

# Advanced Applications for the Freescale USB\_Lite by CMX

by: Eric Gregori  
Product Specialist – Embedded Firmware

## 1 Introduction

This document covers advanced CMX USB applications. The CMX USB stack is covered in detail in AN3492, “USB and Using the CMX Stack.” AN3492 covers USB in general, enumeration, and both the host and device side stack APIs. The firmware discussed in this document, AN3523, is at the application layer of the stack.

The CMX USB stack includes both host and device side stacks. The stack currently runs on both the ColdFire® and the 8 bit JM60. The stack will also be available on future versions of the low-power V1 core.

The applications discussed here are built on top of the USB stack firmware. The stack can be downloaded with source from [freescale.com](http://freescale.com). Available for download with this document are binaries that can be programmed on a board, demonstrating the applications discussed.

### Contents

1	Introduction . . . . .	1
2	Accelerometer-based USB Mouse . . . . .	2
3	USB RC Servo (PWM) Controller and Analog Monitor . . . . .	12
4	USB Host Enumeration Sniffer (Host Driver Example) . . . . .	19
5	USB Flash Stick-based Analog Data Logger (Mass-Storage Class) . . . . .	25
6	USB Flash Stick-based WAV Player / Simple Text to Speech Engine . . . . .	37

## Accelerometer-based USB Mouse

Currently there are three demo boards available supporting the CMX USB stack; M52221DEMO, M52223EVB, M52211EVB, and the 8 bit DEMO9S08JM60. The JM60 currently only supports the device-side demos.

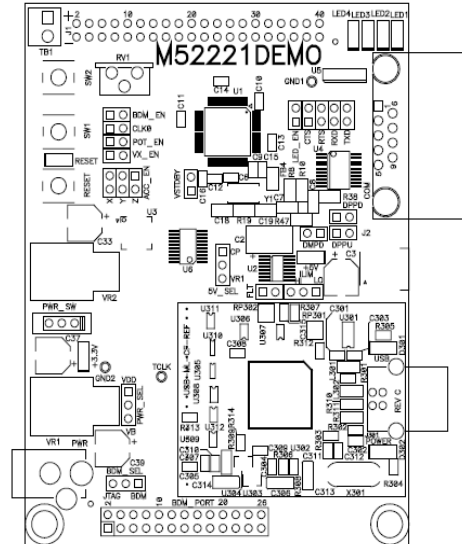
## 2 Accelerometer-based USB Mouse

The accelerometer based USB mouse firmware, demonstrates the CMX USB stack being used as a HID mouse. The accelerometer is used to detect the demo board's orientation relative to gravity. SW1 and SW2 on the demo board are used as the mouse left and right buttons.



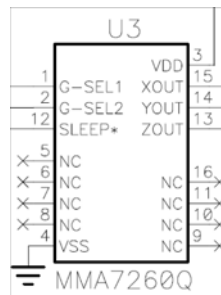
- Tilt the board left or right to move the mouse pointer left or right.
- Tilt the board forward or backwards to move the mouse pointer up or down.
- The more tilt, the faster the pointer moves.

**Tilt board to move mouse pointer (notice connectors on right)**



**Mouse buttons**

The accelerometer part number is MMA7260Q. It is a 3-axis accelerometer with programmable sensitivity from 1.5G to 6G. The tilt information is output in the form of a voltage, requiring a A/D converter on the MCU.

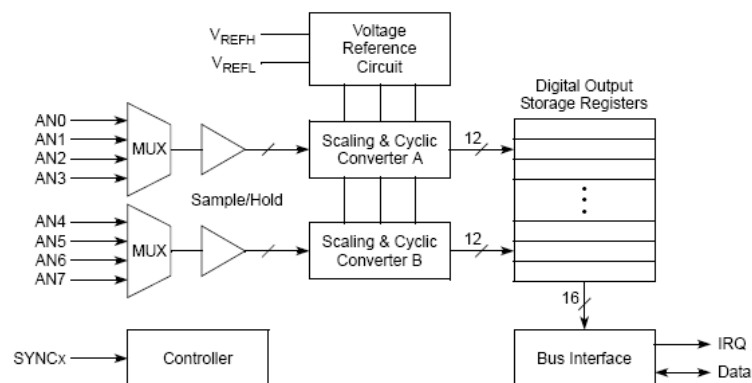


## 2.1 Using the A/D Converter to Read the Accelerometer

The ColdFire ADC's characteristics include the following:

- 12-bit resolution
- Maximum ADC clock frequency of 5.0 MHz, 200 ns period
- Sampling rate up to 1.66 million samples per second.
- Single conversion time of 8.5 ADC clock cycles ( $8.5 \times 200 \text{ ns} = 1.7 \mu\text{s}$ )
- Additional conversion time of 6 ADC clock cycles ( $6 \times 200 \text{ ns} = 1.2 \mu\text{s}$ )
- Eight conversions in 26.5 ADC clocks ( $26.5 \times 200 \text{ ns} = 5.3 \mu\text{s}$ ) using simultaneous mode
- Ability to simultaneously sample and hold 2 inputs
- Ability to sequentially scan and store up to 8 measurements
- Internal multiplex to select 2 of 8 inputs
- Power saving modes allow automatic shutdown/startup of all or part of ADC
- Those inputs not selected tolerate injected/sourced current without affecting ADC performance, supporting operation in noisy industrial environments.
- Optional interrupts at the end of a scan, if an out-of-range limit is exceeded (high or low), or at zero crossing
- Optional sample correction by subtracting a pre-programmed offset value
- Signed or unsigned result
- Single-ended or differential inputs for all input pins with support for an arbitrary mix of input types

The ADC function, shown below, consists of two four-channel input select functions, interfacing with two independent Sample and Hold (S/H) circuits, which feed two 12-bit ADCs. The two converters store their results in a buffer, awaiting further processing.



**Figure 1. ColdFire A/D Block Diagram**

The mouse uses 2 A/D channels (AN4, AN5) to read the X (AN4) and Y (AN5) signals from the MMA7260 accelerometer. The A/D is configured for continuous conversion.

## 2.1.1 A/D Initialization Function

```

/*****
* 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 Starting the A/D Converter

The `start_AD()` function is used to start the continuous conversion. After the `start_AD()` function is called, the A/D converter will continuously scan analog channels 0–7, and store the result in result registers 0–7.

Notice, the `start_AD()` function is starting conversions on the A converter (START0 in the CTRL1 register). When the SIMULT bit in the CTRL2 register is set, the A and B converters are both started and stopped using the START0 or STOP0 bits.

**Table 1. CTRL2 Register**

Field	Description
5 SIMULT	<p>A/D Converter Simultaneous Mode. This bit only affects parallel scan modes.</p> <p>When SIMULT=1 (default value) parallel scans operate in simultaneous mode. The scans in the A and B converter operate simultaneously and always result in pairs of simultaneous conversions in the A and B converter. START0, STOP0, SYNC0, and EOSIE0 control bits and the SYNC0 input are used to start and stop scans in both converters simultaneously. A scan ends in both converters when either converter encounters a disabled sample slot. When the parallel scan completes, the EOSI0 triggers if EOSIE0 is set. The CIP0 status bit indicates that a parallel scan is in process.</p> <p>When SIMULT=0, parallel scans in the A and B converters operate independently. The B converter has its own independent set of the above controls (START1, STOP1, SYNC1, EOSIE1, SYNC1) designed to control its operation and report its status. Each converter's scan continues until its sample list is exhausted (four samples) or a disabled sample is encountered. For looping parallel scan mode, each converter starts its next iteration as soon as the previous iteration in that converter is complete and continues until the STOP bit for that converter is asserted.</p> <p>0 = Parallel scans done independently 1 = Parallel scans done simultaneously (default)</p>

### 2.1.2.1 start\_AD() function

```
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 Reading the A/D Converter

With the A/D converter in continuous mode, A/D results are constantly being updated in the ADRSLT registers. These registers can be read at anytime by the application. The read\_AD(channel) function returns a 16 bit value representing the AD conversion result for the channel specified.

```
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 Mouse Report Overview

The mouse uses the HID class to communicate with the PC. The HID class is discussed in detail in AN3492, “USB and Using the CMX Stack.” The PC will automatically recognize the device as a mouse based on the report descriptor.

A HID report descriptor is used to identify a object, and how data is stored in a object. The HID report structure is defined in the document “Device Class Definition for Human Interface Devices (HID) Version 1.11” available at [www.usb.org](http://www.usb.org).

Report descriptors are groups of items, with each item defining a piece of information. Each piece of information (item) is composed of tags followed by value. The tag determines the type of information, and the value contains the actual configuration data. For example, a mouse descriptor contains a USAGE tag that defines the report as being for a mouse. It contains another USAGE tag defining a section of the report to be for buuton1, and another for button2, and finally 2 additional USAGE tags to define X and Y positioning.

INPUT tags define data from the device to the PC, while OUTPUT tags define data leaving the PC going to the device. Data can be variable or constant.

The tags and data are put together in a table that is read from start to finish by the report interpreter in the host (PC). The interpretation of the data is defined in the specification noted above. A tool is available on the usb.org website to assist in creating report descriptors ([http://www.usb.org/developers/hidpage/dt2\\_4.zip](http://www.usb.org/developers/hidpage/dt2_4.zip)). Thankfully it includes many examples. The [www.usb.org](http://www.usb.org) specification “Universal Serial Bus HID Usage Tables version 1.12” specifically defines the tags for the “mouse” usage.

## 2.3 Example Mouse Report Descriptor

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) */

## Accelerometer-based USB Mouse

```

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 Mouse Report Descriptor Breakdown

#### USAGE\_PAGE (Generic Desktop)

The « Generic Desktop » usage includes support for the following devices: pointer, mouse, joystick, game pad, keyboard, keypad, multi-axis controller. This usage is defined in section 4.1 of the “Universal Serial Bus HID Usage Tables version 1.12” specification.

#### USAGE (Mouse)

The « Mouse » is a device within the « Generic Desktop » category of devices selected above. This line tells the PC (host) that this device is a mouse.

#### COLLECTION (Application)

A collection is a group of tags. In this case, this group of tags defines an application which is a common device: mouse, keyboard, ...

#### USAGE (Pointer)

The following data defines a pointing device.

#### COLLECTION (Physical)

”sensing devices which may need to associate sets of measured or sensed data with a single point.”  
From “Device Class Definition for Human Interface Devices (HID) Version 1.11” section 6.2.2.6.

#### USAGE\_PAGE (Button)

#### USAGE\_MINIMUM (Button 1)

#### USAGE\_MAXIMUM (Button 3)

The following tags describe 3 buttons (the 3 buttons on a mouse).



