國立交通大學
電機與控制工程學系
碩士論文

MOST(Media Oriented System Transport)網路之探討與多媒體應用

Exploration of MOST (Media Oriented System Transport) Network with Multimedia Applications

研究生：吳建勳

指導教授：王啓旭

中華民國九十五年十一月

MOST(Media Oriented System Transport)網路之探討與多媒體應用

Exploration of MOST (Media Oriented System Transport) Network with Multimedia Applications

研究生：吳建勳　　Student：Jian-Xun Wu
指導教授：王啓旭 教授　　Advisor：Chi-Hsu Wang

國立交通大學
電機與控制工程學系

碩士論文

A Thesis

Submitted to Department of Electrical and Control Engineering College of Electrical Engineering and Computer Science

National Chiao Tung University

In Partial Fulfillment of the Requirements

For the Degree of Master

in

Electrical and Control Engineering

November 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年十一月

MOST(Media Oriented System Transport)網路的探討與多媒體應用

研究生：吳建勳　　指導老師：王啓旭 教授

國立交通大學電機與控制工程研究所

摘要

本論文主要在探討多媒體導向系統傳輸 (MOST, Media Oriented System Transportation)技術，並利用 OASIS 公司所提供的網路服務應用程式介面 (NetServices API) 撰寫程式去控制在光纖網路上的 DVD 播放器、收音機、放大器、以及光纖網路介面卡。爲了驗證對 MOST 通訊協定的了解，我們使用 OASIS 公司所提供的網路服務應用程式介面撰寫程式，將光纖網路上的 DVD 播放器上播放的 Mpeg2 串流資訊傳送到個人電腦端解碼後播出。此結果可馬上用在汽車的其他應用上，例如，結合 GPS 和車用麥克風系統，車用麥克風將使用者的語音經由辨識後，傳送至 GPS 定位系統，然後將 GPS 影像傳至車用螢幕，聲音傳至放大器。另外也可結合車用電話與音響系統，當使用者接聽電話時，將音響系統自動切換到靜音模式，讓使用者能更輕鬆的駕駛。這種結合各種車用電子裝置的多媒體應用正是 MOST 技術被制定的主要動機。

Exploration of MOST (Media Oriented System Transport) Network with Multimedia Applications

Student: Jian-Xun Wu **Advisor: Chi-Hsu Wang**

Department of Electrical and Control Engineering

National Chiao Tung University

ABSTRACT

In this thesis, a new MOST (Media Oriented System Transport) is explored. By using the program built in NetServices API (Application Program Interface) provided by Oasis, we can control the DVD player, the audio amplifier, the radio tuner, and the MOST network card under Microsoft Windows. The application of transporting Mpeg-2 stream from the DVD player on the MOST network to the PC under Windows is developed in this thesis. The final outcome is to decode the Mpeg-2 stream and play DVD movie on the PC Windows environment. This application can be immediately applied to other similar automotive applications. For example, a GPS navigation system can work in conjunction with car microphone system for navigation by recognizing user's voice. Also, the car telephone needs to mute the stereo system when a call is requested or received. Based on the MOST technology, the car vendors can develop many multimedia networking applications in intelligent automobile.

ACKNOWLEDGEMENT

I am very grateful to my advisor, Prof. Chi-Hsu Wang, who often gives me a hand in my research. Besides, I would like to thank all the members in the ECL laboratory. They always encourage me so that I always have a good time during this period in NCTU. Finally I would like to thank my family who always stay beside me and support me at their best.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

Automobiles have evolved from having a simple radio with perhaps a cassette or CD player to having a variety of sophisticated information systems[1]. With improvement of car electronics, the car is not only simple vehicle for people but also a place for entertainment. The drivers can ease their drive with the help of car electronic devices like the GPS navigation system, the car telephone system and the night vision system and the anti-crack radar. Also the passengers may even enjoy a DVD film with stereo audio in a long journey instead sitting bored.

Car today may include Car telephone, DVD Players, Amplifiers, Radio Tuner, LCD monitor, GPS, PDA, Bluetooth…etc. These devices need to communicate and interact with each other and with a human user. For example, a GPS navigation system can work in conjunction with a security system to locate a stolen car. The car telephone needs to interact with the stereo system to mute it when a call is requested or received. More and more electronics devices will be added to the car. And control, transportation and communication between theses devices become more and more important issues.

The traditional vehicle bus technology like LIN Bus, CAN Bus, FlexRay[2] can not afford the high bandwidth request of real-time multimedia information. Compared with these technology, the bandwidth of LIN Bus is only 20kbps，the CAN bus is 1Mbps and the FlexRay can be 10 Mbps.

Due to these reasons, a new MOST (Media Oriented System Transportation) technology is proposed. MOST is a standard for building in-vehicle multimedia systems. The MOST multimedia networking technology was designed to enable the efficient transport of streaming information and control over one media in automobiles for infotainment and other products such as consumer electronics or communication devices outside of the car.

MOST opens the door for new ways of networked communication enabling devices to talk to each other using a high-speed connection over plastic optical fiber (POF) or other media. The transmission rate of MOST technology is 24.8Mbps by now and will lead to 50Mbps and 150Mbps. This high transmission rate can meet the high bandwidth requirement of real-time multimedia devices. The technology also allows for transparent transport of various protocols (e.g. TCP/IP) and is object-oriented. It accommodates a wide range of real-time channel for real-time data and packet sizes for non real-time data.

MOST also allows for remote operation, flow control and variable arbitration mechanisms. MOST defines the protocol, hardware, software, and system layers necessary to allow for transport of real-time data in a very efficient and low cost way by using a single and preferred medium. MOST supplies a middleware named MOST NetServices API for engineers to develop self-made application programs.

In this thesis, we will introduce MOST specification in chapter 2. The NetServices API will be individually explained in Chapter 3. In Chapter 4, application programs built in NetServices API that control the DVD Player, the Amplifiers, and the Radio Tuner by MOST PCI Board on the MOST network are showed. At last, an application of transporting Mpeg-2 stream from DVD player to PC is developed. And Mpeg-2

stream will be decoded and played on the PC monitor.

Chapter 2

MOST Specification

In this chapter, we will introduce MOST Specification. Also we will give the reader an overview of MOST System. MOST System often indicates a MOST network, which is a group of MOST Devices linked together by some way. In this network, each MOST Device connects with each other by physical interfaces (Plastic Optical Fibers). Data routing, channel allocation, remote control and clock synchronization in this network are all based on MOST System Services. MOST specifies MOST System Services which are needed to develop MOST Devices. The MOST Specification defines all of required parts to implement MOST System Services. The MOST Specification can be divided into three parts, which are application section, network section, and hardware section as shown in Fig2-1 [3]. MOST System Services will be implemented in theses three parts. And each section may be further divided into sub-sections.

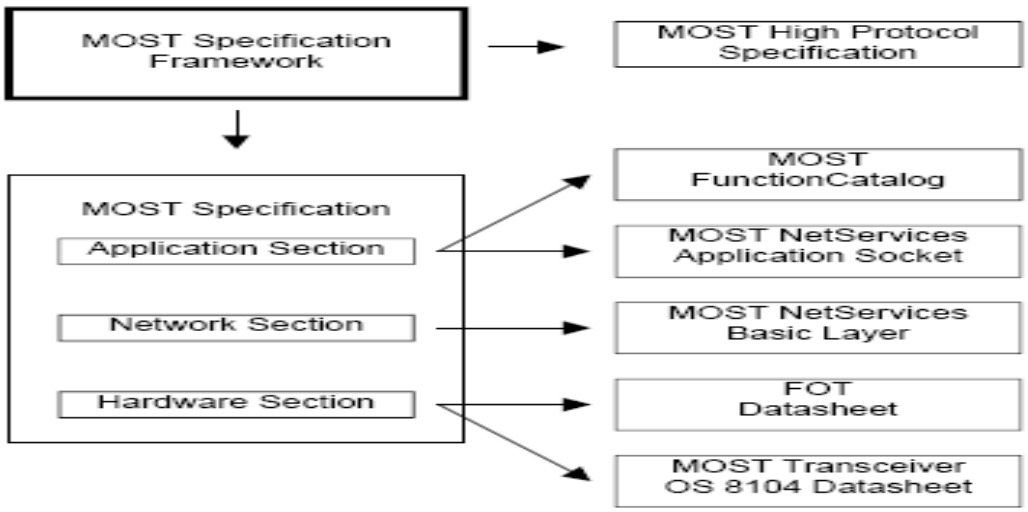

Fig 2-1 MOST Specification Framework

In this chapter, MOST System will be introduced in Section 2.1. MOST System Services will be introduced in section 2.2. MOST Devices will be introduced in section 2.3. The application section, network section and hardware section will be individually described in Section 2.4 ~ Section 2.6.

2.1 MOST System

In this section, we would like to give the reader a roughly understanding of MOST Specification without going deeply into it. The section includes some parts of Section 2.2~ Section 2.6. This section can be regarded as a tutorial of MOST Specification. A MOST System can be described by three definition areas which are MOST Interconnect, MOST System Services, and MOST Devices.

1. MOST Interconnect is the way of communication between MOST devices. During system start up, all devices get their unique device address. Each device has an address relative to its physical position with respect to the timing master. There is one timing master on a MOST network. MOST Frames are generated by the timing master. A synchronous bit stream interconnects the devices and provides synchronization to each node. A bi-phase (‘0’or‘1’) coding scheme with frame and block synchronization is applied to re-synchronize and decode data at each device node.
2. The MOST System Services are defined by the three layers, which are Low Level System Services, Basic Layer System Service, Application Socket. We will explain it in Section 2.2.
3. A MOST Device is a physical unit which can be connected to a MOST network via a MOST Transceiver. We will introduce MOST Devices in Section 2.3.

