IOP Conf. Series: Materials Science and Engineering **449** (2018) 012026 doi:10.1088/1757-899X/449/1/012026

Design of Universal PCIe Interface Module Based on Vs

Tingting Du[1] and Qingzhong Jia[1]

[1]Key Ministry Education Laboratory of Dynamics and Control of Flight Vehicle, Beijing Institute of Technology, BIT, Beijing, China

bitdtt6979@163.com

Abstract. Due to wide variety of interfaces and complex bus protocols, the general detection system has problems such as high bus usage threshold and poor portability of the host computer software, which greatly reduces the user's development efficiency of the detection system. In response to the issue, this paper designs a universal interface module for PCIe based on vs. The module provides unified functional interfaces, through which the data transmission of all bus devices can be easily and reliably controlled. In this paper, the design idea of the universal interface module is explained. Finally, the module is illustrated and the result shows that the designed scheme is feasible.

1. Introduction

With the development of computer technology, the detection system consisting of the main control computer and various expansion modules is widely used in various fields[1]. Due to variety of interfaces of the measured object, the detection system is required to be able to integrate multiple interfaces, and these interfaces need to be able to change as the measured object changes. At present, the interconnected buses with high performance and wide application are mainly Gigabit Ethernet, PCIe, RapidIO, etc[2]. Among them, PCIe (peripheral component interconnect express), a serial computer expansion bus standard, is widely used in the field of embedded detection because of its high bandwidth, optional link number and high transmission speed[3].

For specific application of bus interfaces, each module manufacturer provides mature, complete and powerful APIs to meet various possible needs of transmitting data. However, in most detection systems, only a small part functions of bus interfaces are used, the complicated API parameters and the interfaces that cannot be transplanted due to different application protocols impose high requirements on developers of the detection system. At the same time, the interface matches the respective PC software, so the diversification of interfaces in the detection system results in poor software portability, a large amount of repetitive work for the development of the PC software and high development cost of the detection system. Therefore, how to improve program portability[4], reduce system development costs, and generalize interface applications have become some of the focuses of current research for computer detection systems.

By using the design pattern idea[5] and the technology of virtual function overloading, this paper establishes a unified user-oriented interface model and studies the generalized design of bus interface. The purpose is to make bus interfaces of the detection system intuitive and easy to use, improve the scalability and portability of system software.

2. Universal interface model establishment

Considering the generality requirement of interfaces, a universal interface model is established as shown in figure 1.

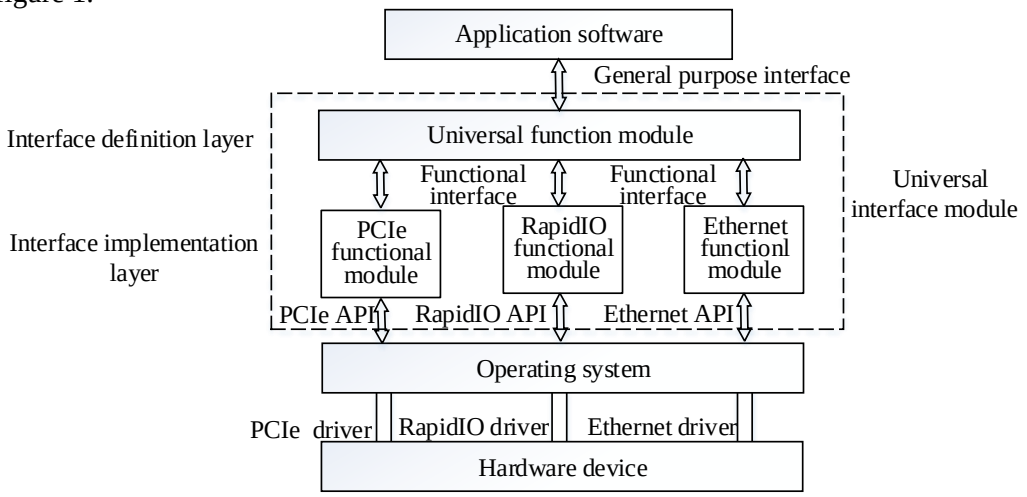

Figure 1. Universal interface model.