LOGICAL\_MINIMUM (0)  
 LOGICAL\_MAXIMUM (1)

The buttons can only have 2 values, 0 or 1.

REPORT\_COUNT (3)  
 REPORT\_SIZE (1)

The button data will be contained in 3 bits (COUNT) in a single byte (SIZE).

INPUT (Data,Var,Abs)

The button USAGE information described above come from the device, into the PC. The data is variable and absolute.

REPORT\_COUNT (1)  
 REPORT\_SIZE (5)  
 INPUT (Cnst,Ary,Abs)

Define a 5 bit padder, so the button data is contained in a single byte.

The padding data comes from the device into the PC, and is constant and arbitrary.

USAGE\_PAGE (Generic Desktop)  
 USAGE (X)  
 USAGE (Y)

The X and Y data are subsets of the « Generic Desktop » usage. This usage is defined in section 4.2 of the “Universal Serial Bus HID Usage Tables version 1.12” specification.

LOGICAL\_MINIMUM (-127)  
 LOGICAL\_MAXIMUM (127)  
 REPORT\_SIZE (8)  
 REPORT\_COUNT (2)  
 INPUT (Data,Var,Rel)

The X and Y data are sent from the device to the PC as 2 bytes with a max of 127 and a min of -127. The data should be read as Variable and Relative.

With this descriptor the PC will treat the X and Y data as relative data, adding it or subtracting it from a base value. To move the pointer right, make X a positive number. The higher the number, the faster the pointer will move. Same rules apply for the Y component. Microsoft requires that both the X and Y components be either absolute or variable, not a combination of the 2.

END\_COLLECTION

End of physical collection

END\_COLLECTION

End of Application collection

## 2.4 Firmware

The CMX USB stack device API is described in AN3492, “USB and Using the CMX Stack.” At the top of the USB device firmware is the main() function. Data is sent and received via report queues. The mouse uses a single in report queue (direction is always relative to host – PC).

The PC is expecting relative X and Y data, so a delta is calculated by subtracting the new A/D reading from a reference A/D reading. The reference A/D reading is taken after reset. The board should be flat on a table after reset, so that the reference A/D reading can be set correctly.

```
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 Position of Data in Reports

The DIR\_REP\_X, DIR\_REP\_Y, and DIR\_REP\_BUTTONS macros insert data into the 3 byte report declared by hid\_add\_report(). These bit positions were defined in the HID report descriptor.

```

REPORT_COUNT (3)
REPORT_SIZE (1)
REPORT_COUNT (1)
REPORT_SIZE (5)

```

The button information is stored in bits 0, 1, and 2 of byte 0 of the report, with button 1 being bit 0. Microsoft defines button 1 as the “left” button, button 2 as the “right” button, and button 3 as the “center” button.