2.1.1 MOST Topology

MOST have high flexibility in network topologies. It allows combinations of Ring and Star topologies. It provides system designer freedom to choose the optimal architecture for any system.

2.1.1.1 Point to Point Link: Unidirectional or Bi-directional

The simplest way of interconnecting devices is one-way point to point link as shown in Fig 2-2. This approach can be used only if data transfer is required in a single direction. For example, this would be the case for connecting an audio source to a speaker. However, this basic configuration can easily be extended to more complex structures, such as branches or bi-directional links.

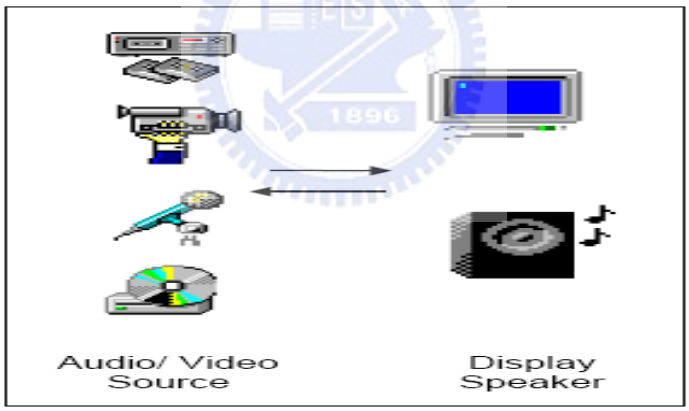

Fig 2-2 Point to Point Link

2.1.1.2 Ring Topology

MOST can run very efficiently on a ring topology as shown in Fig 2-3. There are many of advantages under this topology. First, low cost and constant cost per node since there is no overhead cost in the form of a hub, switch, or other shared network resource. Second, minimum use of physical layer media (e.g. POF), it reduces system cost and weight. Third, ease of expansion since no change in the basic architecture

(such as the wiring infrastructure) is required. Fourth, all source data (e.g. digitized audio) are available at each device.

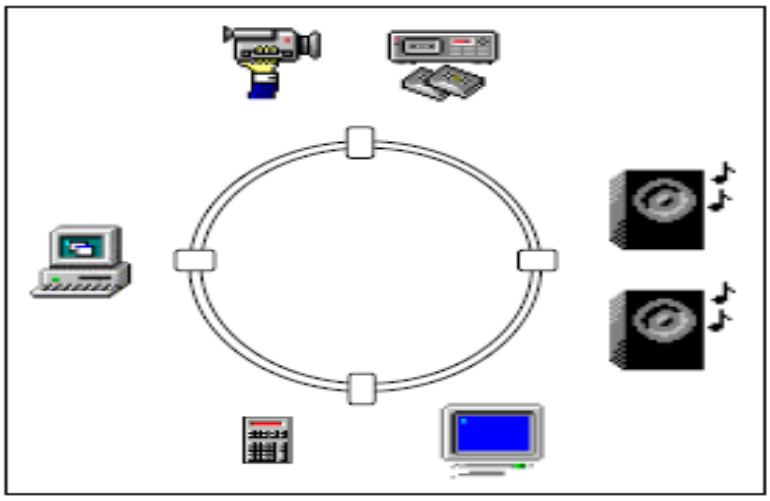

Fig 2-3 Ring Topology

2.1.1.3 Rings Incorporating Splitters

One-way point-to-point connections can also be incorporated in a Ring by incorporating passive or active optical splitters in a plastic optical fiber network, or simple branches in a network with a copper physical layer. Fig 2-4 shows this topology. This would typically be applied in connecting to display and/or active speaker devices that do not have to communicate back to the entire network. One major advantage of this implementation is reduced cost, since no transmitter is required. Active speakers are the most favorable candidates for this application.

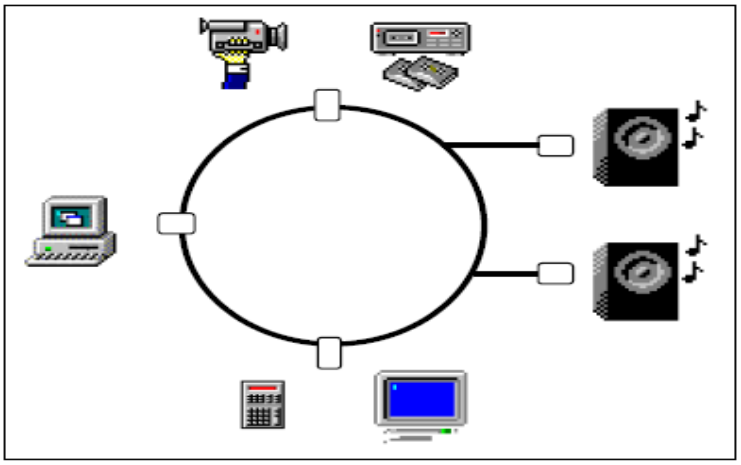

Fig 2-4 Rings Incorporating Splitters

2.1.1.4 Star Topology

Another option is a combination of Star and Ring topologies which may be the best choice for large networks as Fig 2-5 shows. In structures as required for home networking such a configuration with a central server/router (i.e. set-top boxes) can have multiple branches, with each of them configured as a Ring structure. This offers the most flexible way of implementing complex structures with easy fault detection and high bandwidth.

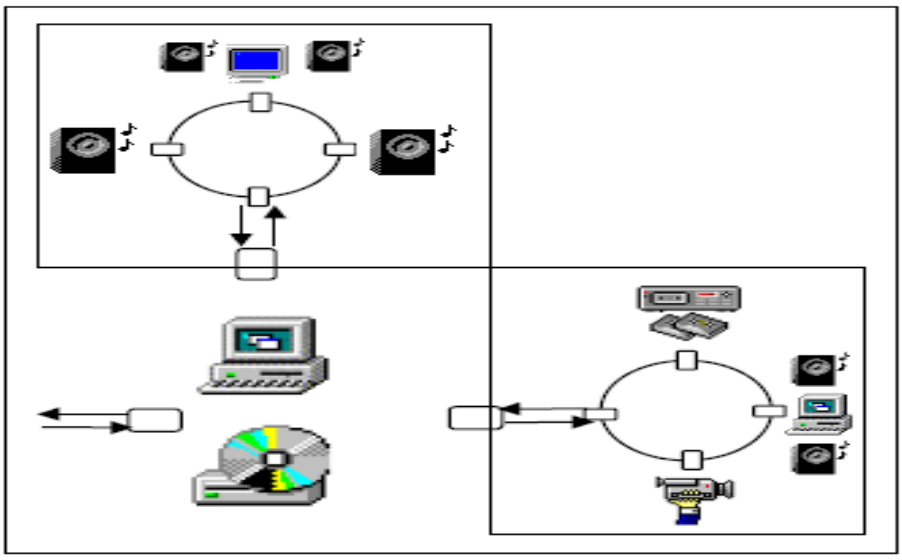

Fig 2-5 Star Topology

2.1.2 MOST Data Type

The MOST system supports a variety of data types such as control data, packet data and synchronous stream data.

1. Control data transfer is for device specific transfers and system management. For example, a remote control sends “Eject CD” control data to a CD player to eject a CD as shown in Fig 2-6. This kind of data is transported in parallel to the real - time and asynchronous data.

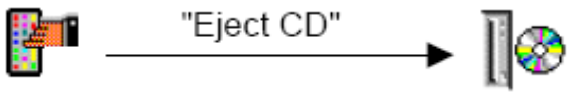

Fig 2-6 Control data

2. Bulk data / burst data transfer in asynchronous packets with variable bandwidth requirement is shown in Fig 2-7. It is often used for PDA or network TCP/IP packets.

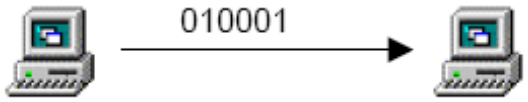

Fig 2-7 asynchronous packet data

3. Real-time data transfer, which requires a guaranteed bandwidth to maintain data quality, is shown in Fig 2-8. The microphone or CD audio/video data is such kind of data. If real-time data is sent to the MOST network, all nodes have access to it. The number of nodes listening to real-time data is only limited by the maximum number of nodes.

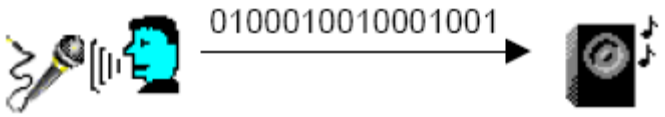

Fig 2-8 synchronous stream data

2.1.3 MOST Frame Structure [4]

Each block consists of 16 Frames running at the System Sample Rate as Frame Rate (typical 44.1kHz). The block boundaries are only relevant for Control Frame handling and network Management.

