An Integrated Expansion USB Bridge for ISA Data Acquisition Card

Ni Zurong
Department of Physics
Xiamen University
Xiamen, Fujian, China
(0086592-2182459; e-mail: zrni@xmu.edu.cn)

Zhou Xiang
Department of Physics
Xiamen University
Xiamen, Fujian, China

***Abstract*—An integrated expansion bus bridge is disclosed that couples the traditional ISA (Industry Standard Architecture) data transfer bus to the USB (Universal Serial Bus), which was implemented in the external placement of a ISA data acquisition card. The main transfer function was achieved by the chip of Cypress, EZ-USB AN2131Q, and the performance of the chip, implementation of driver program development, firmware design and application program communication are presented in detail. The advanced performance of USB with Hot Plug In, Plug and Play, easily to expand and engrossing less system resource was shown in the system. Such technology can also be extended efficiently in the field of industry measurement and control, where the ISA bus peripheral card has a wide application.**

***Index Terms*—AN2131Q, data acquisition, ISA, USB**

I. INTRODUCTION

The availability of a variety of standard expansion bus systems within the PC environment must surely be the single most crucial factor in harnessing the power of the machine. The original PC expansion bus supported an 8-bit data path (ISA) but the bus was soon extended to support the full 16-bit bus (EISA). Despite the emergence of PCI as an enhanced bus standard, many ISA and EISA cards are still in current use in control and instrumentation systems, and are still available from a number of suppliers.

Things had changed when came the Universal Serial Bus (USB) Offering true plug-and-play capability[1], and the Universal Serial Bus (USB) has become the *de-facto* future standard for the interconnection of a host computer to a wide range of simultaneously accessible peripheral devices that share the available USB bandwidth through a host-scheduled, token-based protocol. The Universal Serial Bus was originally specified as an industry-standard extension to the PC architecture with a focus on Computer Telephony Integration (CTI), consumer, and productivity applications. Furthermore, unlike most other forms of computer bus, USB allows peripherals to be attached, configured, used, and detached while the host and other peripherals are in operation. The main features (and notable advantages compared with serial-port data transfer) of USB are shown as follows:

- Devices can be identified and configured automatically.
- Peripheral devices can be 'hot-plugged' and 'hot un-plugged'.
- Conforms with standard plug-and-play architecture.
- Supports transfer of multiple data and message streams between the host and devices.
- Flow control for buffer handling is built into the protocol.

With the development of personal computer technology, more PCs are not equipped with the ISA slots. Another main disadvantage is the location of the ISA connector; they are inside the PC case, which have to be opened, clamping screws undone, boards plugged exactly right into connectors and then the system reassembled. The internal location also causes the electronic magnetic interference (EMI) more complex in the PC case and higher requirement for power supply. All these aspects makes it necessary to build a USB-to-ISA bridge, which supplies bus transmit connection for abundant A/D, D/A, DI/DO ISA expansion card.

II. USB CONTROLLER CHIP AND ITS ARCHITECTURE

A. USB Controller Chip

When a USB controller need to be selected for a project, the good news is that there are plenty of chips to choose from, such as Philips PDIUSBD series, National Semiconductor's USBN9602, or with embedded CPUs, Microchip Technology's PICmicro microcontrollers PIC18F4550, Cypress Semiconductor's EZ-USB family and enCoRe II series, Freescale Semiconductor's MC68HC08 family. As with any project involving embedded systems, the decision depends on what functions the chip has to perform, cost, availability, and ease of development. Because of the USB protocol complexity, every USB device must have the intelligence to implement the USB protocol. The device is able to detect and respond to requests and other events at its USB port, and provide data to be sent on the bus and retrieve data received on the bus.

The Cypress Semiconductor EZ-USB AN2100 series chip packs the intelligence required by a USB peripheral interface into a compact integrated circuit. It provides a soft (RAM-based) solution that allows unlimited configuration and upgrades, and also delivers full USB throughput. Designs that use EZ-USB are not limited by number of endpoints, buffer

This work was supported in part by the Fujian Major Science and Technology Project under Grant 2007HZ0005-2 and Undergraduate Innovative Experiment Project of Xiamen University.