The whole model consists of four parts, in which the application software is responsible for human-computer interaction, and the hardware devices follow the bus protocols to achieve their respective functions. These two parts are not described in detail in this paper. As a link between the hardware device and the operating system, driver enables the host computer to operate the hardware device indirectly by calling APIs (Application Program Interface) provided by the operating system. Referring to the seven-layer architecture of the Open System Interconnect (OSI)[6], the universal interface module designed in this paper is equivalent to the top layer of OSI, the application layer. Its function is to provide the service interface to the users directly and complete the connection between the application and the network operating system.

As a focus of this design, the universal interface module can be divided into two layers. The upper interface definition layer abstracts and defines various operations and provides general purpose interfaces to users. In addition to basic operation interfaces for data transmission, it also includes additional function interfaces and user-setting interfaces. Through this layer, the application can realize the operation of interfaces of various buses without differentiation, and users can control the data transmission process intuitively, simply and reliably.

The lower interface implementation layer distinguishes the specific application, and further designs and encapsulates the basic APIs provided by the operating system into functional interfaces for the upper layer to call, thereby completing the concrete implementation of the general purpose interfaces defined by the upper layer. This layer further designs and improves the original reading and writing functions of the respective bus. Taking PCIe bus as an example, this layer implements functions such as periodic transmission, packet transmission, task scheduling and fault tolerance mechanism setting to ensure reliable data transmission. The interface definition layer and the interface implementation layer together form a bridge between users and different bus devices.

3. Universal interface module design

The universal interface module is divided into an interface definition layer and an interface implementation layer. Taking PCIe, RapidIO and Ethernet protocols as examples, the interface definition layer introduces generality and functionality. The design and packaging ideas of this layer are illustrated through the class diagram and important codes. For the interface implementation layer, it takes PCIe as an example to describe how to further encapsulate the operating system APIs into functional interfaces through working timing diagrams, thereby realizing the concrete implementation of general purpose interfaces reliably.

3.1. Interface definition layer

Analogous to the black box, this layer shields diversity of different protocols, abstracts APIs under different protocols into general purpose interfaces and provides them to the users, realizing that the application does not differentiate the operations of peripheral no matter which protocol is followed. General purpose interfaces defined by this layer are shown in figure 2.

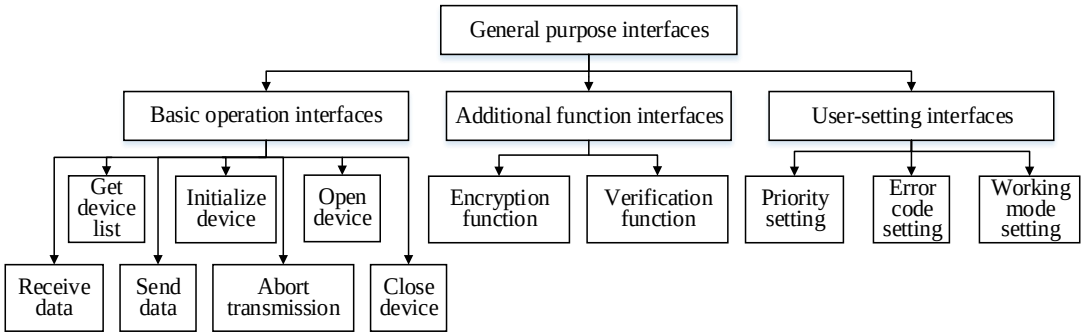

Figure 2. General purpose interfaces.

Through the overloading of above general purpose interfaces, the specific problems solved by this layer include: establishment of unified user-oriented interface model, parameterized design of interface configuration, introduction of general additional functions, design of task scheduling and working pattern. These key issues will be explained in the next section.

3.1.1. Unified user-oriented interface model. According to the defined general purpose interfaces, the main class diagram of the unified user-oriented interface model is shown in figure 3.