```

USAGE (X)
USAGE (Y)
LOGICAL_MINIMUM (-127)
LOGICAL_MAXIMUM (127)
REPORT_SIZE (8)
REPORT_COUNT (2)

```

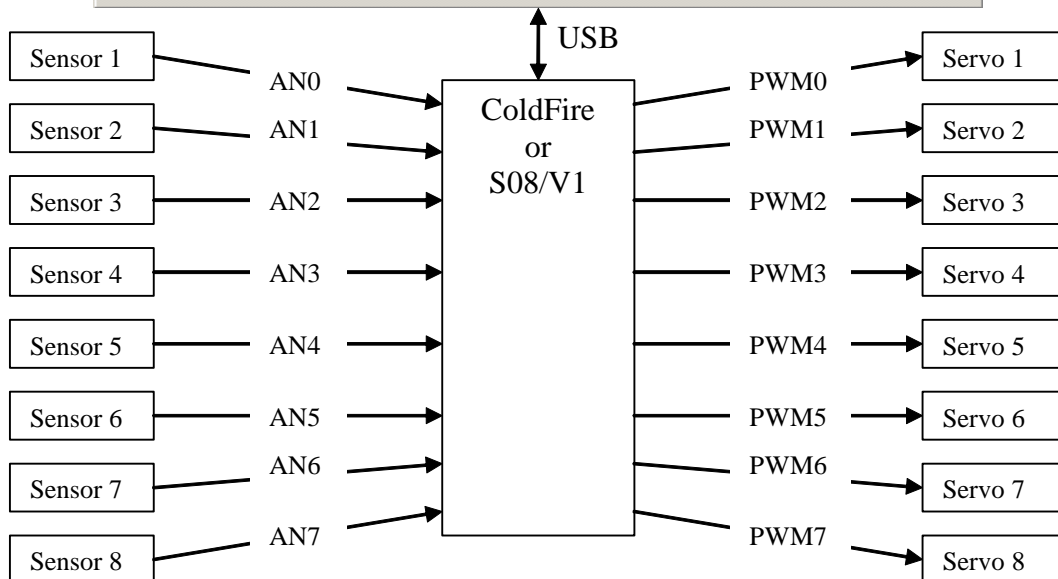
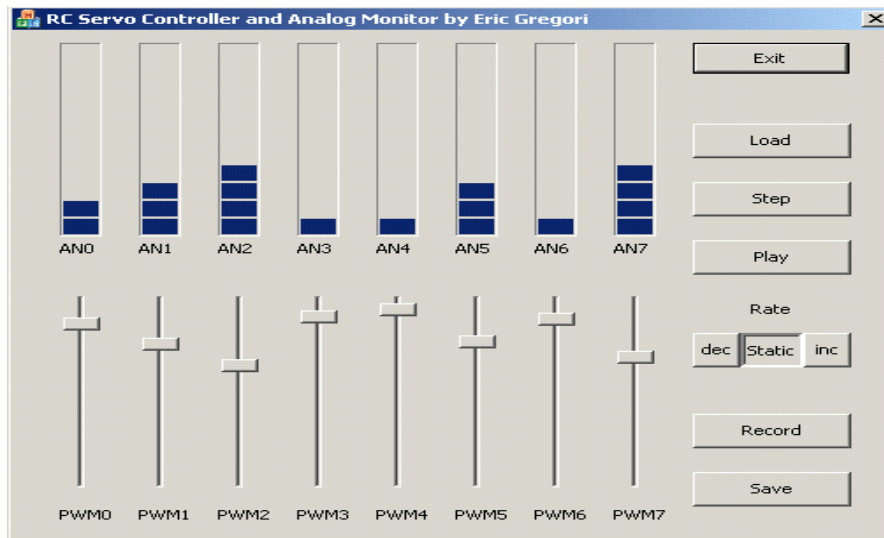
The X position is stored in byte 1 of the report, followed by the Y position in byte 2.

<b>3 button mouse report</b> 3 bytes ( 5 padding bits )									
Byte 2 76543210	Byte 1 76543210	Byte 0							
		7	6	5	4	3	2	1	0
Y Data	X Data						B3 center	B2 right	B1 Left

Figure 2. Mouse Data Structure

### 3 USB RC Servo (PWM) Controller and Analog Monitor

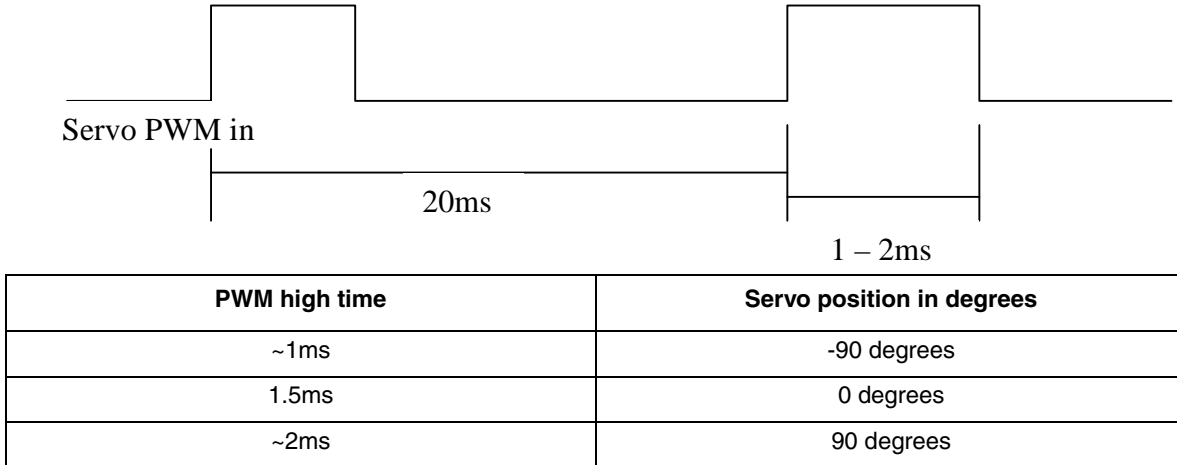
The RC Servo controller demonstrates the use of a custom ID device report descriptor, and the PC (host) side software required to communicate with the custom device. Up to 8 RC servo motors can be controlled via USB. This device side firmware also supports returning 8 bytes of analog data to the PC.



### 3.1 Controlling an RC Servo using the PWM Controller

A RC servo (Remote Control Servo) is a device used to control mechanical devices in the hobby world. Remote control cars, trucks, boats, and robots all use RC servos for steering, throttles, and legs and arms. The RC servo converts a PWM signal into a mechanical action. As the name implies, it is a servo with feedback and closed loop control. The PWM signal sets a position between -90 and 90 degrees, the servo locks in mechanically to a -90 to 90 degree position.

The PWM signal required to control a RC servo is a minimum of 3.3 volts peak (some servos will require up to 5), with a period of about 20ms (this does not have to be exact, but should be greater than 15ms and less than 50ms), and a high time of between 1ms and 2ms with a center of 1.5ms.



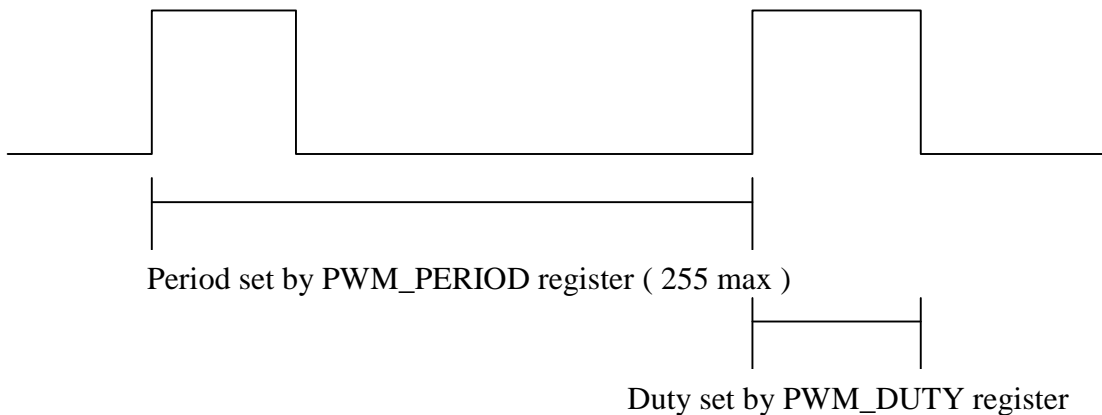
**Figure 3. Servo Position versus PWM High Time**

With a PWM high time range of 1ms, and a mechanical travel of 180 degrees, the servo resolves to about 5.6µs / degree. The actual resolution a servo can resolve to is dependant on the servo. RC servos are primarily analog devices (there are digital servos available) so their specifications vary somewhat.

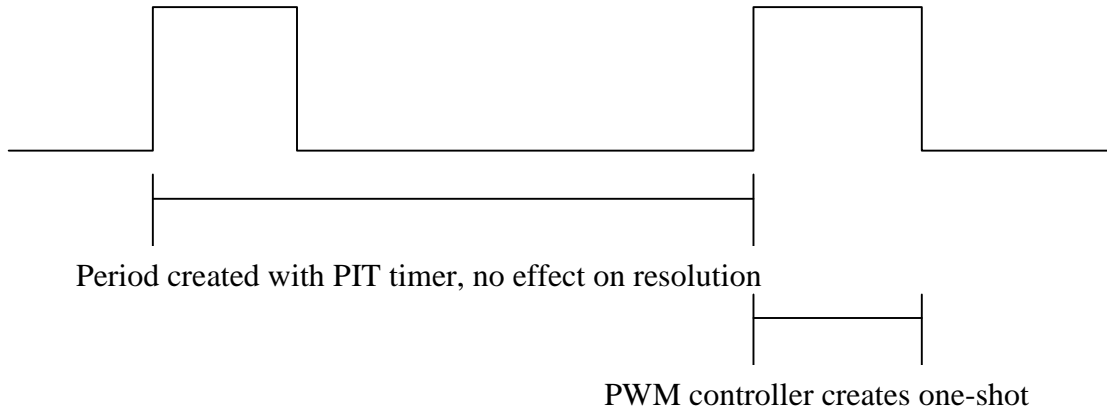
For this design I wanted to support up to 8 servos. This requirement dictated that the ColdFire PWM controller be used in the 8 channel by 8-bit mode. Using only an 8 bit counter, a period of 20ms would yield a resolution of 78µs (14 degrees). This is not acceptable for most applications. A method needed to be devised to provide a better resolution (less then 5 degrees) while still maintaining the 20ms period.

### 3.1.1 Using the ColdFire PWM Controller in “One Shot” Mode

The ColdFire PWM controller is uses a double buffering mechanism to eliminate glitches in the PWM stream. This mechanism can be used to create a “one shot” mode. In this mode, the PWM controller only creates a single pulse. This allows the full 8 bits of PWM counter resolution to be used within the pulse instead of spanning across the entire period.

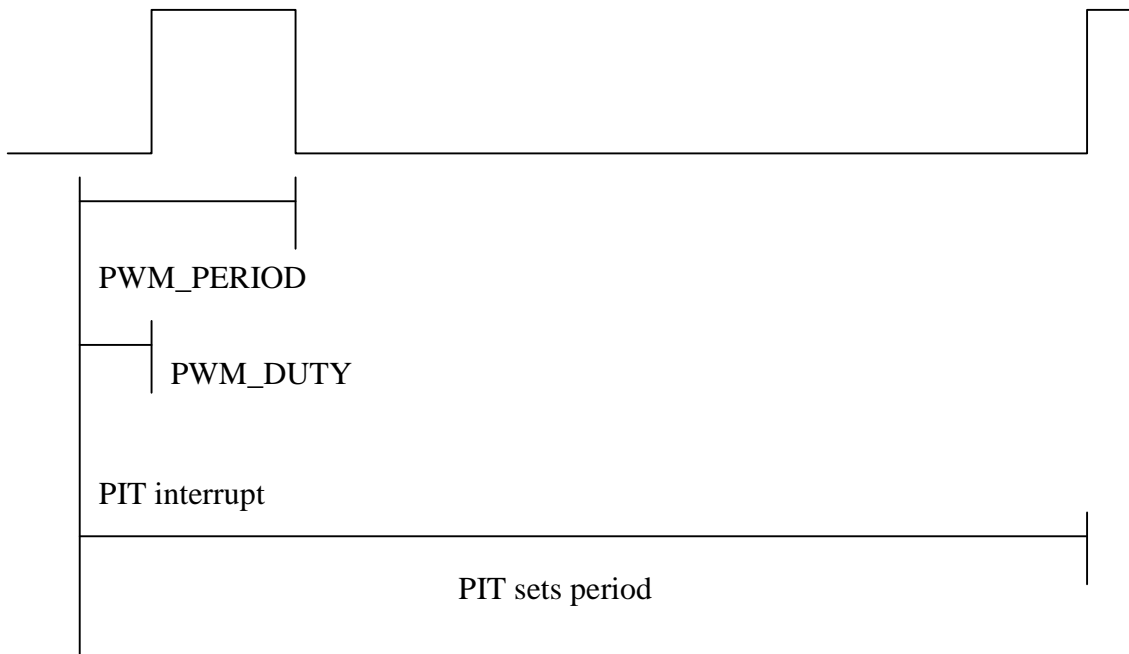


**Figure 4. Standard PWM Configuration – Limits Resolution to 255 Counts Across Entire Period**



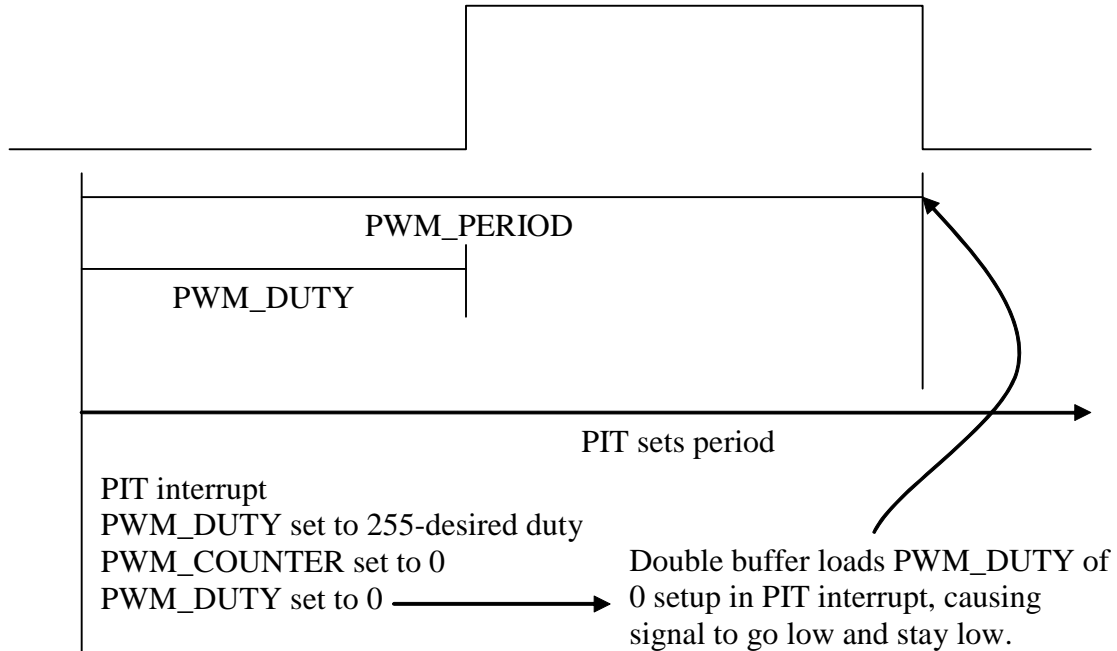
**Figure 5. One-Shot Mode Uses PIT Timer to Create PWM Period, and PWM Module to Create Pulse**

Using the one-shot mode, the high time resolution is increased significantly. The 8 bit PWM counter is used entirely to create the high pulse, while the PWM period is created using a separate PIT timer. The PIT timer creates a interrupt every PWM period, the PWM controller is then used to create a one-shot pulse. The one-shot pulse is created by setting the PWM\_DUTY register with the duty cycle, reset the PWM\_CONTER, then immediately setting the PWM\_DUTY to 0. The PWM\_DUTY = 0 setting does not take effect immediately due to the double buffering, instead it takes effect when the PWM\_PERIOD register times out.



PWM\_PERIOD = Period register in PWM controller  
 PWM\_DUTY = Duty cycle register in PWM controller  
 PIT = Programmable Interrupt Timer (Timer interrupts on modulus)

**Figure 6. PWM Configuration**



The PWM duty cycle register is actually configured to 255 – desired duty cycle. The PWM controller is configured to create a low going pulse. This results in the PWM controller setting the signal to zero after each **PWM\_PERIOD**.

### 3.1.2 PIT Interrupt Handler

```

__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 Reading the A/D Converter

This project uses the same A/D driver described in [Section 2.1, “Using the A/D Converter to Read the Accelerometer.”](#)

## 3.3 HID Report Overview

See [Section 2.2, “HID Mouse Report Overview,”](#) for a complete description of report descriptors. For this project I required a custom report descriptor. Using the tool from report descriptor tool from [http://www.usb.org/developers/hidpage/dt2\\_4.zip](http://www.usb.org/developers/hidpage/dt2_4.zip). I created a report descriptor that supports 8 byte IN and OUT transfers.