Authorized licensed use limited to: University of Washington Libraries. Downloaded on June 16, 2009 at 14:56 from IEEE Xplore. Restrictions apply.

sizes, or transfer speeds. Above all the chip does much of the USB housekeeping in the EZ-USB core, simplifying code and accelerating the USB learning curve.

B. *Architecture of Cypress EZ-USB*

As Figure 1-1 illustrates, an integrated USB transceiver connects to the USB bus pins D+ and D-. A Serial Interface Engine (SIE) decodes and encodes the serial data and performs error correction, bit stuffing, and other signaling-level details required by USB, and ultimately transfers data bytes to and from the USB interface. The internal microprocessor is enhanced 8051 with fast execution time and added features. It uses internal RAM for program and data storage, making the EZ-USB family a *soft* solution. The EZ-USB family uses an enhanced SIE/USB interface (called the "USB Core") which has the intelligence to function as a full USB device even before the 8051. The enhanced core simplifies 8051 code by implementing much of the USB protocol itself.

In addition to the 24 IO pins, it contains a 16-bit address bus and an 8-bit data bus for external memory expansion. A special

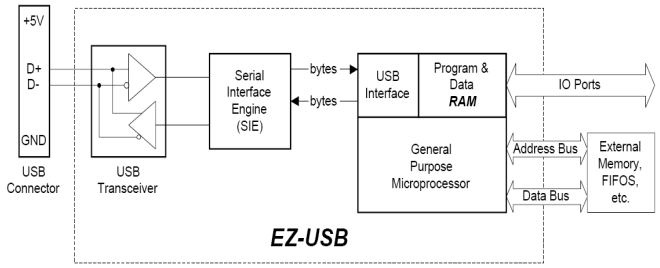

Fig. 1. AN2131Q simplified block diagram

fast transfer mode moves data directly between external logic and internal USB FIFOs. The fast transfer mode, along with abundant endpoint resources, allows the EZ-USB family to support transfer bandwidths beyond the maximum required by the *Universal Serial Bus Specification Version 1.1*.

III. HARDWARE DESIGN OF ISA-TO-USB BRIDGE

We'll present a design of ISA-to-USB bridge which migrates a data-acquisition device from ISA bus to USB. The following function should be achieved by the transfer circuit.

- Receive and transmit the USB format data from host to data-acquisition card, while the data format is transferred to ISA style demanded by the peripheral device.
- Receive and transmit the ISA bus data from peripheral device to host, while the packaged data is compliance to the USB protocol.

The 12-bit data acquisition device provides sixteen analog inputs, one analog outputs, and 24 digital I/O-all on a single ISA card. The ISA-to-USB bridge accept the command signal from the host and translate it to control the ISA peripheral card by the hardware firmware in the USB microcontroller. The whole circuit is comprised of the Cypress EZ-USB AN2131Q, power supply and transfer model, USB firmware load circuit and USB external wakeup circuit, crystal resonant circuit and bus junction.

A. *EZ-USB AN2131Q Firmware Load Circuit*

The EZ-USB I^{2}C controller serves two purposes. First, when the EZ-USB chip comes out of reset, the EZ-USB core makes a decision about how to enumerate based on the contents

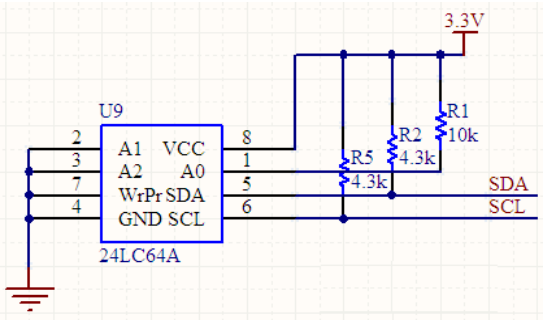

Fig. 2. AN2131Q firmware load circuit

of an external EEPROM on its I^{2}C bus. Second, once the 8051 is up and running, the 8051 can access the I^{2}C controller for general-purpose use. This makes a wide range of standard I^{2}C peripherals available to an EZ-USB system.