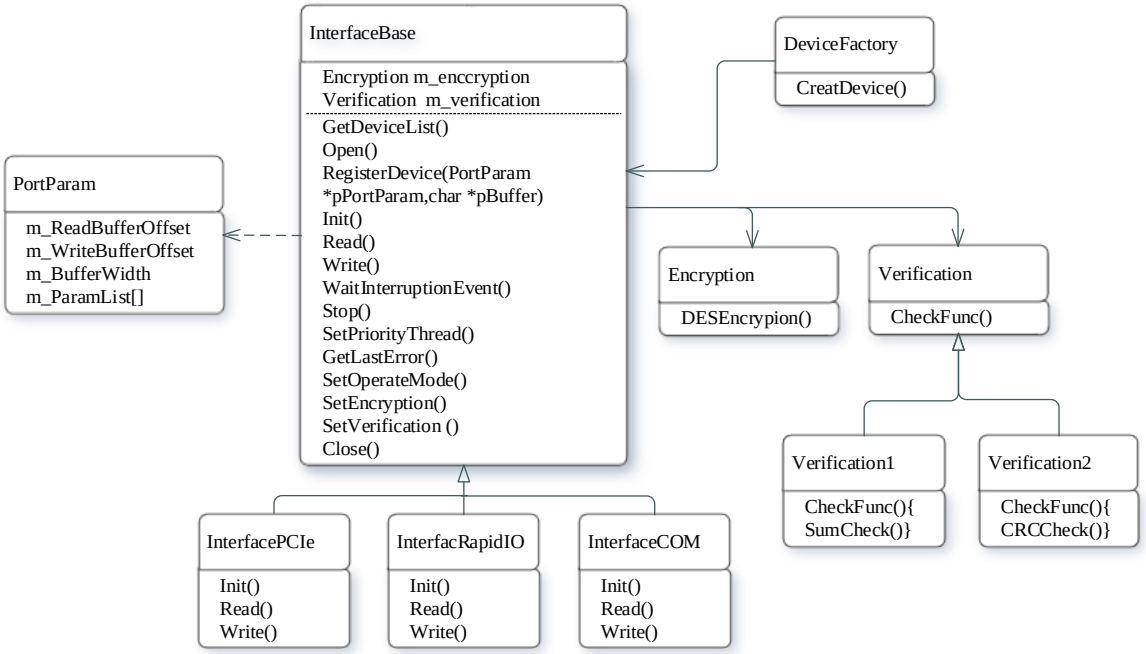

Figure 3. Class diagram of the interface definition layer.

The InterfaceBase class in the class diagram is the base one for all classes and it encapsulates general purpose interfaces defined for users in the data transmission process. Under the premise of satisfying the versatility, the class simplifies interfaces as much as possible, reduces the number of interfaces, lowers the threshold of use and facilitates users to operate the interfaces intuitively as well

as control the entire data transmission process. These interfaces include obtaining the current device list, opening the device, initializing, reading, writing, aborting the transmission, setting the task priority, obtaining the error code, setting the working mode, encrypting, verifying, and turning off the device, which embody the "commonness" of interfaces. The "personality" embodied by different devices is implemented by the factory mode, that is, the DeviceFactory class in the figure 3.

After executing GetDeviceList() in the base class, we can obtain the flag word and the specific number device_num from zero without repetition corresponding to each device. According to the flag word (PCIe device is "PCIe"), the CreateDevice() method in the DeviceFactory class instantiates the InterfacePCIe object, thereby completing the overloading of virtual function under the PCIe protocol. The factory mode makes it unnecessary to change functional functions of each bus class inherited by the base class every time the device is replaced, just instantiating the corresponding object, thereby reducing the amount of code modification and repetition, and achieving the purpose of separating "realization" and "change". Because functional modules of different buses have structural similarities, the PCIe is used for description. The specific implementation code takes c++ as an example:

```
   class  DeviceFactory   //The DeviceFactory class
{
Public:
Void CreateDevice(string DeviceName)
        {
         InterfaceBace  *p_inter_base;
         case(DeviceName)
               {
                PCIe://Taking PCIe as an example
                InterfacePCIe  m_inter_PCIe;
                p_inter_base=&m_inter_PCIe;
                // Instantiate other bus objects
               }
        }
}
```

By instantiating different interface objects, virtual functions in the base class are overloaded in the corresponding interface subclass, and functions under the respective bus protocol are completed in the overloaded functions, thereby realizing the polymorphism required by the system. In other words, the unified user-oriented interface model completes the conversion of dedicated interfaces to general purpose interfaces.