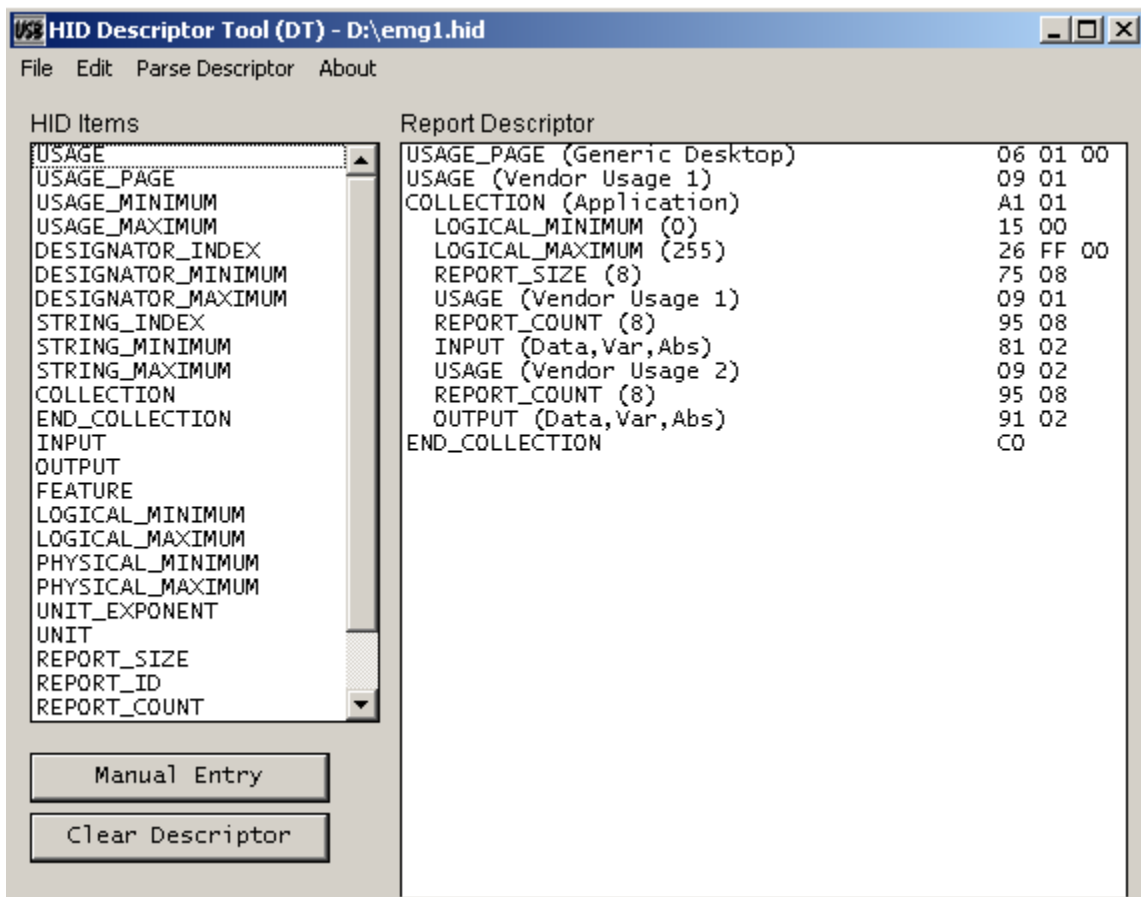


Figure 7. 8 byte IN / OUT Custom Report

### 3.3.1 Custom Report Descriptor Overview

LOGICAL\_MINIMUM (0)

LOGICAL\_MAXIMUM (255)- Range of data

REPORT\_SIZE (8)- 8 bits / piece of data

REPORT\_COUNT (8)- 8 pieces of data for input and 8 for output

This report will transfer 8 bytes of data IN and 8 bytes of data out. The 8 bytes of data into the PC are used to transfer the analog conversion results, and the 8 bytes of data OUT are used to adjust the PWM duty cycles.

### 3.4 Firmware

```
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 Host Application

```

// 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] );
    }
}
}

```

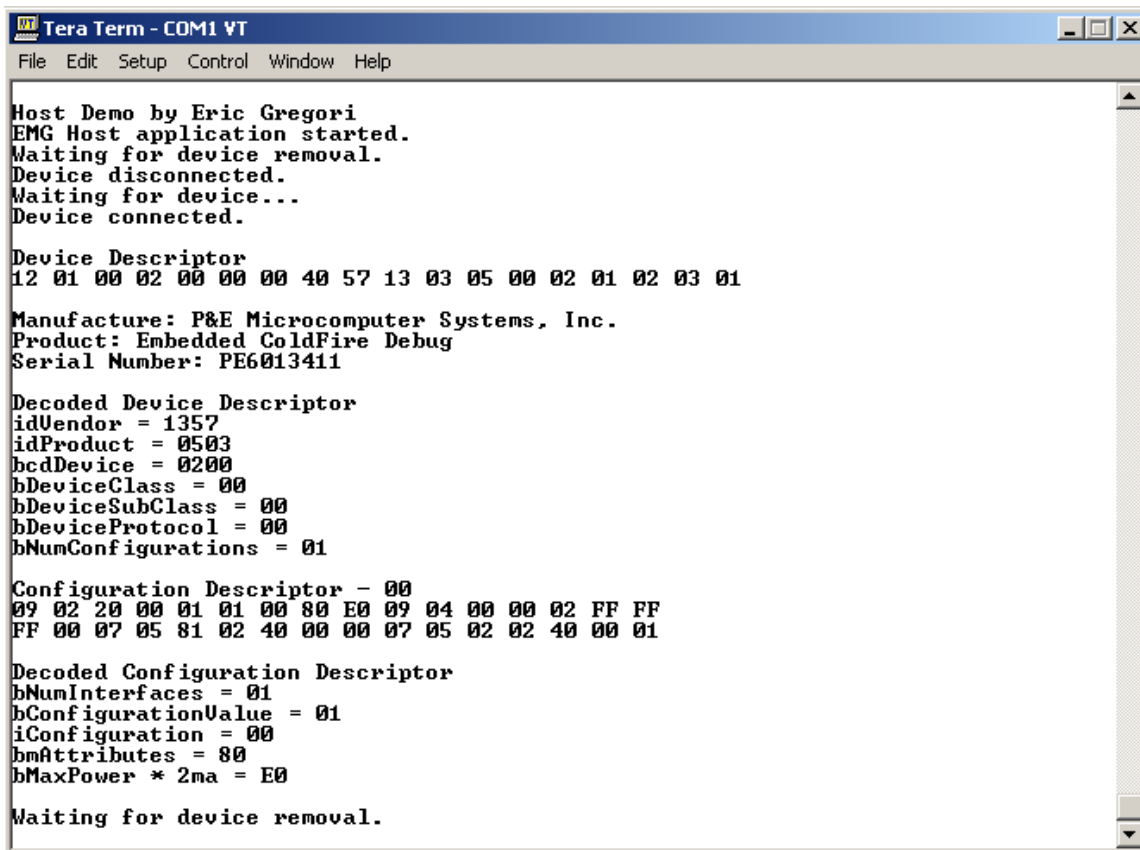
## 4 USB Host Enumeration Sniffer (Host Driver Example)

The CMX USB Stack supports both the host and device side. The host side of the stack is demonstrated here by showing the process of enumerating a device. Enumeration is the transfer of data structures (referred to as descriptors) from the device to the host. This occurs when the device is initially plugged into the host. Enumeration and the various descriptors are defined in the USB 2.0 specification from

## USB Host Enumeration Sniffer (Host Driver Example)

[www.usb.org](http://www.usb.org). It is also described very thoroughly, along with a description of the host firmware API in AN3492, “USB and Using the CMX Stack.”

Using this firmware, you can attach various devices to the demo or EVB board, and display the descriptors sent from the device to the host. Using the descriptor definitions in the USB 2.0 specification, you can decode the meaning of the descriptors.



```

Host Demo by Eric Gregori
EMG Host application started.
Waiting for device removal.
Device disconnected.
Waiting for device...
Device connected.

Device Descriptor
12 01 00 02 00 00 00 40 57 13 03 05 00 02 01 02 03 01

Manufacturer: P&E Microcomputer Systems, Inc.
Product: Embedded ColdFire Debug
Serial Number: PE6013411

Decoded Device Descriptor
idVendor = 1357
idProduct = 0503
bcdDevice = 0200
bDeviceClass = 00
bDeviceSubClass = 00
bDeviceProtocol = 00
bNumConfigurations = 01

Configuration Descriptor - 00
09 02 20 00 01 01 00 80 E0 09 04 00 00 02 FF FF
FF 00 07 05 81 02 40 00 00 07 05 02 02 40 00 01

Decoded Configuration Descriptor
bNumInterfaces = 01
bConfigurationValue = 01
iConfiguration = 00
bmAttributes = 80
bMaxPower * 2ma = E0

Waiting for device removal.
  
```

Figure 8. Sniffer Result from Plugging into On-board ColdFire Debugger (M52221DEMO)

```

Tera Term - COM1 VT
File Edit Setup Control Window Help

Host Demo by Eric Gregori
EMG Host application started.
Waiting for device removal.
Device disconnected.
Waiting for device...
Device connected.

Device Descriptor
12 01 10 01 00 00 00 40 45 0C 0D 60 01 01 00 01 00 01

Product:

Decoded Device Descriptor
idVendor = 0C45
idProduct = 600D
bcdDevice = 0101
bDeviceClass = 00
bDeviceSubClass = 00
bDeviceProtocol = 00
bNumConfigurations = 01

Configuration Descriptor - 00
09 02 17 01 01 01 00 80 FA 09 04 00 00 03 FF FF
FF 00 07 05 81 01 00 00 01 07 05 82 02 40 00 00
07 05 83 03 01 00 64 09 04 00 01 03 FF FF FF 00
07 05 81 01 80 00 01 07 05 82 02 40 00 00 07 05
83 03 01 00 64 09 04 00 02 03 FF FF FF 00 07 05
81 01 00 01 01 07 05 82 02 40 00 00 07 05 83 03
01 00 64 09 04 00 03 03 FF FF FF 00 07 05 81 01
80 01 01 07 05 82 02 40 00 00 07 05 83 03 01 00
64 09 04 00 04 03 FF FF FF 00 07 05 81 01 00 02
01 07 05 82 02 40 00 00 07 05 83 03 01 00 64 09
04 00 05 03 FF FF FF 00 07 05 81 01 A8 02 01 07
05 82 02 40 00 00 07 05 83 03 01 00 64 09 04 00
06 03 FF FF FF 00 07 05 81 01 20 03 01 07 05 82
02 40 00 00 07 05 83 03 01 00 64 09 04 00 07 03
FF FF FF 00 07 05 81 01 84 03 01 07 05 82 02 40
00 00 07 05 83 03 01 00 64 09 04 00 08 03 FF FF
FF 00 07 05 81 01 FF 03 01 07 05 82 02 40 00 00
07 05 83 03 01 00 64
    
```

Figure 9. Sniffer Results from Plugging in a USB Camera

## 4.1 Firmware

### 4.1.1 emg\_host\_demo()

The following code is an example of how to enumerate a device using the host API. In this example, the firmware prints out the device and configuration descriptors to the serial port (38400, 8, n, 1). A new function (not part of the standard stack) was written to request the string descriptors from the device.