B. *EZ-USB AN2131Q Address Latch Circuit*

The data acquisition card occupies 8 data bus and ten address bus, The address latch DM74LS373 needed to be applied for the time multiplexing of lower 8 bits address and data bus.

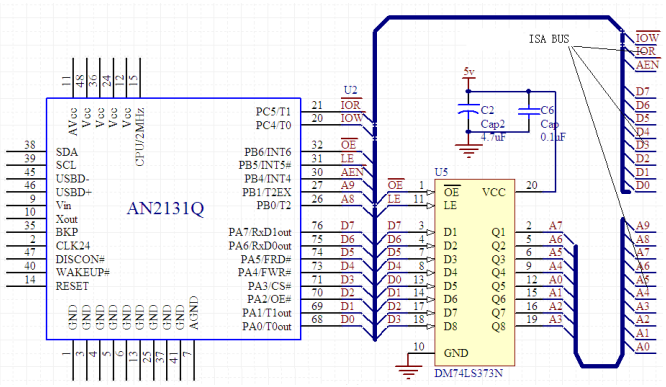

Fig. 3. AN2131Q address latch circuit

IV. SOFTWARE DESIGN OF USB-TO-ISA BRIDGE

A. *Firmware Architecture of Bridge*

The exiting program architecture of EZ-USB can be easily applied for the development of peripheral device based on the Cypress EZ-USB controller chip. It is provided with the 8051 code for the initialization of EZ-USB, the execution of USB device standard request and the power management during the suspend of USB controller. The only things left done is the description of the USB logic architecture and its function. The Firmware Frameworks implements the basic functionality required of a USB compliant peripheral device. By linking code associated with the Anchor provided hooks, it is possible to incrementally build a fully functional device. The Frameworks

Authorized licensed use limited to: University of Washington Libraries. Downloaded on June 16, 2009 at 14:56 from IEEE Xplore. Restrictions apply.

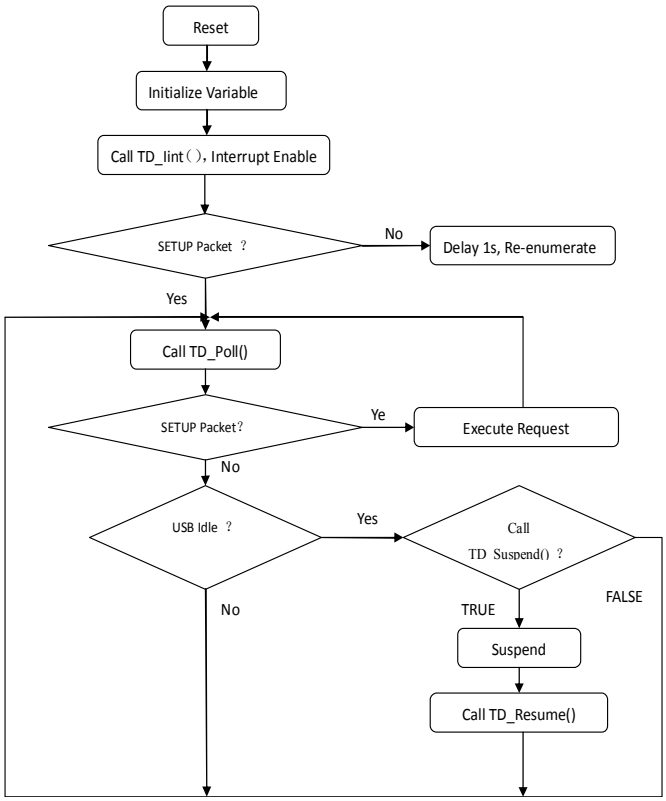

Fig. 4. Firmware architecture of the USB bridge

implements a simple co-operative tasking executive (See figure 4). At start-up, the Frameworks first initializes all of its internal state variables. It then calls the user initialization function TD_Init(). Upon return, the Frameworks initializes the USB interface to the unconfigured state and enables interrupts. It then begins to re-numerate the device at one second intervals until a setup packet is received on endpoint 0. Once a setup packet is detected the Frameworks starts the cooperative tasking dispatcher. The task dispatcher repeatedly performs the following tasks in the given order.