3.1.2. Interface initialization. The types and amounts of configuration information required by the interfaces of different buses vary. To parameterize the initialization work, the PortParam class in figure 3 is designed as follows:

```
   class PortParam    //The PortParam class
{
public:
long m_ReadBufferOffset;   // Reading buffer offset address
long m_WriteBufferOffset;   // Writing buffer offset address
long m_BufferWidth;   // Buffer size
char  m_ParamList[16];
}
```

There is no method in this class, only the property set to configure the interface information. The reading/writing buffer offset address and the buffer size represent the configuration information shared by all buses. In addition, the 16-byte array m_ParamList[] is declared to store different configuration information required by each bus. For example, CAN requires baud rate and ID number. Ethernet requires IP address and port number. The volume of this array can be adjusted.

All searched bus devices should be registered uniformly before the interfaces are initialized, that is, RegisterDevice(PortParam *pPortParam, char *pBuffer) in the base class. All configuration information obtained from users stores in the pBuffer array. According to the size of attributes in the PortParam class, 0-39 bytes in the array represent the configuration information of interface 0, 40-79 bytes represent of interface 1, and so on. In this function, the same number of *pPortParam objects are instantiated to the number of devices, then the configuration information in the pBuffer array is assigned to the corresponding object. The other array DeviceArray[] with the same capacity as the number of devices is created. The assigned *pPortParam pointers are stored in this array, that is, each element in the array represents the configuration information of one device, and the number of each array element represents the number of device, which corresponds to the device_num. In this way, regardless of the addition and deletion of device, it will not affect the registration and configuration of other devices.

In the subclass, the initialization virtual function is overloaded, that is, Init(char *pDeviceArray, device_num). According to the device_num, we can find the corresponding *pPortParam and then parse the configuration information in it, thereby completing the initialization of the interface. At this point, the generalization processing of the universal interface module is also completed.

3.1.3. General additional function. In order to ensure the reliability and security in data transmission process, the module adds an encryption interface and a verification interface. This design uses the strategic mode to implement the introduction of multiple encryption algorithms and verification algorithms. Through this mode, whether encryption/verification or not and which encryption/verification algorithms are used will not affect the client program, which will achieve high cohesion and low coupling of code, and improve the flexibility of the application.

This mode is scalable and can continue to add verification and encryption algorithm without changing the rest of code. Users can expand the module according to this mode if the system needs to support other data processing functions.

3.1.4. Task scheduling. Because the system has multiple bus interfaces to share the tasks together, in order to achieve smooth and reliable work during the data transmission process, task scheduling among interfaces is essential. In the case where users do not set the priority of interfaces, the tasks enter the queue by default in the order of arrival, and are executed in order. In order to meet the user's own requirements for the bus scheduling order, the InterfaceBase class in figure 2 reserves the SetPriorityThread (PKTHREAD Thread, KPRIORITY Priority) interface, which is convenient for the users to customize the bus priority. The priority of the transmission task in the program should correspond to the user-defined priority. If there is a task command with higher priority during the task transmission process, the corresponding thread should be created to ensure that the data transmission process conforms to the bus priority set by users.

3.1.5. Working pattern. In order to make users more intuitive and more convenient to control the data transmission process of the bus device, general purpose interfaces also include GetLastError() and SetOperateMode(). Between them, GetLastError() is used to obtain the error code. Users can customize the error codes according to the possible errors during transmission process and visually monitor the status of the data transmission through the interface.

SetOperateMode(int type, unsigned int port_period, unsigned int interrupt_period) allows users to set the working mode of interfaces manually. Taking PCIe as an example, the first formal parameter represents the custom mode word, including periodic transmission mode: 1, big data transmission mode: 2, and small data transmission mode:3. For the aperiodic transmission, the second and the third parameters default, that is, the timing transmission period and the interruption period to stop periodic transmission. The program determines the required working mode automatically according to the data size when there is no working mode setting. Once the mode is set, it is forced to execute according to the user’s command so that user has full control over interfaces.

The universal interface module is designed and packaged through the above modes, with clear hierarchy, well-organized code and high maintainability. And the designed interfaces meet the generalized and functional requirements on the basis of ensuring the controllable bus.

3.2. Interface implementation layer

In order to improve the reliability of interfaces, in the interface implementation layer, the inherent APIs of each bus are redesigned and encapsulated into functional interfaces, which can be invoked by the interface definition layer to realize generalization of interfaces. Compared with the inherent API, the functional interface makes the data transmission process more reliable. Next, taking PCIe as an example for detailed explanation.