```

//
// Enumerate device, and output device / configuration descriptors to the serial port in hex
// Serial descriptors are also printed to the serial port
//
// Written by Eric Gregori(847) 651 - 1971
//
    
```

## USB Host Enumeration Sniffer (Host Driver Example)

```

int main(void)
{
    hcc_u8          cfg, str1, str2, str3;
    hcc_ul6        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\r\n\r\n" );
    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\rvidVendor = " );
    emg_printwordhex( dev_inf.vid );
    print( "\n\rvidProduct = " );
    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 = " );

```

## USB Host Enumeration Sniffer (Host Driver Example)

```

        emg_printbytehex( cfg_inf.ndx );
    print( "\n\rConfiguration = " );
        emg_printbytehex( cfg_inf.str );
    print( "\n\rAttributes = " );
        emg_printbytehex( cfg_inf.attrib );
    print( "\n\rMaxPower * 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 Displaying a String Descriptor – 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 ((USBDSCTYPE(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);
}

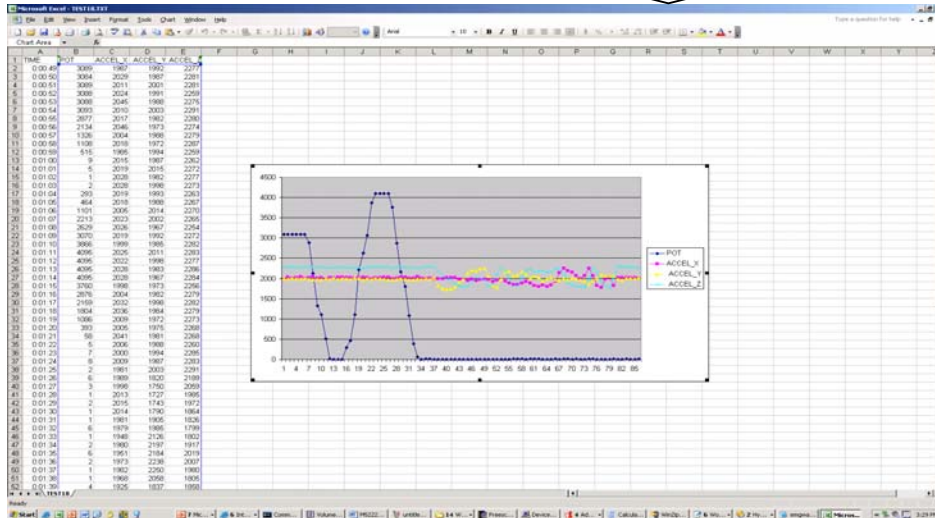
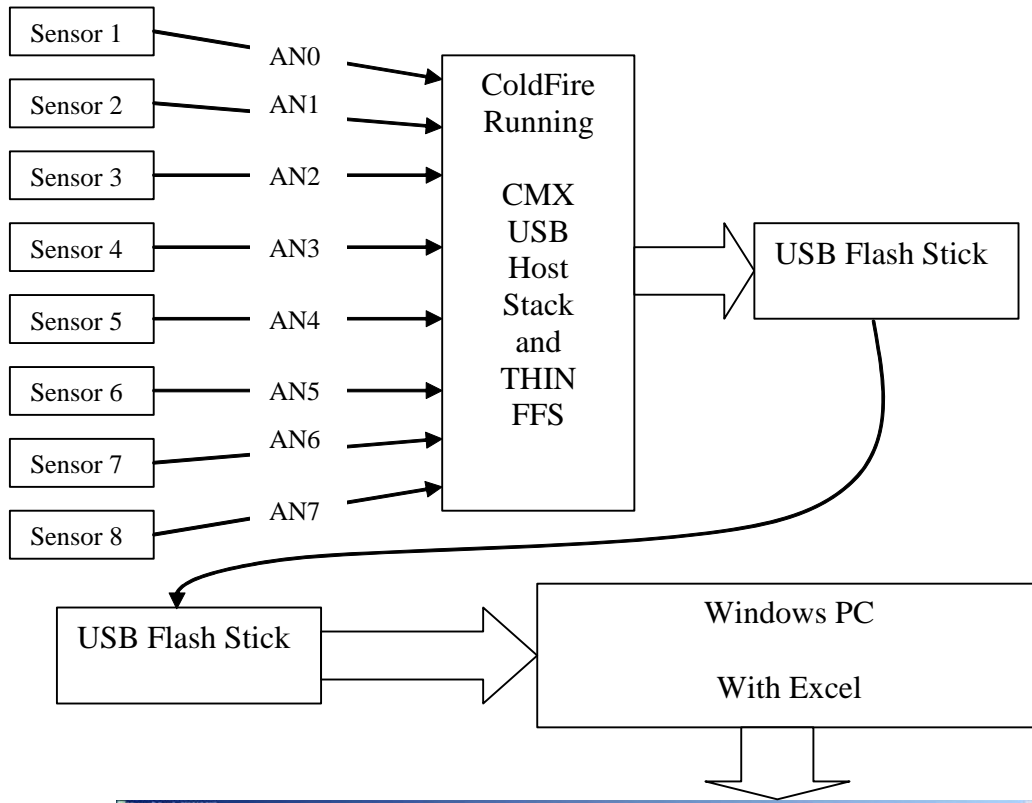
```

## 5 USB Flash Stick-based Analog Data Logger (Mass-Storage Class)

The CMX USB host stack supports the ability to read and write to a flash memory stick. Data written to the flash stick by the CMX USB stack can be read from a PC without any additional software being installed on the PC. Data written to a flash stick by the PC, can be read by the CMX USB stack.

This firmware logs analog data to the flash stick in a standard ascii format that can be imported by Microsoft Excel. The firmware reads 4 analog channels (easily expandable to all 8) and includes a time stamp for each set of data.

## USB Flash Stick-based Analog Data Logger (Mass-Storage Class)



### 5.1 Mass-Storage Class

The USB Mass-Storage class is specified in the “Universal Serial Bus Mass Storage Class Specification Overview” revision 1.2. The specification can be found at [www.usb.org](http://www.usb.org). The Mass-Storage class puts a USB wrapper around the ATAPI (Advanced Technology Attachment Packet Interface) and SCSI (Small Computer System Interface) command sets.

## 5.2 Command Sets

Table 2. SubClass Codes/Command Sets

SubClass Code	Command Block Specification	Typical Use
1	Reduced Block Commands (RBC)	Flash devices
2	SFF-8020i, MMC-2(ATAPI)	CD/DVD
3	QIC-157	Tape Drive
4	UFI	Floppy Drive
5	SFF-8070i	Floppy Drive
6	SCSI	

## 5.3 USB Transport Mechanism (BULK)

The Mass-Storage class uses Bulk transfers to transport data to and from the Mass-Storage device. Bulk transfers are handshaked transfers with 64 byte max payload sizes. There can be more than 1 transfer per frame, with a maximum of 19 transfer per frame for a total max transfer speed of 1216 bytes / millisecond or 1216000 bytes / second.

Bulk Transfer Features:

1. “Bandwidth available” access to the USB
2. Retry of transfer (handshaking)
3. Guaranteed delivery of data with no guarantee of latency

Bulk transfers are the least priority transfer for bandwidth allocation.

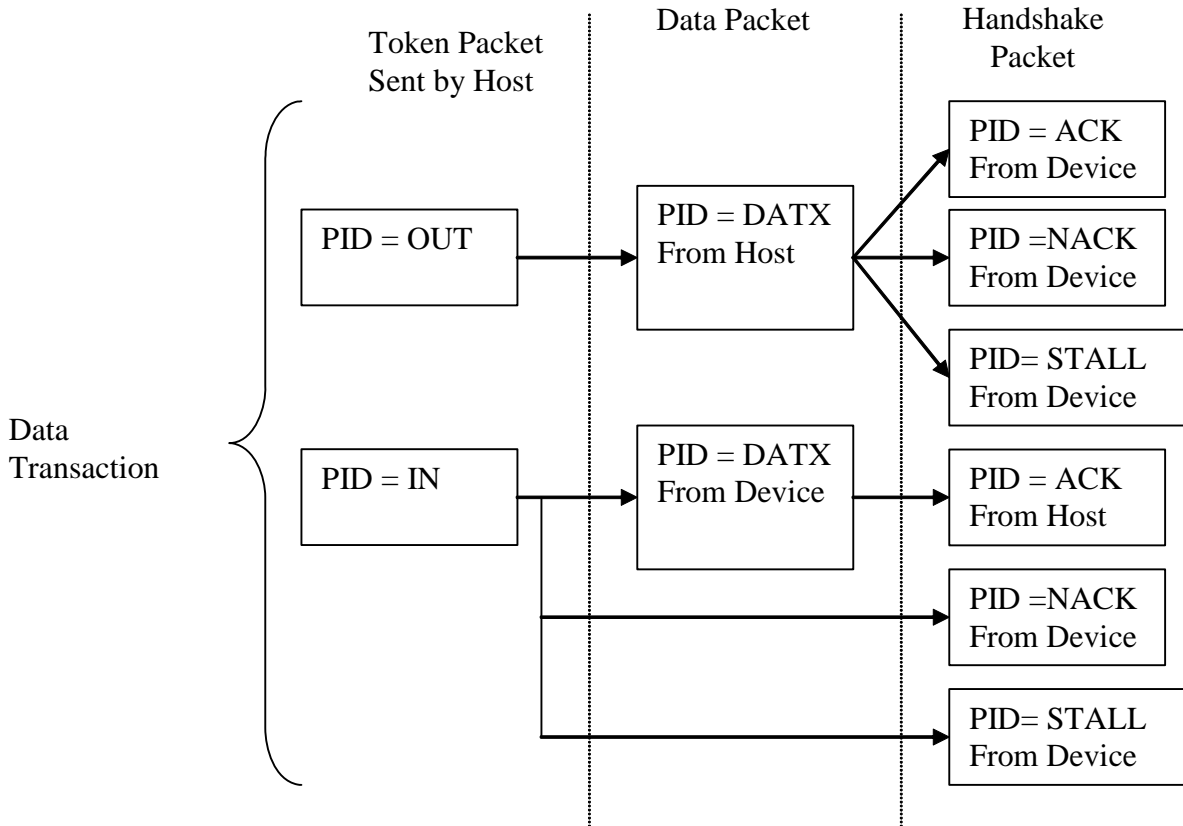


Figure 10. Bulk Transfer Transaction

## 5.4 CMX THIN Flash File System

The CMX THIN flash file system is designed for embedded systems with “limited resources.” The THIN file system provides FAT12, FAT16, and FAT32 file system support. The THIN file system is a layer of software that sits on top of the Mass-Storage Class, and provides a standard API to the users application. The CMX THIN flash file system is described in detail, including API, in the `CMX_FFS_THIN_5222x.pdf` file located in the docs directory of the USB stack.

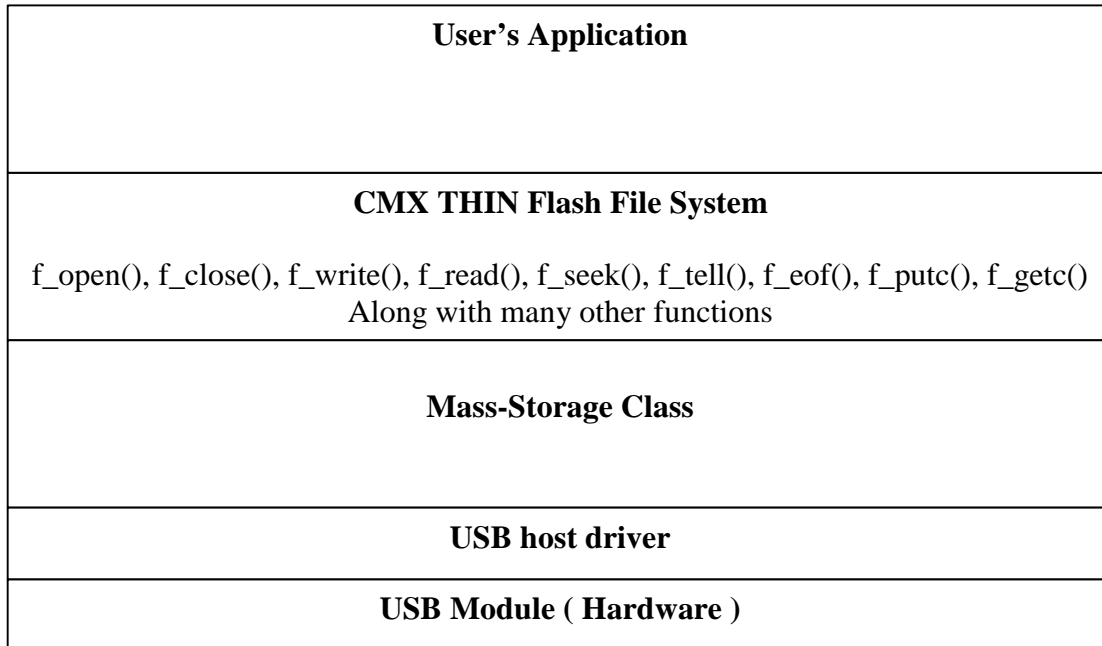


Figure 11. Mass-Storage USB Stack

## 5.5 Using the A/D Converter

The A/D converter is used to sample the analog channels for the data logger. The A/D converter is setup in a continuous mode. The data logger simply reads the A/D data via the simple driver after each sample period. No attempt here has been made to synchronize the A/D channels. In fact, this firmware reads each A/D channel one at a time, doing the binary to decimal conversions and flash writing between reads. This most likely will not be acceptable for a real-world data logger, but this sample project could easily be modified to support the required synchronization. The A/D converter is covered in section 2.1.

## 5.6 Using the RTC

The real-time clock (RTC) is used to provide a “timestamp” to each sample. The following code is for the RTC in the MCF5222X (M52223EVB and M52221DEMO). The RTC is clocked from the CPU’s primary clock through a divider. The RTC provides the hours, minutes, and seconds in binary format.

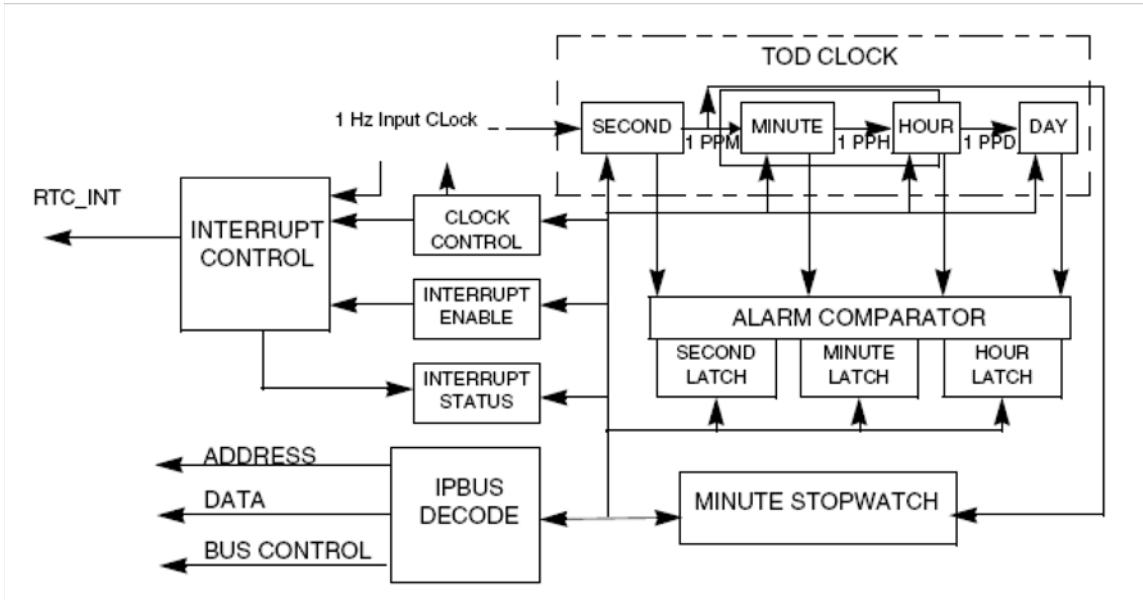


Figure 12. RTC Block Diagram

### 5.6.1 Configuring the RTC

The RTC requires a 1 Hz clock input. This clock is derived by dividing the system clock. The divider is configured with the RTCDR register.

Table 3. MCF\_CLOCK\_RTCDR Register

Field	Description
31-0 RTCDF	Real Time Clock Divide Factor. This field is used to divide down the system clock by a factor of RTCDF+1 (1 to 4,294,967,296) for the Real Time Clock module.

The “system clock” specified as the input into the divider is actually the INPUT into the PLL. The input into this divider is the crystal frequency.