- Calls user function TD_Poll().
- Determines if a standard device request is pending. If so, it parses the received command and responds accordingly. By default, it will implement the USB defined response for the given standard request. However, the Frameworks provides hooks into the parser that allow the user code to handle and/or over-ride specific device requests.
- Determines if the USB core has reported a USB suspend event. If so, it calls the user function TD_Suspend(). Upon a successful return, it tests for a resume event. If one is not detected, it then places the processor in suspend mode. When a resume event is detected it calls the user function TD_Resume() and continues to step 3. Upon an unsuccessful return from TD_Suspend() continues to step 3.

B. Function of the Request Command

The Firmware frameworks was written using the Keil C51 C compiler and tools and provides function hooks to simplify the addition of user code. Two different request B1 and B2 are defined in order to implement the control transfers by EZ-USB. Such request can be identified by the definition appended in the head file EZUSB.h, shown as follows:

```
#define SC_ISAWrite 0xB1
#define SC_ISARead 0xB2
```

Two sub-functions are appended in the request correspondent function SetupCommand() of the main program FW.c, shown as follows:

```
BOOL DR_ISAWrite(void)   // Send Data to ISA port
BOOL DR_ISARead(void)    // Read Data from ISA port
```

The important section of the sub-functions is list below in brief.

```
BOOL DR_ISAWrite(void)
{
  unsigned char aah,aal,cc;
  aah=SETUPDAT[1];
  aal= SETUPDAT[2];
  cc= SETUPDAT[3];
  OUT0BUF[0]=0xB1;
  OUT0BC=1;
  OEA=0xFF;
  OEB=0xFF;
  OEC=0xFF;
  OUTC=0xFF;
  OUTB=0x30;
  OUTA=aal;
  OUTB=0x03&aah;
  OUTC=0xEF;
  OUTA=cc;
  OUTB=0x70;
  OUTC=0xFF;
  EP0CS |= bmBIT1;
  return(TRUE);
}
BOOL DR_ISARead(void)
{
    unsigned char aah,aal,cc;
    aah=SETUPDAT[3];
    aal=SETUPDAT[2];
    cc=0x20; //write the control word 0x20
    ISA_Write(aah,aal+4,cc);
    cc=0x00; //Select the CH00
    ISA_Write(aah,aal,cc);
    cc=0x00; //Start tran
    ISA_Write(aah,aal+1,cc);
    cc=ISA_Read(aah,aal+1); //Judge the finished bit
    if((cc&0x80)==0x80)
    { aah=0x02;
      aal=0x82;
      IN0BUF[0]=ISA_Read(aah,aal+2);
      aah=0x02;
      aal=0x83;
      IN0BUF[1]=ISA_Read(aah,aal+3);
    }
  EP0CS |= bmBIT1;
  return(TRUE);
}
```

Authorized licensed use limited to: University of Washington Libraries. Downloaded on June 16, 2009 at 14:56 from IEEE Xplore. Restrictions apply.