Typical API functions and corresponding callback routines in the kernel driver of the PCIe bus are shown in table 1.

Table 1. Typical APIs and callback routines of PCIe.

API function	KMDF callback routine
CreateFile	EvtDeviceCreate
ReadFile	EvtIoRead
WriteFile	EvtIoWrite
DeviceIoControl	EvtDeviceIoControl
CloseHandle	EvtFileCleanup EvtFileClose

By calling API functions and executing the corresponding routines, the basic communication functions between the host computer and the hardware device following PCIe protocol can be realized[7]. For the data transmission process, this layer firstly solves the selection of bus working mode. Taking PCIe as an example, the design of working mode judgement is shown in figure 4.

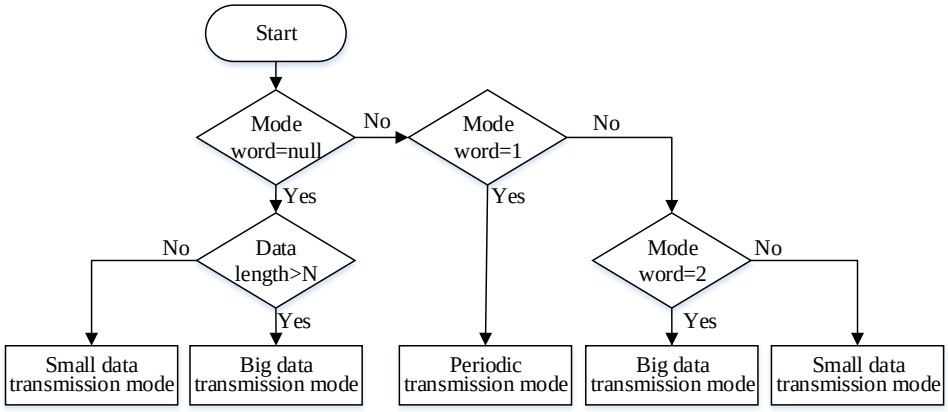

Figure 4. selection of the data transmission mode.

Next, according to the user's settings, two types of working modes, periodic transmission and single transmission, will be introduced in detail. In the periodic transmission process, the operation timing diagram is used to explain the judgment and processing of the periodic transmission routine. In the single transmission, PCIe packet transmission mechanism, priority mechanism and fault tolerance mechanism are introduced in detail through timing diagrams of reading and writing processing.

3.2.1. Periodic transmission. Because the periodic transmission can be regarded as resending data multiple times in a fixed cycle. it is necessary to judge whether the data is periodically sent before processing the data transmission tasks. According to the user's settings, the working timing of the periodic transmission is as shown in figure 5.

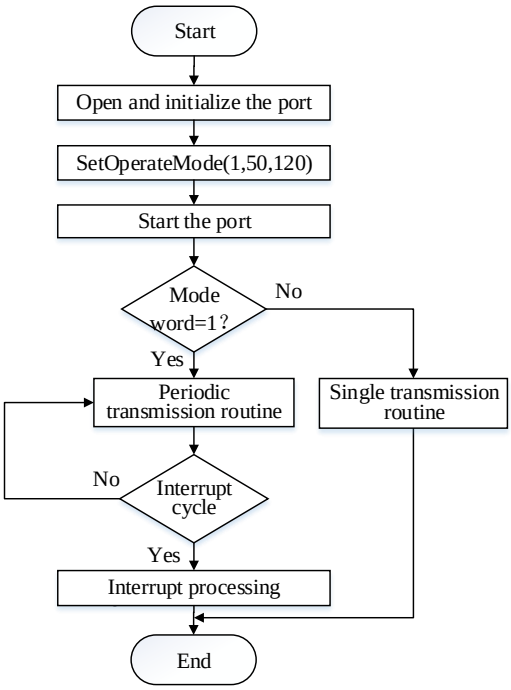

Figure 5. Working timing of periodic transmission.

According to the user's setting SetOperateMode (1, 5, 120), the flag word is 1, which means the data is sent periodically. In the periodic transmission routine, one timer is initialized according to the interface period set by the user for 5s, and the data is transmitted every 5 seconds. The interrupt timer is initialized simultaneously in the routine. After 120s, the transmission is aborted and the corresponding routine of interrupt processing is executed.

