer[256];
volatile unsigned char mute = 1;
//
```

```
// Author: Eric Gregori (847) 651 – 1971
//
void cmd_emgplay(char *param)
{
	F_FILE *file;
	unsigned char temp[204];
	unsigned char index;
	file=f_open(param, "r");
	if (file == 0)
	{
		print("Failed to open ");
		print(param);
		print(".\r\n");
		return;
	}
	print( "\f\rPlaying " );
	print(param);
	print( " - " );
	data_in = 0;
	data_out = 0;
	(void)f_read(temp, 1, 64, file);
	while(1)
	{
		int r = f_read(temp, 1, 200, file);
		if (r>0)
		{
			// Write temp into data_buffer
			for( index=0; index<r; index++ )
			{
				while( (unsigned char)(data_in + 1) == data_out );
				data_buffer[data_in++] = temp[index];
			}
			mute = 0;
		}
		else
		{
			if (!f_eof(file))
			{
				print("Error while reading ");
				print(param);
				print(".\r\n");
			}
			f_close(file);
			break;
```

```
            }
        }
        mute = 1;
        print( "Done" );
        print(".\r\n");
        return;
}
```

## 6.3 将块数据转换成流（环形缓冲）

环形缓冲是音频同步的核心。正如名字所示，环形缓冲没有头和尾，缓冲自身头和尾相连。它使用一个OUT下标（index）和一个IN下标（index）来完成工作。

使用IN下标向缓冲区中写入数据，使用OUT下标从缓冲区中读取数据。下标是无符号的字节，因此很自然地以256字节来循环。这即是环形缓冲的长度。OUT下标只有在当它不等于IN下标时被读取及增加。

```
if( !mute && (data_out != data_in ))
{
    MCF_PWM_PWMDTY1 = data_buffer[data_out++];
    LED1_TGL;
}
```

IN下标保持比OUT下标少一个计数。IN下标不能超过OUT下标。

```
while( (unsigned char)(data_in + 1) == data_out );
data_buffer[data_in++] = temp[index];
```

如上的代码循环执行知道OUT下标（data_out）超过IN下标(data_in)一个计数为止。

## 6.4 简易文本语音引擎

因为有播放WAV文件及访问大容量存储空间（任何版本的最高达2G的大容量存储栈）的能力，下一个逻辑选择是文本语音处理。独立的单词以WAV文件的形式存储在闪存棒中。文件名为单词：单词“eric”存储为eric.wav。一个简单的函数将句子（单词间有空格）中的单词转换为文件名，然后播放文件名。

使用命令emfsay<文件名>实现上述过程。简单创建一常用字的单词字典，复制到闪存棒中并且把它插入到ColdFire或者V1核心中。这种技术可以用于菜单，乐器，警报，游戏等。

## 6.4.1 emgsay 固件