C. WDM Driver of the Bridge

A device driver is a software component that enables applications to access a hardware device. A device driver insulates applications, the programs that users run, from having to know details about the physical connections, signals, and protocols required to communicate with a device. The device driver translates between application-level and hardware -specific code. Applications communicate with device drivers using functions supported by the operating system. The hardware-specific code handles the protocols needed to access the peripheral's circuits, including detecting the states of status signals and toggling control signals at appropriate times. A WDM driver is a PnP driver that also understands power management protocols and is source-compatible with both windows 98 and windows 2000. The WDM drivers for USB devices use a layered driver model (see fig. 5) where each driver in a series performs a portion of the communication. Dividing communications into layers is efficient because devices that have tasks in common can use the same driver for those tasks. The top layer contains a function driver that manages communications between applications and lower-level bus drivers. It understands all the details about how to make the hardware work. It's responsible for initiating I/O operations, for handling the interrupts that occur when those operations finish, and for providing a way for the end user to exercise any control over the device that might be appropriate. The bottom layer contains bus drivers that manage communications between the client driver and the hardware. One or more filter drivers may supplement the client and bus drivers. An upper-level filter driver can monitor and modify communications between applications and a client driver. A lower-level filter driver can monitor and modify communications between a client driver and the bus drivers.

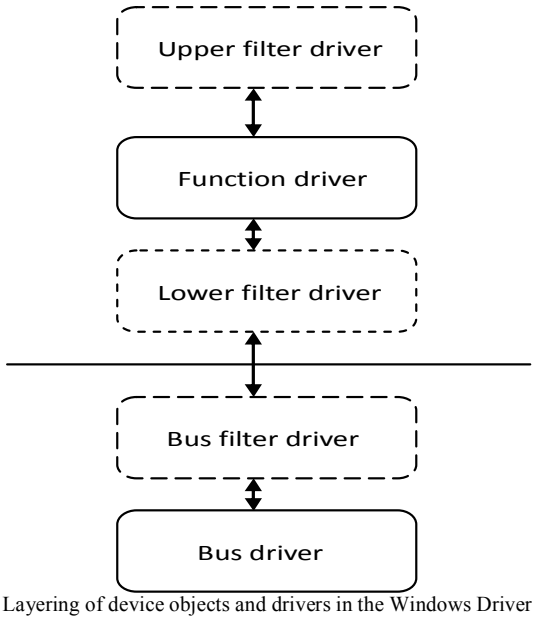

Fig. 5. Layering of device objects and drivers in the Windows Driver Model

The EZ-USB General Purpose Driver (EZUSB GPD) is a general purpose device driver that can be used to interface with an EZ-USB based peripheral. The driver provides a user mode interface to common USB device requests and data transfers. it provides example code for USB device driver development. A custom driver or mini-driver can be created using the general purpose driver as a starting point. The USB device driver gets loaded by its Vendor ID (VID) and Product ID (PID) which are reported to Windows in the device descriptor. Windows uses the VID and PID to find the appropriate device driver in a INF file, which ties a VID/PID combination to a specific driver. The INF file is used to associate a device with a device driver the first time the device is attached to the system. This information is then stored in the system registry so that subsequent device attachments will complete faster and not require user interaction. Some sections of our INF file are shown below:

```
[Cypress]
%USB\VID_0547&PID_2131.DeviceDesc%=EZUSB.D
ev, USB\VID_0547&PID_2131
[EZUSB.Files.Ext]
ezusb.sys
[EZUSB.Files.Inf]
ezusbw2k.Inf
[Strings]
Cypress="Cypress Semiconductor"
USB\VID_0547&PID_2131.DeviceDesc="Cypress
EZ-USB (2131Q/2131S/2135S) - USB-ISA"
EZUSB.SvcDesc="Cypress General Purpose USB Driver
(ezusb.sys)"
```

All User mode access to the EZ-USB GPD is through I/O Control calls. A user mode application first gets a handle to the device driver via a call to the Win32 function CreateFile(). The user mode application then uses the Win32 function DeviceIoControl() to submit an I/O control code and related input and output buffers to the driver through the handle returned by CreateFile(). The EZ-USB GPD can communicate with multiple EZ-USB devices. For each EZ-USB device attached to the host, the driver creates a symbolic link of the form ezusb-*i*, where *i* is an instance index starting at 0. The symbolic link name is used when calling CreateFile() to get a handle to the device driver. What CreateFile() is really doing is getting a handle to the device object created by the device driver. The following code sample demonstrates getting a handle to EZ-USB device ezusb-0.

```
HANDLE DeviceHandle;
DeviceHandle = CreateFile("\\\\.\\ezusb-0",
GENERIC_WRITE,
FILE_SHARE_WRITE,
NULL,
OPEN_EXISTING,
0,
NULL);
```