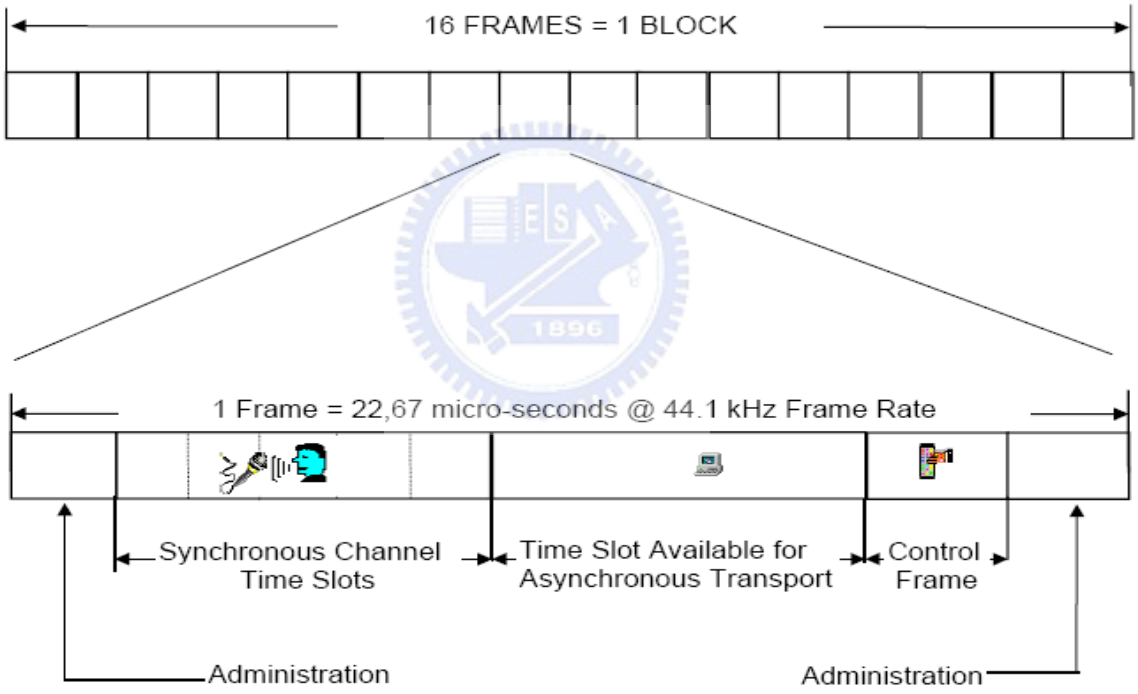

Fig 2-9 MOST Block and Frame Structure

The MOST Frame consists of 5 sections. There are sections that contain administrative data for synchronization and data security. 60 bytes are reserved for transporting synchronous source data and packet data, 2 bytes are available for control messaging. For dividing up the 60 bytes between synchronous source data and asynchronous data, there is a boundary descriptor value which is transported in the

administrative sections. The smallest portion accessible for the user is one byte also referred to as a channel in case of synchronous data. The MOST frame structure is shown in Table 2-1.

byte	Bit	Task
0	0-3	Preamble
0	4-7	boundary descriptor(synchronous area count value)
1	8-15	data byte 0
2	16-23	data byte 1
: :	: :	: :
60	480-487	data byte 59
61	488-495	control frame byte 0
62	496-503	control frame byte 1
63	504-510	frame control and status bits
63	511	parity bit

Table 2-1 Structure of MOST frame

1. Preamble

The preamble bits are used for synchronization of MOST nodes. For Slave nodes, the reception of the preamble bits indicates that the nodes are in-phase and in-frequency to the bit stream generated by the timing master (Master node). For the Master node, the received bit streams are phase shifted with respect to the transmitted bit stream. The delays are accumulated by each active node. The Master node will re-synchronize the incoming bit streams with the PLL (Phase-Lock-Loop).

2. Boundary Descriptor

The value is used to changing the bandwidth for synchronous data and asynchronous data transmission in each frame. The unit of the boundary descriptor value is 4 bytes or a quadlet (in MOST, 4-bytes is called a quadlet). It represents the number of quadlets used for synchronous data. There is some restriction in the boundary descriptor value. It can be a count value between 6 and 15 because the maximum number of bytes for asynchronous data per frame is 36 bytes (or 9 quadlets). The boundary descriptor is managed by the Timing Master of a MOST network. All synchronous connection must be re-built after having changed bSBC.

2.1.4 MOST Data Channels

2.1.4.1 Control Channel

On the control channel, data packets are transported to certain address, as they are on the asynchronous (packet) channel. Both channels are secured by CRC (Cyclical Redundancy Check). Each Control message transports 17 bytes of user data. The control channel also has an ACK/NAK mechanism with automatic retry. There are two kinds of control messages. Normal messages provide control of applications, while system messages handle system-related operations such as resource handling or remote access. During remote access, additional handshaking guarantees reliable remote operation. A control message is 32 bytes long and has the following structure. The delay between two messages is specified in register bXTIM. The structure of a control data frame is shown in Table 2-2.

Byte	**Task**
0-3	Arbitration

4-5	Target address
6-7	Own address (Source address)
8-25	Data area + 1byte message
26-27	CRC
28-29	Transmission status
30-31	Reserved

Table 2-2 Structure of a control data frame

2.1.4.2 Synchronous Channel

The Synchronous Channel Time Slots are available for real-time data such as audio/ video or sensors and eliminate the need for additional buffering in analog-to-digital converters (and digital-to-analog converters) or in single speed CD devices for audio and video. Continuous data streams that demand high bandwidth are transported over the synchronous channels. There is Routing Engine (RE) used to route data to and from the appropriate sources or sinks within a node. The connections are administered dynamically via the control channel. Although synchronous connections can be built directly by source and sink nodes, it is recommended that available bandwidth be administered in a center manner, particularly in large networks.

2.1.4.3 Transparent Channel

Some applications, however, may provide asynchronous data interfaces such as RS 232 and as a consequence require transparent asynchronous data transfer capability. In this case the asynchronous data lines can be oversampled at the synchronous channels and routed over the network to any other source data port.

2.1.4.4 Asynchronous Channel

The asynchronous channel is mainly used for transmitting data with large block size and high demand for bandwidth in a burst-like manner (graphics, some picture formats and navigation maps). There are two types of asynchronous packets in the data link layer. One is for 48 bytes data and the other is for 1014 bytes data as shown in Table 2-3 and Table 2-4. It may take several frames to complete a message. The management of arbitration and channel allocation is provided by the chip. The CRC is automatic calculated by the hardware and can be indicated in a register at the end of each asynchronous data.

Byte	Task
0	Arbitration
1-2	Target address
3	Length (in Quadlets = 4 bytes)
4-5	Own address (Source address)
6-53	Data area
54-57	CRC

Table 2-3 Structure of a frame in the asynchronous area (48 bytes data link layer)

Byte	Task
0	Arbitration
1-2	Target address
3	Length (in Quadlets = 4 bytes)
4-5	Own address (Source address)
6-1019	Data area

1020-1023	CRC

Table 2-4 Structure of a frame in the asynchronous area (alternative data link layer)

2.2 System Services

The MOST system services provide all the basic functionality to operate a MOST system. They are consisted of four parts which are application socket, basic layer system services, low level system services, and stream services as shown in Fig 2-10.

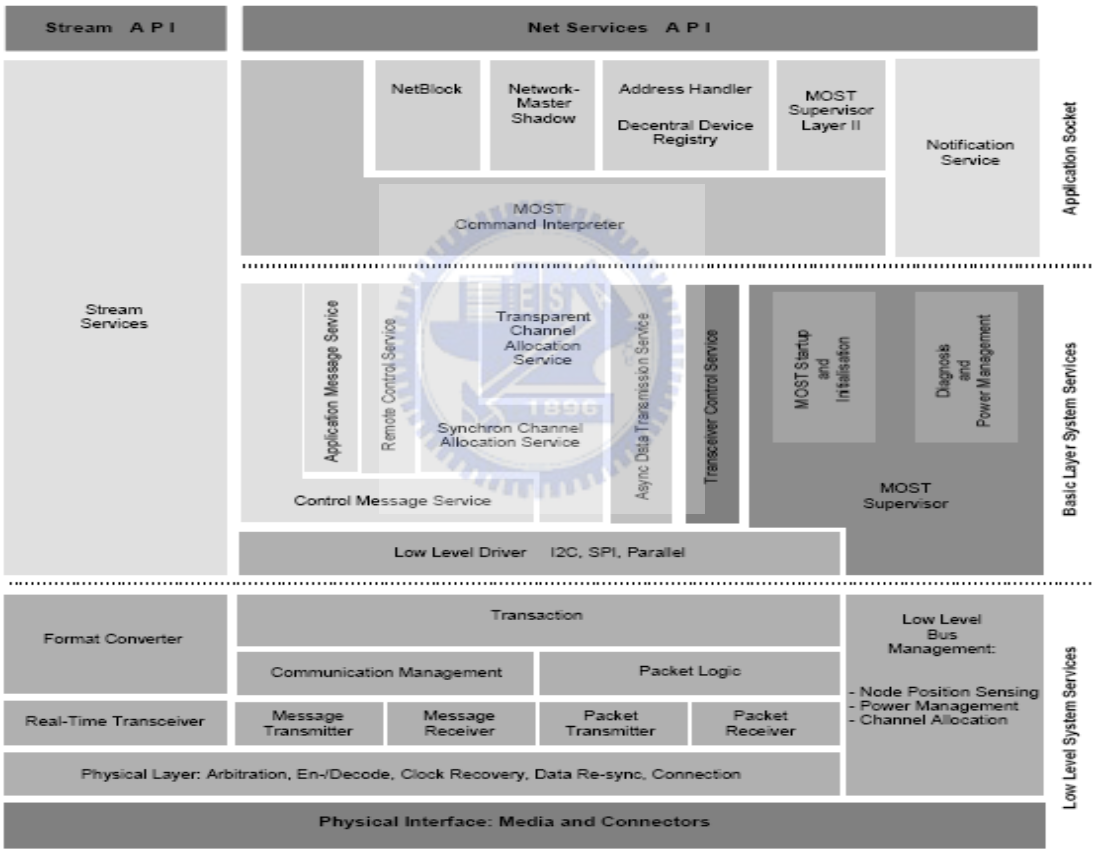

Fig2-10 Block diagram of the MOST System Services

The low level system services, except for the physical interface, are implemented in the MOST Transceiver. Stream Services, basic layer system services, and application socket are implemented in the NetServices.