```
/* Set real time clock freq - Oscillator clock (crystal frequency) = 48Mhz */
MCF_CLOCK_RTCDR = 48000000-1;
```

### 5.6.2 Reading the RTC

Reading the RTC simply requires reading the hours, minutes, and seconds registers.

```
//
// 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 Data Logger Firmware

### 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 char 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_ul6 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;

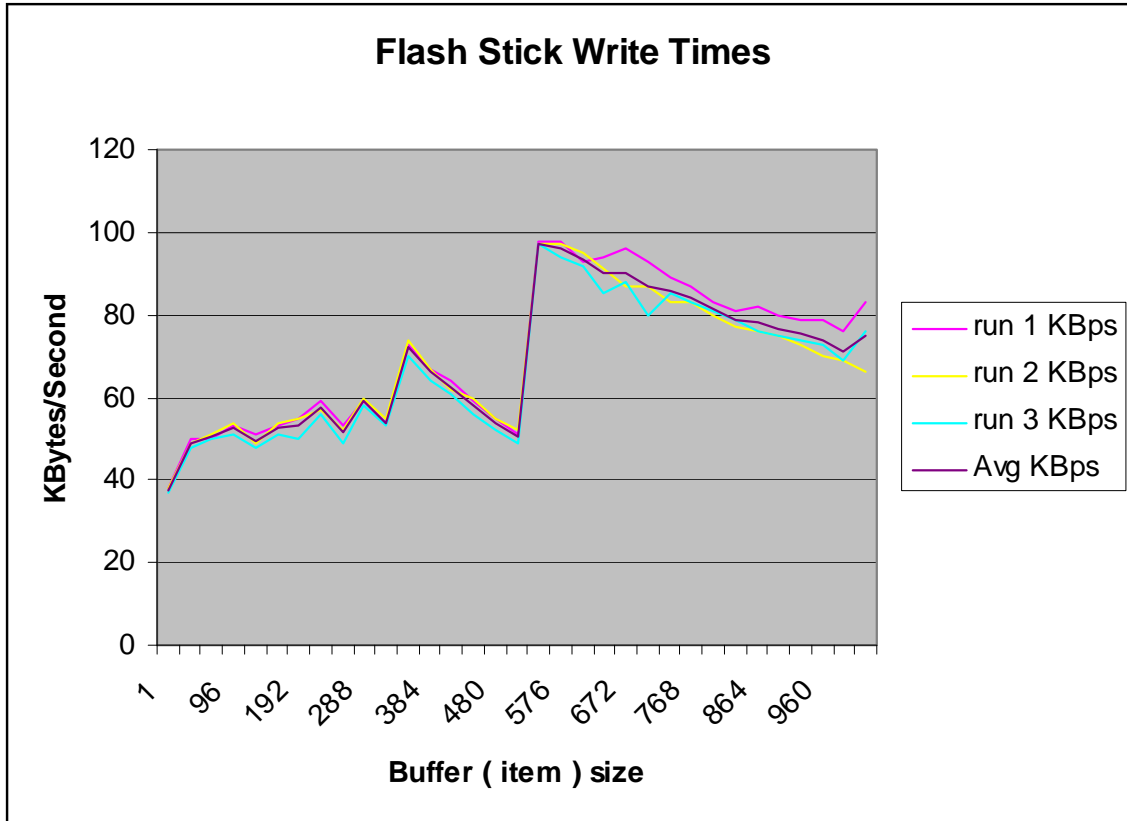
        if (uart_input_ready())
        {
            break;
        }
    }

    f_close( file );
    print( "\nFile Closed" );
    return;
}
```

## 5.8 The Speed At Which Data Can Be Written

To test the maximum write transfer speeds to the flash stick, a simple piece of firmware was written. The firmware simply writes items of data to the memory stick as fast as it can for 1 second. The result is sent out through the serial port.

Many different size items were used to measure the performance. As expected, the larger the item, the better the performance. The USB 2.0 specification for BULK transfers indicates that the maximum theoretical data transfer is 1187.5 Kbytes/second assuming a packet size of 64 bytes. Each packet is only 64 bytes of data, the critical variable is the number of transactions that can be done in a frame. Each transaction carries 64 bytes of data.



### 5.8.1 The emgtest <filename> Command

Writes to the file <filename> as fast as possible using different size items.  
 Uses `f_write( buff, size, 1, file )` to write to file.

where; buff is a un-initialized region of memory (data written to flash stick),  
 size is the number of bytes written / call to `f_write()` – “item size”.

```

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 Real-Time Usage

The Mass-Storage firmware, including the file system is single threaded. The firmware does not use interrupts, but does use a single timer PITO to keep track of hardware timeouts. The only time the firmware “spins” is when it is waiting for the USB OTG module to become available. The heart of the USB host driver is the function `usb_host_start_transaction()` in the file `usb_host.c`.