User mode applications use the Win32 function DeviceIoControl() to send requests to device drivers. The following code sample demonstrates submitting a request to the EZ-USB GPD to read the device's USB Device Descriptor. It uses the DeviceHandle returned in the CreatFile().

```
bResult = DeviceIoControl (
HANDLE DeviceHandle,
DWORD dwIoControlCode,
LPVOID lpInBuffer,
DWORD nInBufferSize,
LPVOID lpOutBuffer,
DWORD nOutBufferSize,
LPDWORD lpByteReturned,
```

Authorized licensed use limited to: University of Washington Libraries. Downloaded on June 16, 2009 at 14:56 from IEEE Xplore. Restrictions apply.

LPOVERLAPPED lpOverLapped,);

The I/O control code (IOCTL), dwIoControlCode, and their corresponding input/output structures are defined in the Anchor Chips provided header file EZUSBSYS.H. Standard USB structures are defined in the file USB100.H which is provided with the WDM DDK. The IOCTL_EZUSB _VENDOR_OR_CLASS_REQUEST was used in the ISA bridge driver to send a Vendor or Class specific request to the control endpoint.

D. Dynamic Link Library

A dynamic-link library (DLL) is an executable file that acts as a shared library of functions. Dynamic linking provides a way for a process to call a function that is not part of its executable code. The executable code for the function is located in a DLL, which contains one or more functions that are compiled, linked, and stored separately from the processes that use them. DLLs also facilitate the sharing of data and resources. Multiple applications can simultaneously access the contents of a single copy of a DLL in memory.

Using dynamic linking instead of static linking offers several advantages. DLLs save memory, reduce swapping, save disk space, upgrade easier, provide after-market support, provide a mechanism to extend the MFC library classes, support multi-language programs, and ease the creation of international versions.

A DLL file contains an exports table. The exports table contains the name of every function that the DLL exports to other executables. These functions are the entry points into the DLL; only the functions in the exports table can be accessed by other executables. Any other functions in the DLL are private to the DLL. Functions can be exported from a DLL using two methods:

- Create a module definition (.def) file and use the .def file when building the DLL. Use this approach when exporting functions from the DLL by ordinal rather than by name.
- Use the keyword __declspec(dllexport) in the function's definition.

The main port of exported function Outp() and Inp() were defined as follows:

```
extern "C" int _stdcall Outp(int address,int content)
    {
     HANDLE hDriver=NULL;
     hDriver=hOpenDevice();
     if(!hDriver) return 0;
     if(!VendorOutCmmd(hDriver,address,content))
     return 1;
     if(!CloseDevice(hDriver)) return 2;
     return 3;
    }
extern "C" int _stdcall Inp(int address)
    {
     HANDLE hDriver=NULL;
     hDriver=hOpenDevice();
     if(!hDriver) return 0;
     if(!VendorInCmmd(hDriver,address)) return 1;
     if(!CloseDevice(hDriver)) return 2;
     return 3;
    }
```

The function VendorOutCmmd() and VendorInCmmd() shown above were used to transfer the write and read request to the ISA bus, both functions are same as the function VendorCmmd().

```
int VendorCmmd(HANDLE hDriver, int address, int data)
{
  VENDOR_OR_CLASS_REQUEST_CONTROL vmd;
  vmd.direction=0;//out
  vmd.requestType =2;//2 vendor
  vmd.recepient=0;// recipient(0=device, 1=interface,
                                    2=endpoint, 3=other)
  vmd.requestTypeReservedBits=0x40;
  vmd.request=0xb1;// 0xb1=write, 0xb2=read
  vmd.value=address;
  vmd.index=0;
  UINT inLength=sizeof(VENDOR_OR_CLASS_REQUEST
                                        _CONTROL);
  OutBuffer=data;
  UINT BytesTxd;
   if( !DeviceIoControl(hDriver,
   IOCTL_EZUSB_VENDOR_OR_CLASS_REQUEST,
              (LPVOID )&vmd,
              inLength,
              &OutBuffer,
               1,//
              (LPDWORD )&BytesTxd,
              NULL))
    {
       return 0 ;
    }
         else
         return 1;
}
```