2.2.1 Application Socket [5]

The Application Socket offers a wide variety of functions for building an application. Some of the functions are mandatory and some depend on the kind of application and are therefore optional. The Application Socket together with the Basic Layer System Services form a “basic device“, that contains all basic functionality a MOST Device needs. The Application Socket contains “MOST Command Interpreter”, “NetBlock”, “Network Master Shadow”, “Address Handler/ De-Central Device Registry”, “MOST Supervisor Layer II”, and “Notification Service”.

2.2.1.1 MOST Command Interpreter

The MOST Command Interpreter is based on the Application Message Service of the Basic Layer. It interprets and distributes received messages to the different function blocks of a MOST Device.

2.2.1.2 NetBlock

The NetBlock is a function block which is mandatory for every MOST Device. It contains basic properties of the entire device (e.g. Addresses and available function blocks).

2.2.1.3 Network Master Shadow

The Network Master is a central instance (on highest level) in a MOST network, which handles administrative tasks like the “Central Registry“. Network Master Shadow is a local image of the Network Master which is needed for being able to receive messages of the Network Master. It is optional, since these functions can be handled in a de-central manner too.

2.2.1.4 Address Handler, De-Central Device Registry

If the actual logical MOST address of a function block is unknown, this optional service seeks it anywhere in the MOST network. It is possible to keep a local device registry, which then contains the respective logical addresses.

2.2.1.5 MOST Supervisor Layer II

MOST Supervisor Layer II provides initialization of a device on highest level, e.g. of the logical address.

2.2.1.6 Notification Service

This service organizes the sending of notification messages, which are sent if a property is changed.

2.2.2 Basic Layer System Services

The Basic Layer of the MOST NetServices provides comfortable access on the Low Level System Services. The Basic Layer System Services consist of “MOST Supervisor”, ”Low Level Driver”, “Control Message Service”, “Synchronous Channel Allocation Service” , “Transparent Channel Allocation Service”, “Asynchronous Data Transmission Service”, “Transceiver Control Service”.

2.2.2.1 MOST Supervisor

This service offers transceiver initialization and network startup on a higher level. Failure diagnosis and power management are also supported by MOST Supervisor.

2.2.2.2 Low Level Driver

The Low Level Driver provides an interface between microcontroller area and the

MOST transceiver. It therefore offers different interface formats. For adapting to the respective micro controller (μC) environment, the Low Level Driver is complemented by user definable hardware specific functions.

2.2.2.3 Control Message Service

The Control message Service buffers normal control or system messages when sending and receiving. Error detection and notification is provided too.

2.2.2.4 Synchronous Channel Allocation Service

The Synchronous Channel Allocation Service is based upon embedded Channel Allocation which is part of the Low Level Bus management. It allocates or de-allocates resources for real time data streaming on the MOST network and "routes" this data to the desired destination.

2.2.2.5 Transparent Channel Allocation Service

This service provides allocating and de-allocating as well as routing of transparent data.

2.2.2.6 Asynchronous Data Transmission Service

Sending and receiving of asynchronous data is the task of this service. Errors are detected and notified.

2.2.2.7 Transceiver Control Service

This service provides access to configuration and address registers of the MOST transceiver.

2.2.3 Low Level System Service

The Low Level System Services consist of the “Physical Interface”, “Physical Layer”, “Low Level Bus Management”, “Packet Logic”, “Communication Management”, “Transaction Level”, “Real Time Transceiver”, “Format Converter”.

2.2.3.1 Physical Interface

The physical interface provides mechanical connection between the Fiber Optic Receiver/ Transmitter (FOT) units and the Plastic Optic Fiber (POF). In addition to that, it connects the FOT units electrically with the MOST transceiver. The Physical Layer is optimized for the use of POF as the transfer medium, but copper cable (e.g. UTP, Coax) is permissible too.

2.2.3.2 Physical Layer

The physical layer as implemented in the MOST transceiver OS 8104, provides connection of the MOST devices. In addition to that, clock recovery, data de- and encoding as well as arbitration for asynchronous channels are implemented.

2.2.3.3 Low Level Bus Management

The Low Level Bus Management provides a set of functions that handles power management as well as Channel Allocation for real - time data, and Node Position Sensing. In addition to that, it provides information for delay compensation. A message to a particular device, a group of devices, or all devices as a result of a status change of the INT pin is available in standalone mode too.

2.2.3.4 Packet Logic

Packet Logic controls sending and receiving of packet/ bulk data in Packet

Transmitter and Packet receiver. It includes error detection for packet data.

2.2.3.5 Communication Management

Communication management controls sending, receiving and error detection of MOST control messages (normal messages as well as system messages).

2.2.3.6 Transaction Level

Provides access to data received either as packet data or MOST control message, and the transmit areas for sending data (asynchronous and control).

2.2.3.7 Real Time Transceiver

The Real Time Transceiver receives real time data from the external world, or sends this kind of data to the external world. In addition to that, it performs „Routing“ on real time data, that means it puts the data to the right destination (network or application).

2.2.3.8 Format Converter

The format converter converts a variety of serial source data formats (S/PDIF, Sony, Matsushita, I2S,...) into a format that can easily be transported via MOST network, or vice versa.

2.2.4 Stream Services

The Stream Services provide transport services for real-time Source data. This allows to handle Source data in the respective parts of the application area.

2.3 MOST Devices

2.3.1 Logic Device Modal

From the logical point of view, we can separate MOST Device into four parts which are “Function Block”, “NetServices”, “MOST transceiver”, and “Physical interface” as shown in Fig2-11.

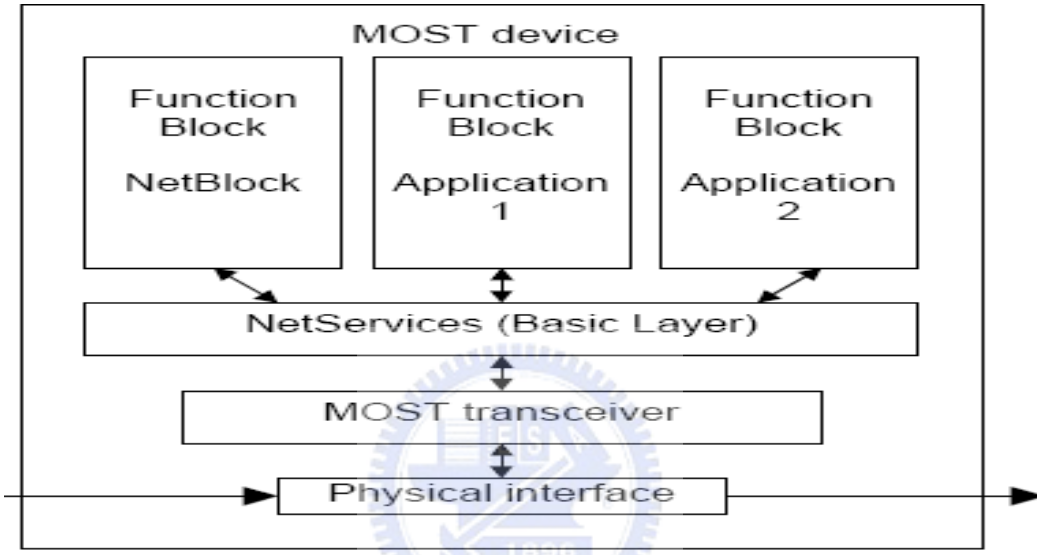

Fig 2-11 MOST Device Logic Model

A MOST device consists of several function blocks, an interface level providing a standard interface (NetServices, Basic Layer) for communication between the function blocks and a unit providing MOST functionality (e.g. a MOST transceiver chip) and the physical interface (e.g. POF) to the MOST network. The function blocks will be explored in section 2.4. Also MOST Transceiver will be explained in section 2.5. The NetServices of Basic Layer will be investigated in Chapter 3. The Physical Interface will be described here.

2.3.2 Physical Interface

MOST technology complies with all requirements for different physical layers and is

able to support complex topologies required for plastic optical fiber applications or infrared controlled systems. A detailed description of the proposed physical interface for MOST is provided in the MOST Physical Interface description. The physical interface consists of at least two signal lines which are RX (receive data line) and TX (transmit data line). Optional lines could be Power, Ground, Wake-Up which is an optional wake up line for waking the MOST network. The wake up function in general is derived from sensing activity on the RX line. A receiver device should be able to detect activity and wake up the target device. As an additional option, an electrical wake up line might also be implemented. A MOST device would wake up on activation of that line. The power supply can come from anywhere since common mode problems are not expected due to the use of optical transmission techniques.

2.4 Application Section

In this section, we will explain the Function Block and its functions in detail. As described in section 2.3.1, a MOST Device contains multiple components, which are called function blocks. Every function block may contain a set of functions. The function block identifier is represented as FBlockID

To make a function accessible from outside, the function block provides a function interface (FI), which represents the interface between the function in a function block and its usage in another function block as shown in Fig2-12.

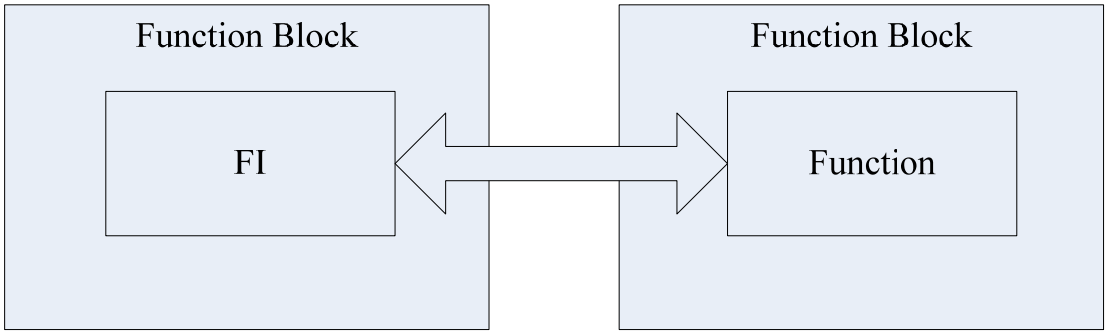

Fig 2-12 Function Block and Function Interface

2.4.1 Function Block Types [6]

There are three types of function blocks. Function Blocks which are always controlled are called slaves. Function Blocks which have an interface to the user are called Human Machine Interfaces (HMIs). Function Blocks using functions in other function blocks are called controllers. Controllers themselves may also be controlled.