3.2.2. Single transmission. A round of single data transmission includes the most basic reading and writing functions required by the detection system. According to the user's demand for the amount of the transmitted data, this design adopts two working modes: DMA transmission for big data[8] and register transmission for small data.

In order to avoid long-term occupation of the bus caused by data transmission, a packet transmission mechanism is introduced, which is convenient for other operations in the packet transmission gap. In order to ensure the controllability and correctness in the data transmission process, the custom transmission protocol is designed to perform fixed encapsulation processing on the transmitted data in the form of a structure. The data packet protocol is shown in table 2.

Table 2. Data packet protocol.

Type	Byte number	Remark
Frame head	2	Fixed value
Target, source	1	The upper four bits are the target, the lower four bits are the source; 1 is the PC, and 2 is the hardware device.
Way ID	1	1:Event 2:Packet head 3:Sub-packet transmission 4:Packet cancel 5:Sub-packet retransmission 6:Small data transmission

Type	Byte number	Remark
Event ID	1	User-defined event content
Frame ID	2	Total packet number in the header packet transmission;Sub-packet number in the sub-packet transmission
Data total length	4	Special for packet head
Data section	10240	
Checksum	4	

As shown in the data packet protocol, the data portion occupies 10 KB of space, which is intended to select the data transmission mode automatically according to the data size. When there is no working mode setting, if the data size is less than 10KB, IODeviceControl direct transmission mode is adopted, and if it is larger than 10KB, DMA transmission mode is adopted. The DMA packetization method uses a header packet plus sub-packets. Each of packets has fixed length, and the last one is filled with 0 if it is not full. The sender is the process of unpacking and sending data, while the receiver is the process of receiving and combining packages. Following the data packet protocol, the writing timing is shown in figure 6.

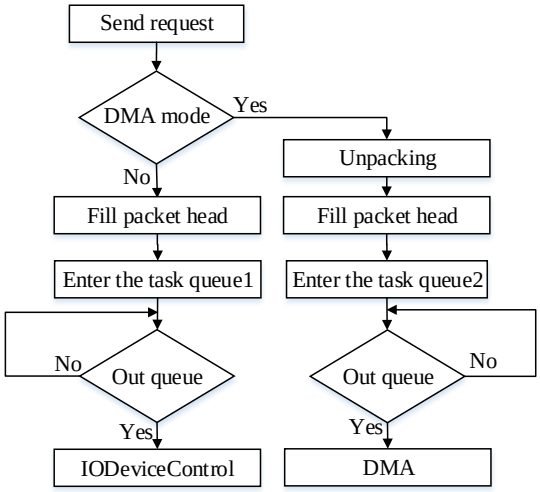

Figure 6. Working timing of writing process.

Unlike sending data, receiving data has no periodic concept and it is a passive behavior for the host computer. When the hardware device sets the interruption signal of transmitting data, the corresponding event is triggered in the kernel program, that is, WaitInterruptionEvent() in the base class. This interface listens to the event triggers of all buses. Once the event trigger is detected, it will enter the message response function of specific bus according to the trigger message, that is, the overloaded Read() in the corresponding subclass, thereby realizing the generalization of reading process. The reading timing under this design is shown in figure 7.

It can be seen from the reading timing that the fault tolerance mechanism needs to be set for receiving data. In this design, if the verification fails, data will be retransmitted automatically. If the number of retransmissions is more than three, the cause of error will be recorded for later investigation and the receiving process will be cancelled. To continue the data transmission process, sender must resend actively.

In addition, in order to ensure the real-time performance in the data transmission process, considering the dual influencing factors of waiting time and task time[9], this design sets the task priority mechanism - double queue mechanism. The data packets of big data and small data are

respectively stored in the two queues, the corresponding number of data packets in the two queues are alternately popped in a fixed number ratio (e.g. large: small = 5:1). After one queue pops up packets, it switches to the other queue. If the number of packets in the queue is less than the specified number, all packets in it are popped up. If the number is zero, the other queue is switched. This kind of priority mechanism, which is mainly based on transmission of big data packets and supplemented by small data packets, has a simple algorithm and can also solve the problem that the time of big data transmission is so long that the system cannot respond in time. At the same time, it can avoid the situation that the big data transmission waits too long due to the accumulation of small data packets.