E. Application Software of the Bridge

LabVIEW constitutes a graphical programming environment that allows one to design and analyze a custom system in a shorter time as compared to text-based programming environments. LabVIEW graphical programs are called virtual instruments(VIs). VIs run based on the concept of data flow programming. This means multiple operations to be performed in parallel since its execution is determined by the flow of data and not by sequential lines of code. In LabVIEW Programming Environment, the ability to reuse code written in other languages as well as being able to be called from external code are also one of the most important aspects. The external DLLs list above can be called from LabVIEW by using the Call Library Function Node, shown like fig. 6. An AD conversion example after interpolation is illustrated in fig. 7.

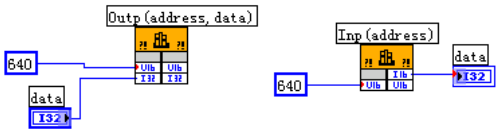

Fig. 6. LABVIEW call library function

Authorized licensed use limited to: University of Washington Libraries. Downloaded on June 16, 2009 at 14:56 from IEEE Xplore. Restrictions apply.

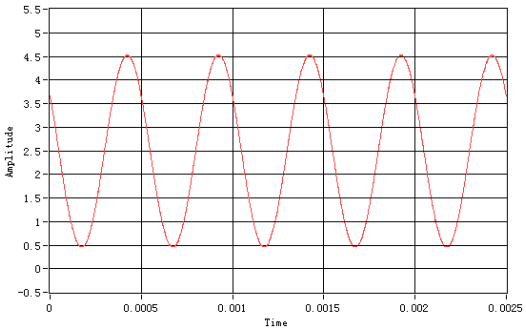

Fig. 7. AD conversion data after interpolation

REFERENCES

[1] The Universal Serial Bus Specification Version 2.0 . Available: http://usb.org.

[2] EZ-USB FX2 Manual Technical Reference[Z], Cypress Semiconductor, Available: http://www.cypress.com, 2002.

[3] John Hyde. *USB Design by Example a practical Guide to Building I/O Device. Intel Press. 2nd Edition May 2002*

[4] Wynn, R. , "Plug&play, networked data acquisition systems reduce test costs", Computing & Control Engineering Journal, Vol. 16(1), 2005 PP. :23 - 25

[5] Bi Bo; Sun Shuying; Wang Chunping, "Design of Data Acquisition Equipment Based on USB," Electronic Measurement and Instruments, 2007. ICEMI '07. 8th International Conference, China,2007, PP.:1-866 - 1-869.

[6] Wang Weiguo; Liu Hongbo; Yang Fangtao, "Design and Implementation of Automatic Test Instrument for Digital Circuits Based on USB 2.0 Interface," Electronic Measurement and Instruments, 2007. ICEMI '07. 8th International Conference, China,2007, PP.:4-758 - 4-761.

[7] Giron, C.; Rodriguez, F.J.; Huerta, F., "Implementing high speed communication buses for a FPGA-DSP architecture for digital control of power electronics", IEEE International Symposium on Intelligent Signal Processing, 2007. WISP 2007. Oct. 2007, PP.:1 - 6

[8] Darren Abramson, "Method and Apparatus for Connecting Expansion Buses to a Peripheral Component Interconnect Bus (Patent style)," U.S. Patent 5 951 667, Sep. 14, 1999.

Authorized licensed use limited to: University of Washington Libraries. Downloaded on June 16, 2009 at 14:56 from IEEE Xplore. Restrictions apply.