2.4.2 Function Block and Function ID

On the application level, a function is addressed in independently of the device it is in. Functions are grouped together in function blocks with respect to their contents. To distinguish between the different function blocks (FBlocks) and functions(FKt) of a device, each function and function blocks has an identifier(ID). To specify a function in a device, we need to give the Function Block ID and Function ID like FBlockID.FktID.

2.4.3 Function

A Function is a defined property of a function block that can communicate with the external world through the boarders of its function block. Function can be subdivided into two classes which are “methods” and “properties”.

The class of function that can be started and which lead to a result after a definable period of time is called “methods”. The class of function for determining or changing the status of a device, which refer to the current properties of a device, is called “properties”. In addition to that, there are also “events”. Events results from properties if the properties are requested to report changes (Notification). The relationship between Function Block and classified functions is shown in Fig 2-13. When accessing functions, certain operations are applied to the respective property or method. The kind of operation is specified by the OPType. For example, ‘set’ or ‘get’ the frequency of the radio tuner. And the parameters of operation is followed by the OPType, resulting in the structure, FBlock.FktID.OPType(Data)

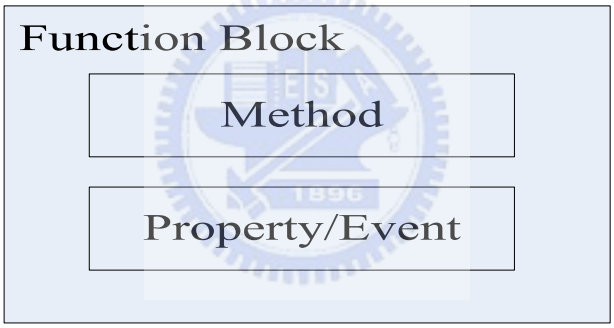

Fig 2-13 Class of Functions

2.4.3.1 Methods

Methods are used to control function blocks. It will start some process of controlled function blocks and return the results to the controlling function block after finishing the process. In general, a method is triggered only once, for example, allocating audio channels for a MOST DVD player or starting auto-scanning of a radio tuner. The results of process may be the allocated audio channel numbers for the DVD player or a found frequency for the radio tuner. It is possible to use parameters in a method, for example, auto-scanning upward or downward.

If a process runs for long time, it might be useful to return intermediate results before finishing, such as informing the controlling function block about the successful start of the process. For executing methods, the following kinds of messages are exchanged via the bus as shown in Table 2-5.

Controller	**Slave**
Start of a Method with Parameters (Start/Start Result)	Error with cause for error(Error) Execution report with results(Result) Intermediate result(Processing)

Table 2-5 Function of Method

2.4.3.2 Properties

Properties are used to change or get the state of the device. For example, change the status of the MOST DVD player from ready state to playing a DVD. For changing and reading of properties, the following types of messages are exchanged via the bus as shown in Table2-6.

Controller	**Slave**
Setting a property (Set/SetGet) Reading a property(Get) Increasing/decreasing a property	Status of property (Status) Error message with cause of error(Error)

Table 2-6 Function of Property

2.4.3.3 Events

Properties of a function block may change without an external influence, e.g. the

current time of a DVD player. In this situation, a cyclic reading of the properties (polling) will be required. This will add the overhead of communication. To avoid this, it would be useful if function blocks can send status report about changes in properties without explicit requests. These are events that occur in a controlled function block, which initiate the sending of report (notification). For example, if the play time of a CD player has changed, an event will occur and send the new play time to notify the controlling function blocks. The changed property can be sent to several devices, this will be administrated in the notification matrix. The devices that should be notified of changes to the status of a function are registered in this matrix. The notification matrix looks like Table 2-7. This matrix is only a model. It doesn't dictate the software implementation method.

Entry	Fkt1	Fkt2	Fkt3	Fkt4	Fkt5
DeviceID1	X	X	X	X	X
DeviceID2		X		X	
free for entry					
free for entry					

Table 2-7 Notification matrix (X = notification activated)

A notification matrix is implemented in every function block. The size of a notification matrix depends on the function block, on the number of properties, and on the number of device entries, each of which must be registered individually. The DeviceID has 16bits and a FktID has 12 bits. Group addresses are allowed as DeviceID in the notification matrix. The notification matrix is administrated via the function Notification. If a controller desires to register, or to remove registration, it

sends the protocol like

Controller->Slave:FBlockID.InstID.Notification.Set(control,DeviceID,FktID1,FktID2,…)

As described above, the InstID is used to distinguish the same type of function blocks in a MOST network. ‘Notification’ is FktID and ‘Set’ is OPType.

2.4.4 Function Interface

A function interface (FI) represents the interface between a function in a function block, and its usage in another function block. To communicate with a function, a controller or an HMI needs information about the available parameters, their limitation, and the allowed operations. Take the Fig2-14 for example. The FI contains information about the data type of the function and about minimum and maximum value. In real implementation, as FI contains much more information.

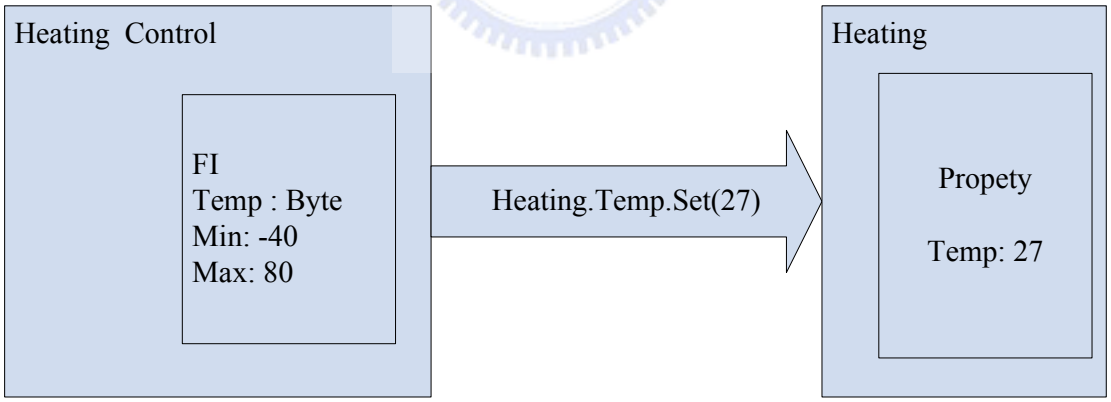

Fig 2-14 Example for a function interface (FI)

2.5 Network Section

In this section, we will introduce the features of the MOST Transceiver OS8104 [7].

2.5.1 Master/Slave

Basically, a MOST system consists of up to 64 nodes with identical MOST Transceivers. By configuration, any of the transceiver can be Timing Master, all the others are slaves. The Timing Master as well as active Slave devices (source data bypass is open, device can put source data on the bus) add two samples of delay to the path of source data. A passive slave device has a closed source data bypass. Since in that case the routing engine is inactive, no delay is generated.

2.5.2 Internal Services

2.5.2.1 Addressing

The MOST Transceiver supports four different ways of addressing which are node position in the ring, unique node address (2 bytes), group address (1 byte), and Broadcast. The node position is generated automatically in each node during the locking procedure of a MOST network. The unique node address can be set by the application. The MOST Transceiver supports a procedure on the chip level to verify whether an address is unique or not. This procedure is called SAI (Address Initialization). The group address can be set by the application. A group is made up of devices that have the same number in the group address register. The broadcast address is a special group address. When used, the message is receive by all nodes in the ring. Until the last node in the ring has acknowledgement a broadcast message, communication via the control channel is suppressed for other messages. The different ways of addressing are mapped into the address area of a MOST Transceiver as shown in Table 2-8.

Address range	Mode
0x0001..0x02FF 0x0500..0xFFFF	Normal addressing (Point to point) based on unique node address
0x0400..0x04FF	Node position addressing: Address = 0x400 + Node position of target node
0x0300..0x03C7 0c03C9..0x03FF	Group addressed: Address = 0x300 + Address of desired group Group address 0xC8 reserved for broadcast
0x03C8	Broadcast addressing

Table 2-8 Address modes VS address range

2.5.2.2 Power Management

There are three power states in the MOST Transceiver. They are normal operating mode, low power mode, and zero power mode.

Low power mode is used to lower the power consumption of devices that are not in use at the moment. All sections of the chip that handle source data are switched off, source data bypass is active. The chip is visible from the network's point of view, but no communication is possible. Based on the Timing Master, all components that are in low power mode can be awakened by a wakeup-preamble.

On "zero power" mode, all sections of the chip are deactivated except a small wakeup logic. The wakeup logic activates the receiving FOT unit by a special pin of the chip in regular intervals and scans for modulated light. If there is modulated light, the chip wakes up. When using an FOT unit with wakeup feature, the zero power mode of the chip does not need to be used. The chip can be connected to the switched branch of

the power supply.