```
//
// Author: Eric Gregori (847) 651 - 1971
//
void cmd_emgsay( char *param )
{
	unsigned char param_index;
	unsigned char file_index;
	unsigned char filename[32];
	for( param_index=0, file_index=0; 1 ; param_index++ )
	{
		if( param[param_index] < 0x41 )
		{
		  if( file_index )
		  {
		    if( file_index > 8 )
		    {
			filename[6] = '~';
			filename[7] = '1';
			file_index = 8;
		    }
		  // Add .wav to filename
		  filename[file_index++] = '.';
		  filename[file_index++] = 'w';
		  filename[file_index++] = 'a';
		  filename[file_index++] = 'v';
		  filename[file_index++] = 0;
		  file_index = 0;
		  cmd_emgplay( (char *)filename );
		  if( param[param_index] == 0 )
			break;
		  } else
			continue;
		} if( param[param_index] == '.' )
			cmd_emgplay( "dot.wav" );
		if( param[param_index] >= 0x41 )
		{
			filename[file_index++] = param[param_index];
		}
	}
}
```

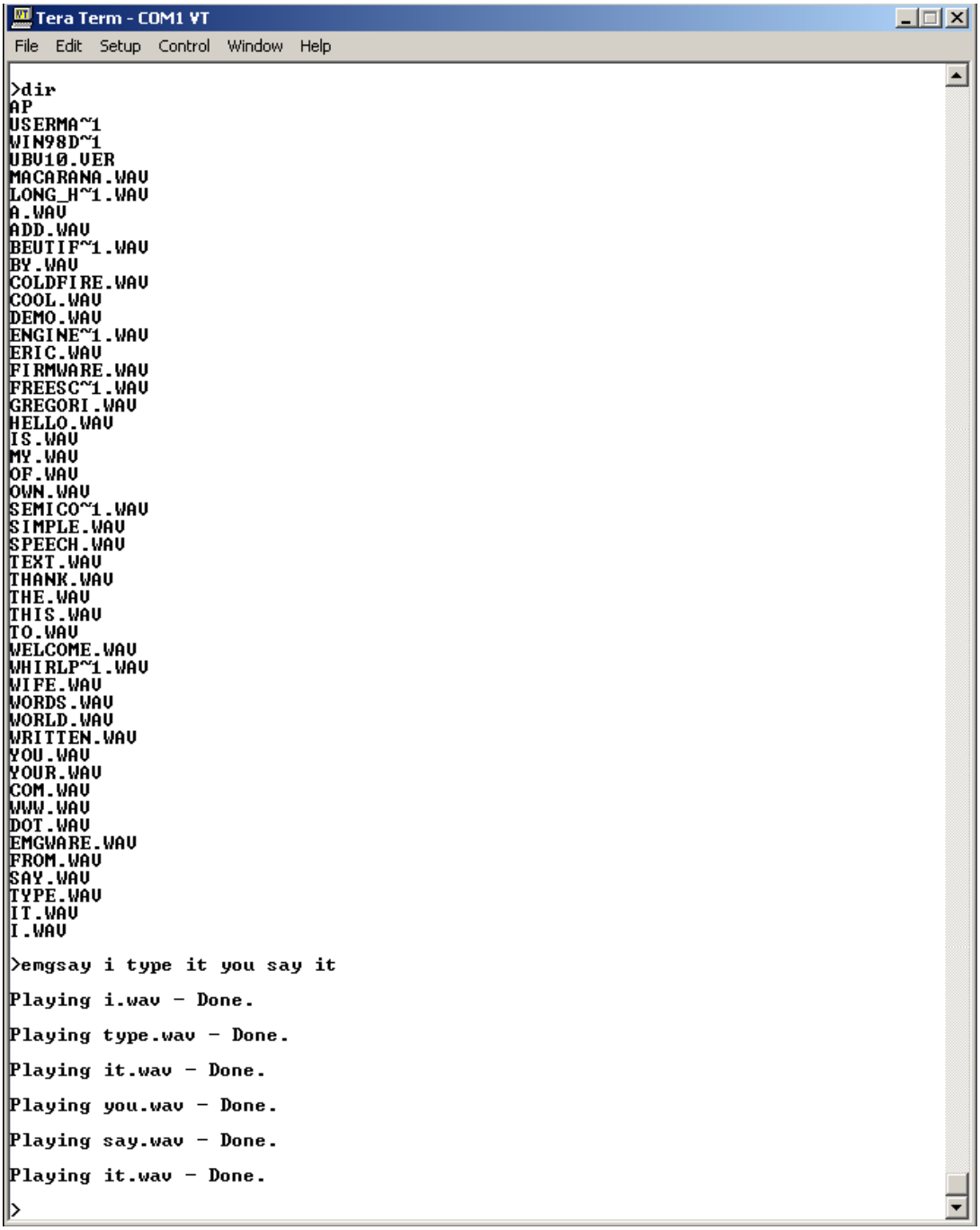



图18 闪存棒上的字典目录及emgsay命令使用