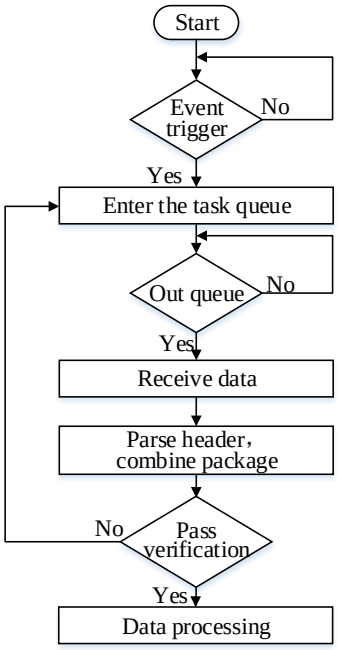

Figure 7. Working timing of reading process.

By introducing the design of periodic transmission and single transmission, this section describes design ideas and implementation methods of the interface implementation layer completely. The functional interfaces designed in this layer are secondarily encapsulated by the interface definition layer to implement general purpose interfaces that meet the design requirements.

4. Analysis of experimental result

In order to realize the communication bridge with reliability, functionality and versatility, this design proposes a universal interface model. To verify the feasibility of the scheme, this paper uses the PCIe interfaces applying to this model as an example to test. The test contents include reading and writing under the register transmission mode and the DMA transmission mode, and periodic transmission between the host computer software and DSP. The experimental results show that the error rate of data transmission process is low and the communication function of PCIe bus is implemented reliably.

Through multiple tests on the previous designed interfaces, this module is well adapted to the communication needs of different data sizes and multiple transmission modes. The universal module works stably and has simplified, intuitive and unified user-oriented interfaces. Its feasibility is demonstrated by the illustration of PCIe.

5. Conclusion

The universal interface model designed in this paper is intended to provide users with a simple, intuitive and easy-to-use universal interface module. By modelling, designing and packaging the interfaces, the program is deeply decoupled, which improves its portability and scalability.

Although it is inevitable to complicate the design and sacrifice certain performance while obtaining flexibility and scalability of the interfaces, it is acceptable for most application environment of the detection systems. What's more, the generalized and modular design of the detection system, as well as the scalability and portability of the application software, are greatly improved.

Reference

[1] Song LZ, Zhou HC and XU YP, The design and development of the test system for numerical control equipment interfaces, Modular Machine Tool & Automatic Manufacturing Technique, pp 26-9, 20 Dec. 2012.
[2] Wood.B, Backplane 101: RapidIO, PCIe, Ethernet, *World of Computer Automation*, pp 61-9, 2010.
[3] Yu W, Zhang Y, Xu H, Huang J, Gan C and Lu W, High Performance PCIe Interface for the TPCM Based on Linux Platform,*8th Int. Symp. on Computational Intelligence and Design* (ISCID 2015),11 May. 2016, Vol.2, pp 422-425.
[4] Lam H, Kirchgessner R, George A D, Reconfigurable Computing Middleware for Application, Portability and Productivity,*24th Int. Conf. on Application-Specific Systems, Architectures and Processors*, 5-7 June.2013, Washington, DC, USA, pp 211-18.
[5] Yan H and Zhang YR, Application of design pattern to communication interface design, *Computer Systems & Applications*, vol.5, pp 172-5,2012.
[6] Orzen S N, Interaction Understanding in the OSI Model Functionality of Networks with Case Studies,*9th Int. Symp. on Applied Computational Intelligence and Informatics* (SACI),15-17 May.2014, Timisoara, Romania, pp 327-30.
[7] Meng S and Lu J, Design of a PCIe Interface Card Control Software Based on WDF, *5th Int. Conf. on Instrumentation and Measurement, Computer, Communication and Control* (IMCCC 2015), 11 February.2016, pp 767-70.
[8] Hou HC, Wang YW and Li H, A high-speed DMA transmission system based on PCI express bus, *Microelectronics*, vol.43, pp 383-6, 20 June.2013.
[9] Zhao CY, Yan CX and Yu P, Real time scheduling of messages on 1553B bus, *Optics & Precision Engineering*, vol.18, pp 732-40, March.2010.