2.5.3 Dynamic Behavior of Device

This section will describe the states and the state transitions of the system. The figure 2-15 below shows a layer model of a device. The lowest layer is the power supply. On this layer, every hardware function is built, that is, the hardware of the NetInterface, which is made up of the MOST Transceiver, the optical interface, and the controller on which the NetServices are running. The NetServices make up the next layer, on which the highest services of address management, power management and network error management are based. At the top layer there is the application itself.

Generally, for each device, the device specification must define all the possible combinations of the states of the application section and the communication section. Especially from the view of the network, there are three states that are mandatory for each device.

1. DevicePowerOff: Communication section is in state NetInterfacePowerOff. The application section in a non-waking device is in state ApplicationPowerOff, or in a waking device in states ApplicationSleep.
2. DeviceStandBy: This state is mainly influenced by state ApplicationLogicOnly. The logical function of the application is running, while peripherals with high power consumption such as drives are switched off. This state is reached after state DevicePowerOff. The communication section is in state NetInterfaceNormalOperation.
3. DeviceNormalOperation: The communication section as well as the application section are in state NormalOperation.

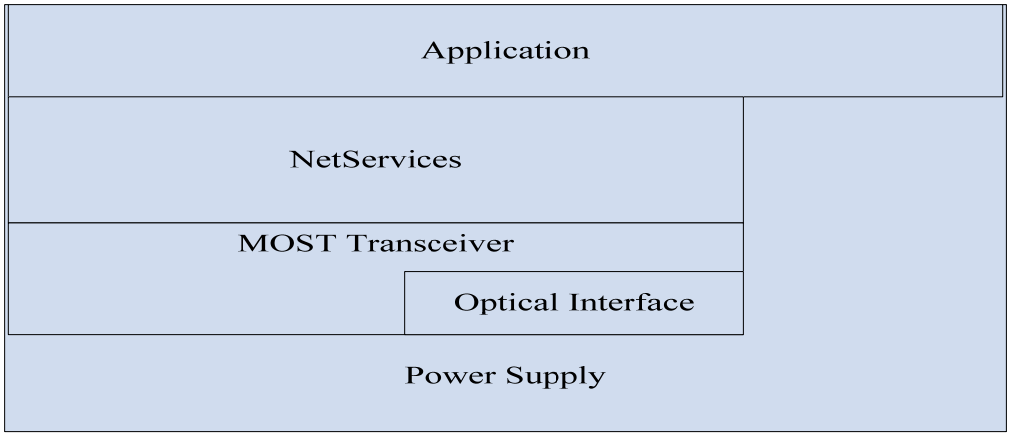

Fig 2-15 Layer model of a device

2.5.4 NetInterface

The states of a device are seen from the view of the NetInterface. The state diagram of the NetInterface is shown in Fig2-16.

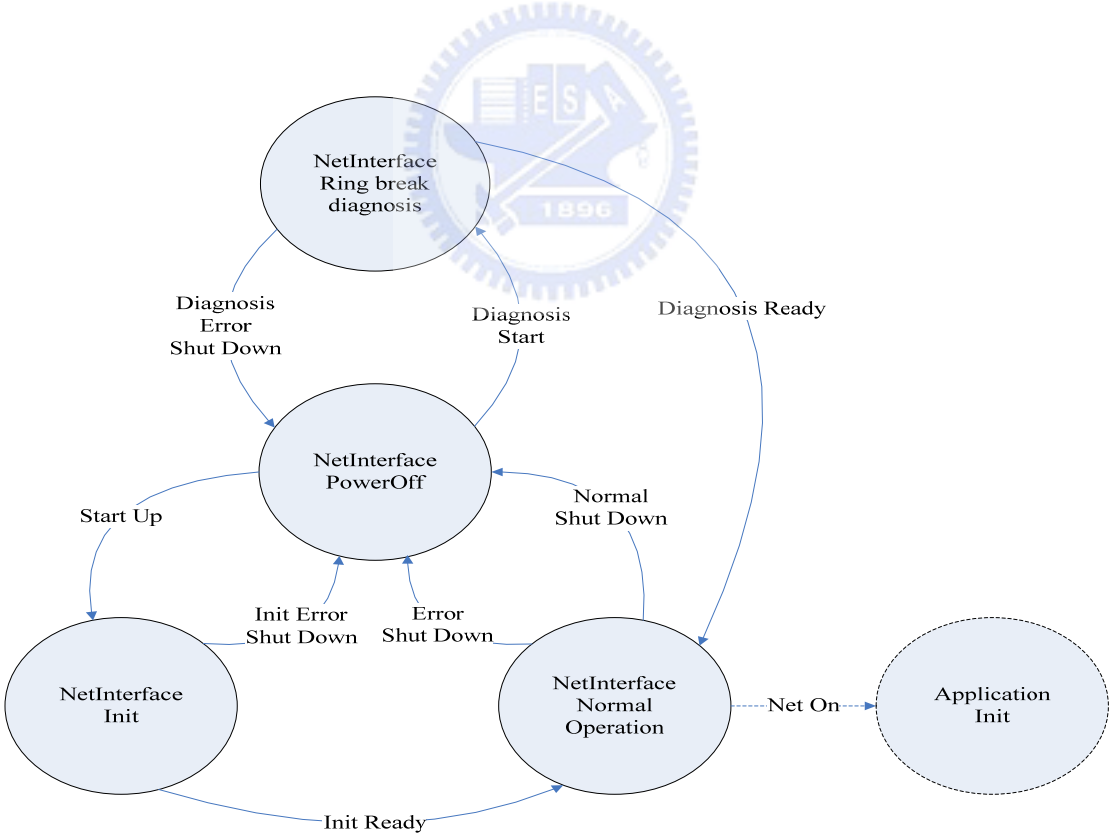

Fig 2-16 Flow Chart “Overview of the states in NetInterface”

2.6 Hardware Section

In this Section, we will introduce the hardware structure of a MOST Device. The hardware structure of a MOST Device is displayed in the block diagram as shown in Fig 2-17. A MOST Device consists of optical interface area, MOST function area, uC area, application area, power supply area. Not all of the areas are necessary. For example, an active speaker does not need a microcontroller. Areas that are not mandatory are displayed in gray.

The optical interface area consists of an optical receiver and optical transmitter. The MOST function area consists of MOST Transceiver, Crystal, PLL-Filter. The micro controller (uC) area mainly consists of the uC and some memory, and is not mandatory for a MOST Device (standalone mode). Application area refers to the application peripherals such as receiver, amplifiers, drive, etc. Power Supply Area describes the power supply for a device that is usually active when the network is active, so a low standby-current I_{STBY} must be achieved. For more information on the hardware structure of a MOST Device, please refer to MOST Framework **[3]**.

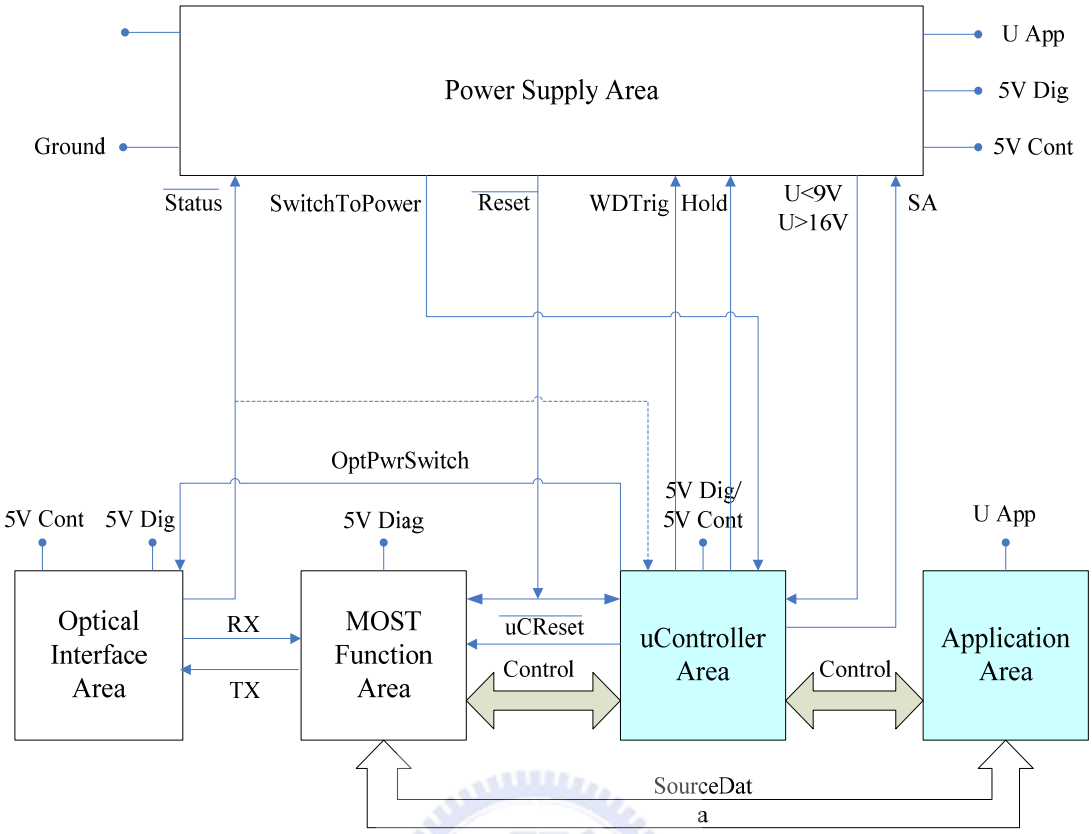

Fig 2-17 Structure of a MOST Device

Chapter 3

NetServices API

In this chapter, we will introduce MOST NetServices API. With NetServices API Basic Layer, we can construct our program to control the MOST DVD Player and the radio tuner and Amplifiers. The hierarchy of MOST NetServices is shown in Fig 3-1 [8].