I instrumented the USB host to output a high on a test pin whenever it was spinning waiting for the hardware to complete a transaction. The following scope image was taken using this code:

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

The LED3 signal is marked as “Start” on the image below. The more active signal is the instrumentation in the `usb_host_start_transaction()` function. Everytime the function spins waiting for a transaction to complete, the signal goes high. The function `usb_host_start_transaction()` is spinning 76% of the time.

**NOTE**

Writing to USB flash only uses 24% of real-time, the rest of the time it is spinning.

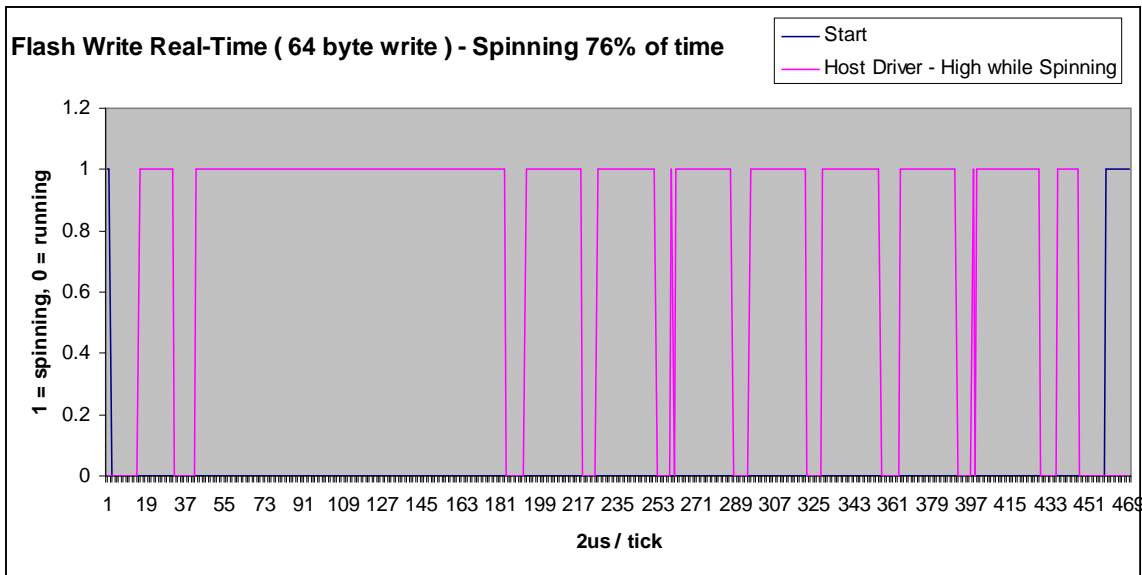
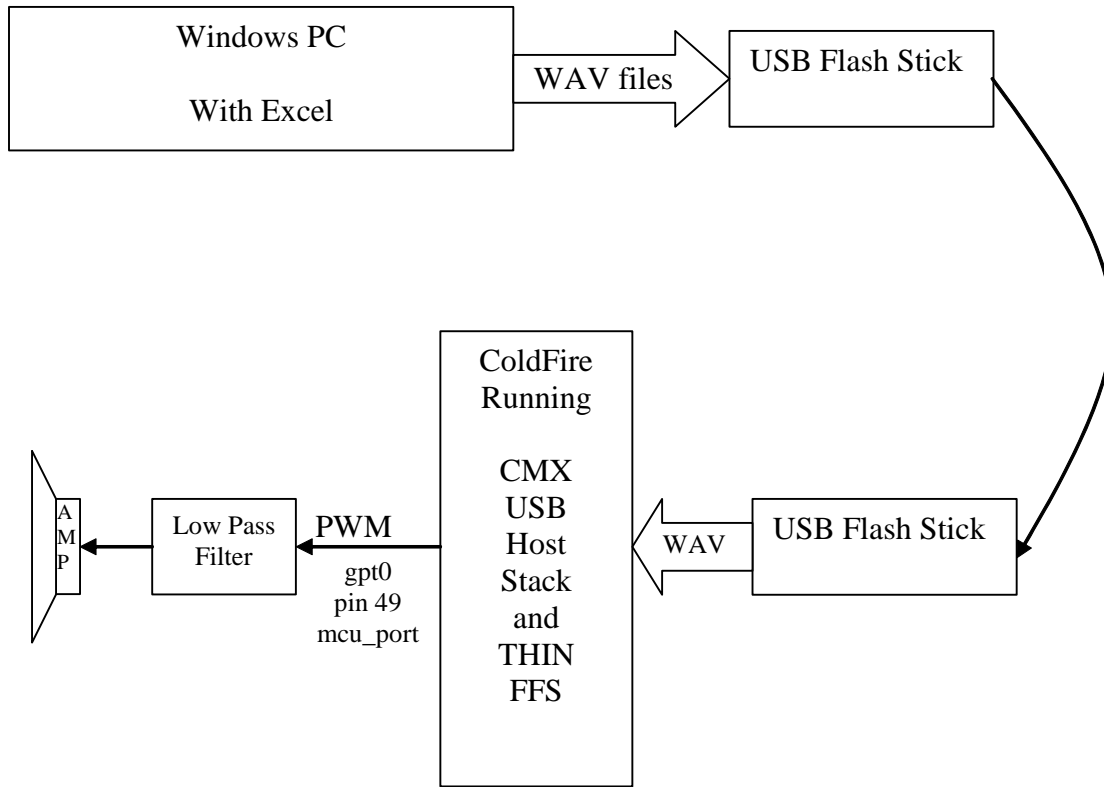


Figure 13. Real-Time analysis data

## 6 USB Flash Stick-based WAV Player / Simple Text to Speech Engine

Using the PWM module as simple digital-to-analog converter, audio can be played from the flash stick. Use a PC to either convert a audio file to a WAV format, or create a new WAV file on the PC using the sound recorder tool. Save the file to the flash stick. Plug the flash stick into the DEMO or EVB board, and playback the file using the `emgplay <filename>` command.

Currently the firmware is setup to work with 8 bit mono PCM files sampled at 8 Khz. The sample rate was an arbitrary decision based on supporting some old audio samples I had done for an 8 bit project (audio over 802.15.4). The USB stack can easily support higher playback rates, stereo, and compression with additional firmware.



## 6.1 Using a PWM Channel for Audio Applications

The ColdFire parts include an 8 channel 8 bit PWM module, that can be configured to be a 4 channel 16 bit module. The number of channels on a S08 (JM60) or V1 core based part will depend on the number of timers available, and the number of channels per timer.

For the ColdFire part, we use the PWM module in it's 8 channel, 8 bit mode. The PWM controller is configured to produce a period of 25.5 $\mu$ s. The ColdFire PWM module allows for each of the 8 channels to be configured with a different period. At 12.75 $\mu$ s, the PWM frequency is 78.431Khz. The Duty cycle is modulated from 0 to 100% providing a voltage of 0 to 3.3 volts when passed through a low pass filter. The higher PWM frequency was chosen to simplify the low pass filter design. At this high of a frequency, most amplifier circuit have enough input impedance to eliminate the requirement for a separate low pass filter all together.

### 6.1.1 Initializing the PWM Controller

The ColdFire Init tool available from [freescale.com](http://freescale.com) is a fantastic tool. It's a Windows-based GUI, that allows you to configure ColdFire peripherals in a graphical manner, and automatically creates the initialization code for you.

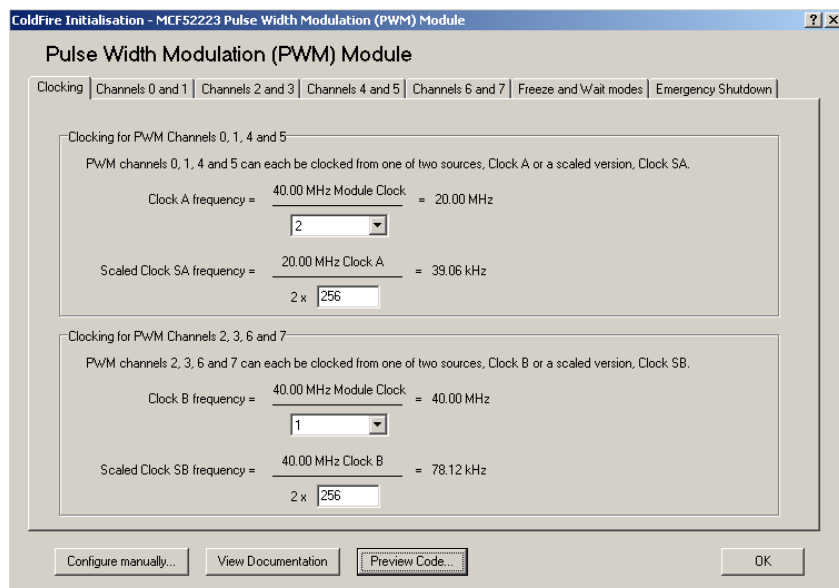


Figure 14. Configuring the PWM Clock

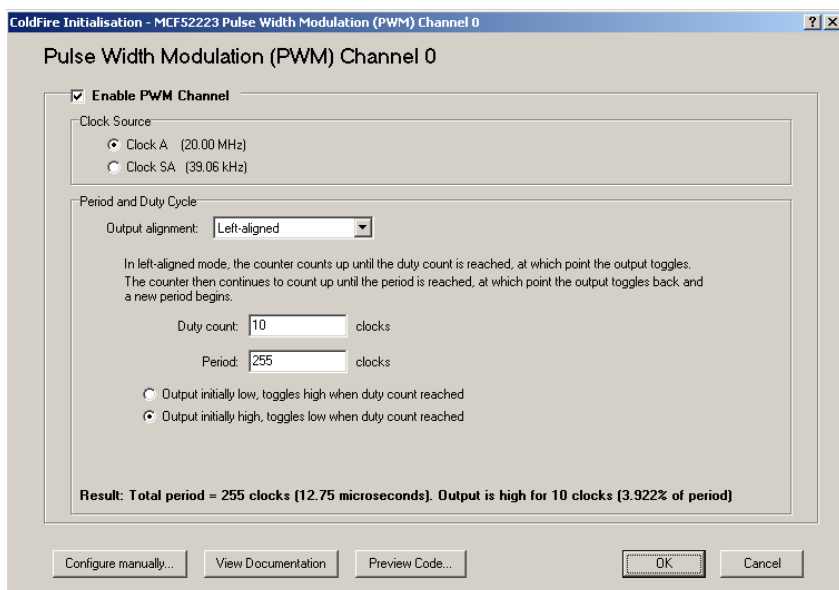


Figure 15. Configuring PWM Channel 0

```

ColdFire Initialisation - untitled
File Edit
/*****
 * init_pwm - Pulse Width Modulation (PWM) Module
 *****/
static void init_pwm (void)
{
    /* Clock A frequency = 20.00 MHz
    PWM continues to run in freeze mode
    PWM continues to run in wait mode

    Settings for PWM Channel 0:
    Clock source is Clock A
    Left-Aligned output mode
    Period = 255 clocks (12.75 microseconds)
    High for 10 clocks per period (3.922% duty cycle)
    */
    MCF_PWM_PWMPER0 = MCF_PWM_PWMPER_PERIOD(0xff);
    MCF_PWM_PWMDTY0 = MCF_PWM_PWMDTY_DUTY(0xa);

    MCF_PWM_PWMPOL = MCF_PWM_PWMPOL_PPOL0;
    MCF_PWM_PWMCLK = 0;
    MCF_PWM_PWMPRCLK = MCF_PWM_PWMPRCLK_PCKA(0x1);
    MCF_PWM_PWMCAE = 0;
    MCF_PWM_PWMCTL = 0;
    MCF_PWM_PWMSCLA = 0;
    MCF_PWM_PWMSCLB = 0;
    MCF_PWM_PWME = MCF_PWM_PWME_PWMEO;
    MCF_PWM_PWMSDN = 0;
}
    
```