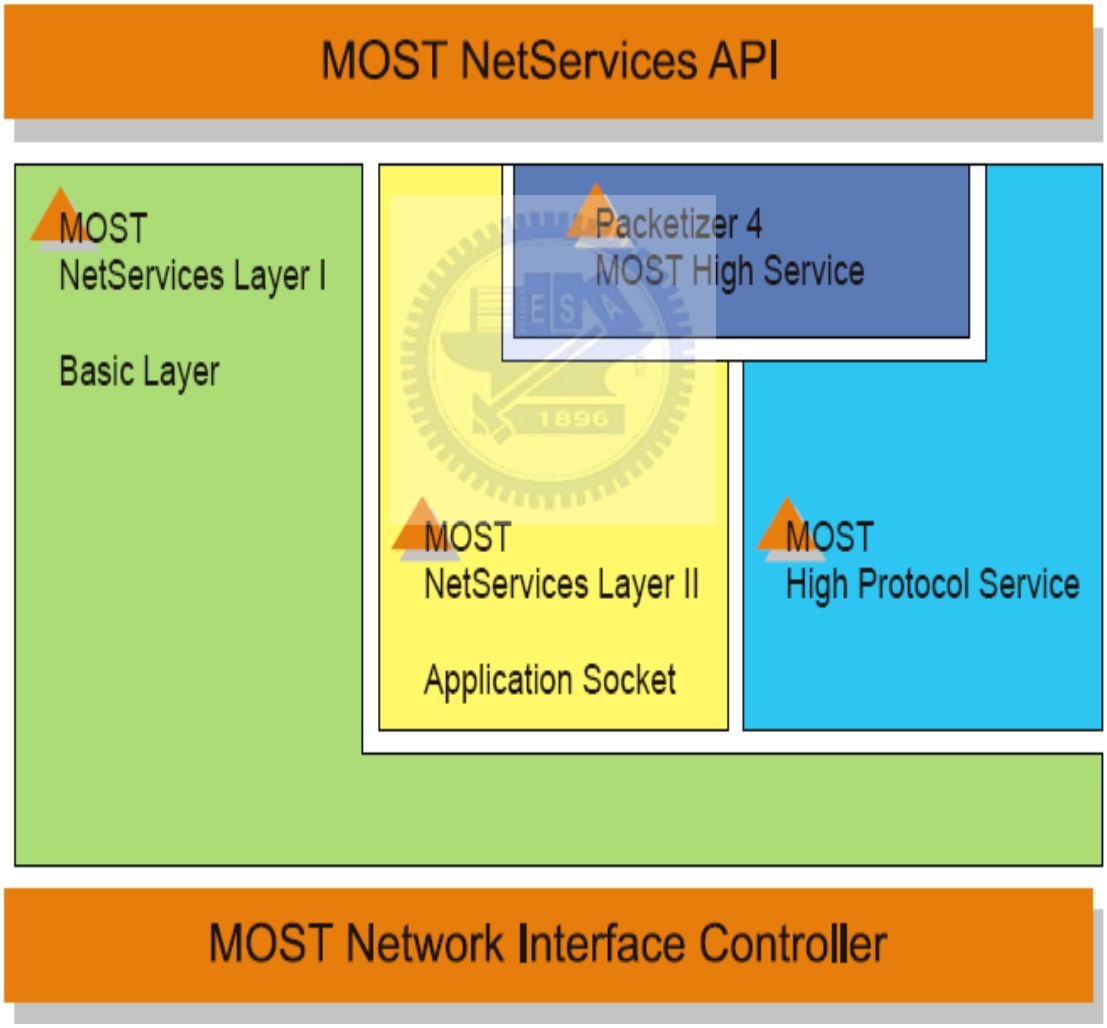

Fig 3-1 Hierarchy of MOST NetServices

Compared to the OSI Layer Model, the MOST Layer Model is shown in Fig 3-2. The

NetServices Basic Layer APIs are consists of MOST Supervisory and Startup Functions (MSV), Control Message Service (CMS), Application Message Service (AMS), Remote Control Service (RCS), Synchronous Channel Allocation Service (SCS), Transparent Channel Allocation (TCS), Asynchronous Data Transmission Service (ADS), MOST Transceiver Control Services (MSC) [9].

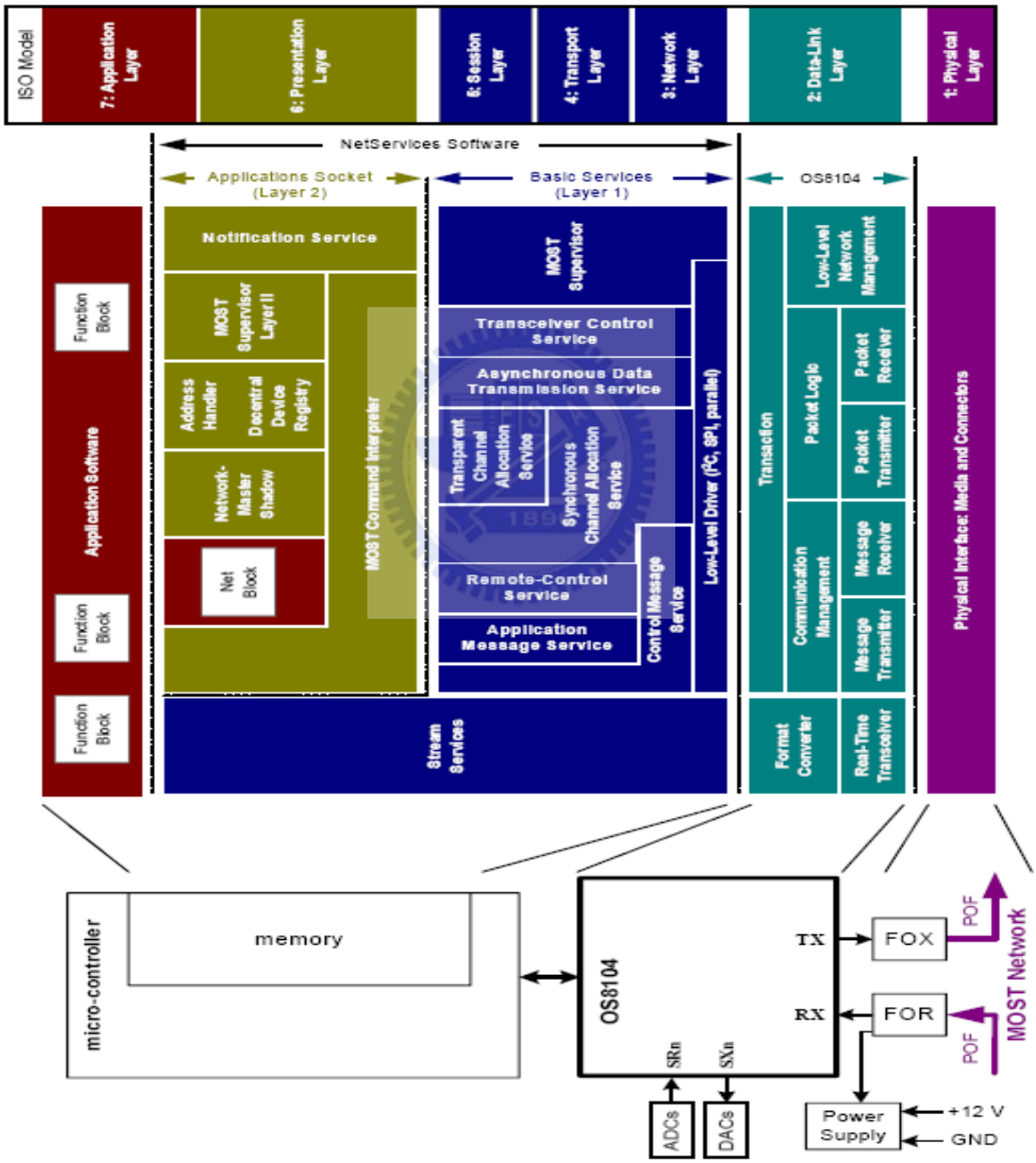

Fig 3-2 Hierarchy of MOST Layer Model

3.1 MSV (MOST Supervisor Service)

void MostStartUp(byte dev_mode, byte options): This function will initialize the State Machine of MOST Transceiver as shown in Fig 3-3.

“dev_mode” is used to set up the device mode of MOST Transceiver. The value and corresponding mode is shown in Table 3-1.

Mode	Value	Comment
SLAVE	0x00	The device will be configured in slave mode
MASTER	0x01	The device will be configured as timing master

Table 3-1 Values of the device mode

“options” is used to set the behavior of the state machine. The value and corresponding behavior is shown in Table 3-2.