Figure 16. Results from Clicking on the Preview Code Button

### 6.1.2 Modulating the PWM’s Duty Cycle

The PWM duty cycle must be updated every 125µs (for a 8 Khz sample rate). This is done on the ColdFire using a PIT (programmable interval timer). The PIT is configured to interrupt every 125µs. During the interrupt is copies a byte from a ring buffer (described in the next section) to the PWM duty cycle register.

#### 6.1.2.1 Initializing the PIT

PIT1 is used because the CMX USB stack uses PIT0 for internal timeouts.

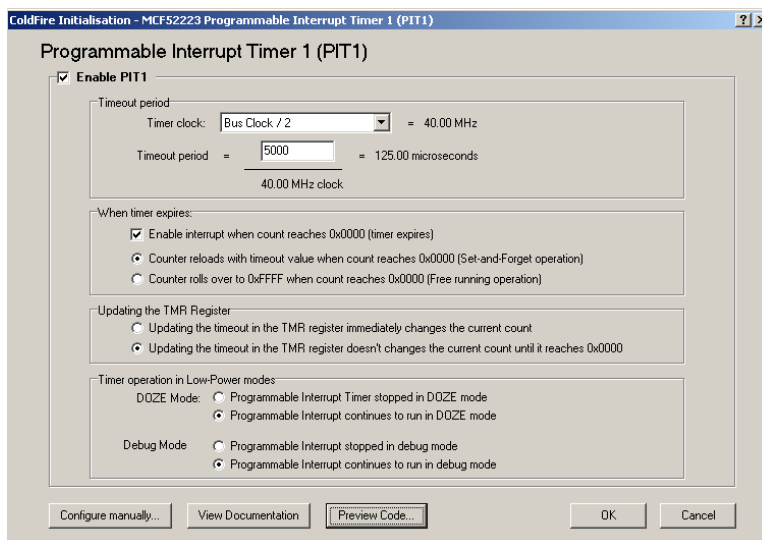


Figure 17. Using CFInit to Configure PIT



Clicking on preview code results in init code ready to be cut and past into source.

```

ColdFire Initialisation - untitled
File Edit
/* PIT1 enabled
PIT1 timeout period = 125.00 microseconds
Timeout value reloaded when counter reaches zero
Writing to PMR replaces value in PIT counter when count reaches 0x0000
Interrupt when timer expires is enabled
Timer continues to run in DOZE mode
Timer continues to run in Debug mode
*/

/* Set OVM bit so first PMR update is immediate */
MCF_PIT1_PCSR |= MCF_PIT_PCSR_OVM;

/* Update PMR and then enable timer */
MCF_PIT1_PMR = MCF_PIT_PMR_FM(0x1387);
MCF_PIT1_PCSR = MCF_PIT_PCSR_PIE |
                MCF_PIT_PCSR_RLD |
                MCF_PIT_PCSR_EN;
    
```

### 6.1.2.2 The PIT Interrupt Handler

The PIT timer rolls over every 125 $\mu$ s. When it rolls over an interrupt is generated. The interrupt controller must be configured to enable the PIT1 interrupt, and set its priority. The vector table must also be initialized with a pointer to the new interrupt handler.

```

//
// 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
}

```

The PIT1 interrupt is number 56 in the interrupt controller. This information is available in the reference manual, or from CFIInit.

The interrupt controller vector number is not the same as the vector number in the vector table. The USB stack's vector table includes the interrupt controller vector numbers as comments next to the stubs for unused vectors.

```

vector77:  .long   _irq_handler77 /* PIT0 PIF v55*/
vector78:  .long   _PIT1_it_handler /* PIT1 PIF v56*/
vector79:  .long   _irq_handler79 /* reserved */

```

The PIT interrupt handler simply reads a byte from the ring buffer and writes it to the PWM Duty cycle register. Then it increments the out pointer for the ring buffer.

## 6.2 Reading WAV Files from the Flash Stick

Data from the flash stick is read in 200 byte chunks. The data is then copied into a ring buffer one byte at a time. This is done to allow the PIT interrupt and the flash stick reading code to be completely independent with respect to time (no synchronization required).

Data is read from the flash stick using the `f_read()` function.

WAV files are played using the `emgplay <filename>` command. You can also just call this function with a pointer to a filename NULL terminated string.

```

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 Converting Data in Chunks to a Stream (Ring Buffer)

The ring buffer is the secret to audio synchronization. As the name implies, the ring buffer has no start or end, the buffer wraps around on itself. This is done using 2 indexes; a OUT index and a IN index.

Data is put into the buffer using the IN index, and data is taken out of the buffer using the OUT index. The indexes are unsigned bytes, so they naturally wrap at 256 bytes. This is the length of the ring buffer. The OUT index is only read and incremented if it does not equal the IN index.

```

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

```

The IN index is kept one count behind the OUT index. The IN index is never allowed to increment to the OUT index.

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

The code above spins until the OUT index (data\_out) is more than one count away from the IN index (data\_in).

## 6.4 Simple Text to Speech Engine

With the ability to play WAV files, and access to a large amount of storage space (memory sticks up to 2G can be used with the free version of the Mass-Storage stack) the next logical choice was a text to speech processor. Individual words are recorded as WAV files onto the flash stick. The filename used is the word: the word “eric” is stored as eric.wav. A simple function converts the words in a sentence (separated by spaces) into filenames, then plays the filenames.

This has been implemented with the emgsay <sentence> command. Simply create a dictionary with the words you would like to use, copy it to a flash stick, and plug it into the ColdFire or V1 Core. This technology can be used for menus, instructions, alarms, games, ...

### 6.4.1 emgsay Firmware

```
//
// 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];
    }
}
}
```

```

Tera Term - COM1 VT
File Edit Setup Control Window Help

>dir
AP
USERMA~1
WIN98D~1
UBU10.UER
MACARANA.WAU
LONG_H~1.WAU
A.WAU
ADD.WAU
BEUTIF~1.WAU
BY.WAU
COLDFIRE.WAU
COOL.WAU
DEMO.WAU
ENGINE~1.WAU
ERIC.WAU
FIRMWARE.WAU
FREESC~1.WAU
GREGORI.WAU
HELLO.WAU
IS.WAU
MY.WAU
OF.WAU
OWN.WAU
SEMICO~1.WAU
SIMPLE.WAU
SPEECH.WAU
TEXT.WAU
THANK.WAU
THE.WAU
THIS.WAU
TO.WAU
WELCOME.WAU
WHIRLP~1.WAU
WIFE.WAU
WORDS.WAU
WORLD.WAU
WRITTEN.WAU
YOU.WAU
YOUR.WAU
COM.WAU
WWW.WAU
DOT.WAU
EMGWARE.WAU
FROM.WAU
SAY.WAU
TYPE.WAU
IT.WAU
I.WAU

>emgsay i type it you say it

Playing i.wav - Done.

Playing type.wav - Done.

Playing it.wav - Done.

Playing you.wav - Done.

Playing say.wav - Done.

Playing it.wav - Done.

>

```

Figure 18. Directory of Dictionary on Flash Stick, and Using emgsay Command

THIS PAGE IS INTENTIONALLY BLANK

**How to Reach Us:****Home Page:**

[www.freescale.com](http://www.freescale.com)

**Web Support:**

<http://www.freescale.com/support>

**USA/Europe or Locations Not Listed:**

Freescale Semiconductor, Inc.  
Technical Information Center, EL516  
2100 East Elliot Road  
Tempe, Arizona 85284  
+1-800-521-6274 or +1-480-768-2130  
[www.freescale.com/support](http://www.freescale.com/support)

**Europe, Middle East, and Africa:**

Freescale Halbleiter Deutschland GmbH  
Technical Information Center  
Schatzbogen 7  
81829 Muenchen, Germany  
+44 1296 380 456 (English)  
+46 8 52200080 (English)  
+49 89 92103 559 (German)  
+33 1 69 35 48 48 (French)  
[www.freescale.com/support](http://www.freescale.com/support)

**Japan:**

Freescale Semiconductor Japan Ltd.  
Headquarters  
ARCO Tower 15F  
1-8-1, Shimo-Meguro, Meguro-ku,  
Tokyo 153-0064  
Japan  
0120 191014 or +81 3 5437 9125  
[support.japan@freescale.com](mailto:support.japan@freescale.com)

**Asia/Pacific:**

Freescale Semiconductor Hong Kong Ltd.  
Technical Information Center  
2 Dai King Street  
Tai Po Industrial Estate  
Tai Po, N.T., Hong Kong  
+800 2666 8080  
[support.asia@freescale.com](mailto:support.asia@freescale.com)

**For Literature Requests Only:**

Freescale Semiconductor Literature Distribution Center  
P.O. Box 5405  
Denver, Colorado 80217  
1-800-441-2447 or 303-675-2140  
Fax: 303-675-2150  
[LDCForFreescaleSemiconductor@hibbertgroup.com](mailto:LDCForFreescaleSemiconductor@hibbertgroup.com)

Document Number: AN3523  
Rev. 0  
09/2007

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals", must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

RoHS-compliant and/or Pb-free versions of Freescale products have the functionality and electrical characteristics as their non-RoHS-compliant and/or non-Pb-free counterparts. For further information, see <http://www.freescale.com> or contact your Freescale sales representative.

For information on Freescale's Environmental Products program, go to <http://www.freescale.com/epp>.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org  
© Freescale Semiconductor, Inc. 2007. All rights reserved.