Opt	Value	Comment
ZERO_REQUEST	0x10	If this option is set, MOST Transceiver will go to Zero-Power-Mode, as soon as the network is switched off
ZERO_WAKEUP	0x20	This option allows to “wake up” the MOST NetInterface if it is in Zero-Power-Mode. The MOST Transceiver needs not to be reset. The following registers stay unaltered: bNAH, bNAL, bGA, dXTIM, bXRTY, bAPAH, bAPAL. The other registers are reset to their default values.
FAKE_OFF	0x08	When setting this option, MOST NetInterface will go

		immediately to the Virtual-Zero-Power state. From the network's point of view, the MOST Transceiver will be in ZeroPower mode(not light will be emitted). In actual fact, it is up and running(Generating RMCK, local routing of source data…).
SLAVE_WAKEUP	0x04	This option allows to wake up the entire MOST network from a slave node. Valid from version 1.10.00: This option is ignored, when the network has been already started. Valid until version 1.09.xx: This option must only be used, when the network is off. This option must not be combined with option DIAGNOSIS
DIAGNOSIS	0x02	By this option, Ring break Diagnosis will be activated.
RESET	0x01	If this option is set, the MOST Transceiver will be set to RESET state, if the State Machine is initialized. It will be waited for a POR-INT.(Power-on-after-Reset-Interrupt)
(rest)	-	Reserved

Table 3-2 Values of options

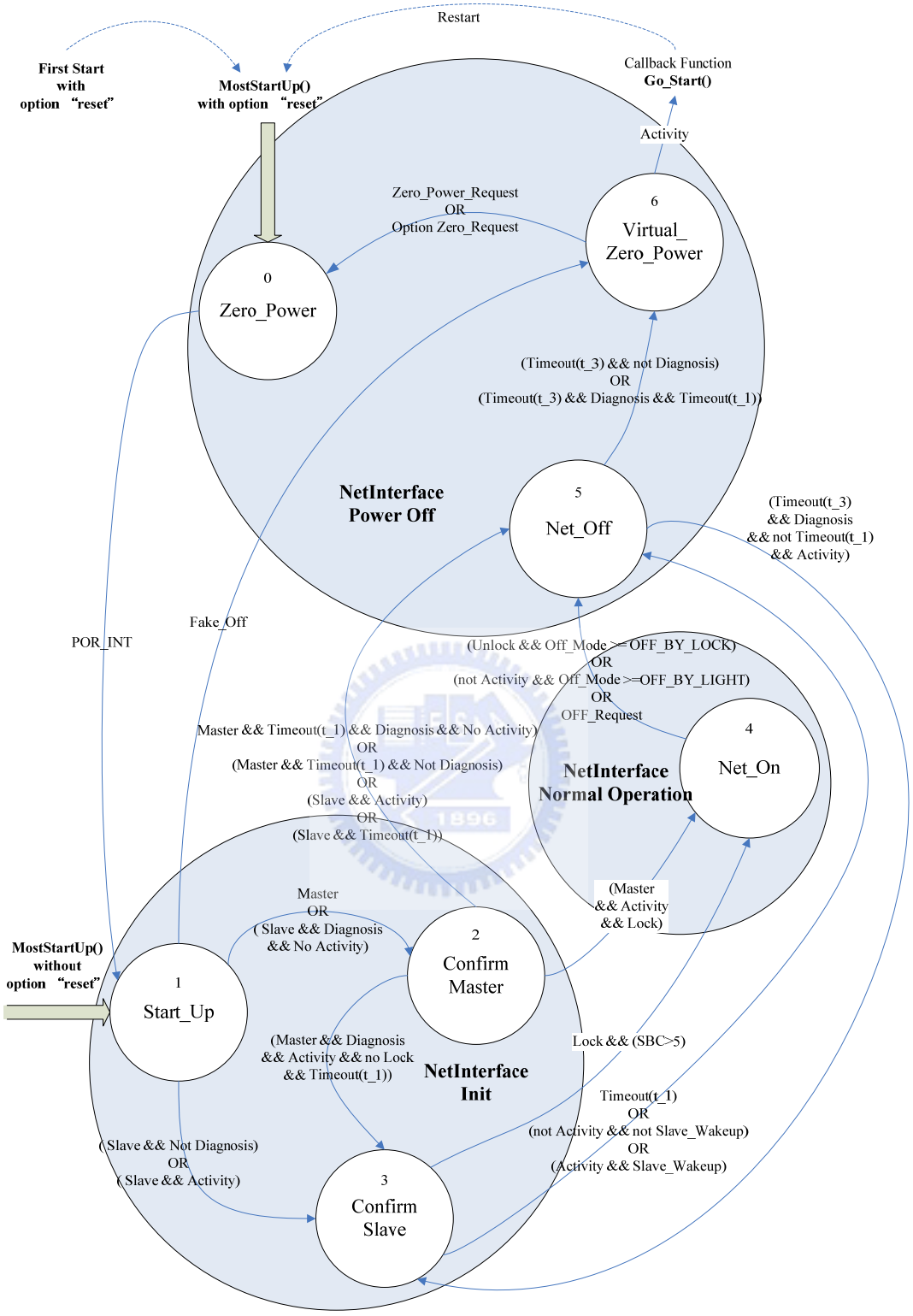

Fig 3-3 State flow of MOST Transceiver

void MostShutDown(): This function will switches off the light at the device's output. State machine will be in Power-Off then. MOST Transceiver automatically changes to

Zero-Power-Mode. If option Zero-Power-Mode, if option ZERO_REQUEST was set.

The following callback functions will return the state of the MOST Transceiver.

void Most_Reset (void): when the transceiver is reset

void Go_Confirm_S(void): when the state of transceiver goes to CONFIRM_S.

void Go_Confirm_M(void): when the state of transceiver goes to CONFIRM_M.

void Go_Net_On(void): when the state of transceiver goes to Net_ON.

void Go_Net_Off(void): when the state of transceiver goes to NET_OFF.

void Go_Virtual_Zero(void): when the state of transceiver goes to virtual_ZERO_POWER.

void Go_ZERO_POWER(void): when the state of transceiver goes to ZERO_POWER.

void Go_Start(void): when the state of transceiver goes to Start_Up.

void Store_Error_Info(byte error): indicate an error event.

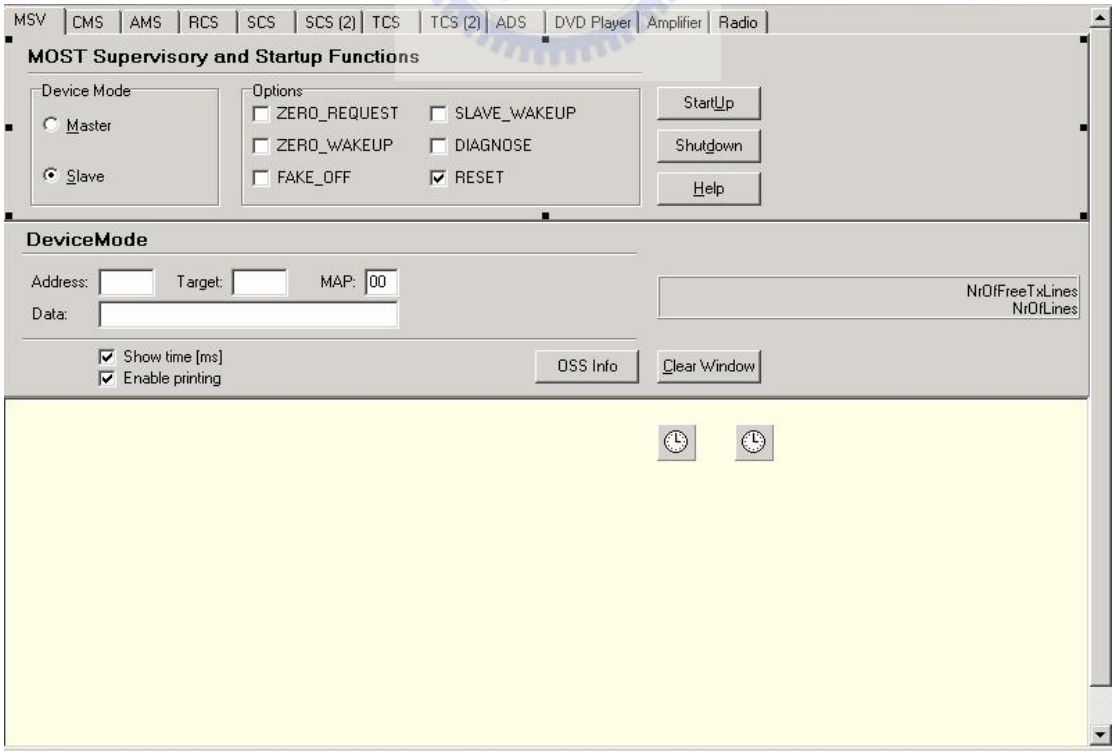

Fig 3-4 MSV application

3.2 CMS (Control Message Service)

The control message services is divided into transmission section and receiving section. Messages in both sections are represented by data structures. They are stored in a buffer of these structures. Access is handled via pointers to the buffers.

```
typedef struct Ctrl_Tx_Type
{
    byte Status;
    byte Priority;
    byte MsgType;
    byte Tgt_Adr_H;
    byte Tgt_Adr_L;
    byte Data[17];
    byte Length;
    #if (MAX_TX_HANDLE >0)
    byte TxHandle[MAX_TX_HANDLE];
    #endif
    #ifdef CTRL_FILTER_ID
    byte Filter_ID;
    #endif
} TCtrlTx, *pTCtrlTx;

TCtrlTx CtrlTxBuf[MAX_CTRL_TX_MSG];
pTCtrlTx TxCPtrOut, TxCPtrIn;

typedef struct Ctrl_Rx_Type
{
```

```
    byte Status;
    byte Rcv_Type;
    byte Src_Adr_H;
    byte Src_Adr_L;
    byte Data[17];
    #if (MAX_EXT_DATA > 0)
    byte ExtData[MAX_EXT_DATA];
    #endif
    #ifdef CTRL_FILTER_ID
    #endif
} TCtrlRx, *pTCtrlRx;

TCtrlRx CtrlRxBuf[MAX_CTRL_RX_MSG];
pTCtrlRx RxCPtrout, RxCPtrIn;
```

To send a message using Control Message Services, we can call the following APIs.

1. first get the pointer of transmitting
2. Using the gotten pointer to fill the message.
3. Send the message

The following example shows the steps to send a control message.

```
pTCtrlTx myPtr;

myPtr = CtrlGetTxPtr();    // get pointer
if (myPtr)
{
```

```
  // fill the message
  myPtr->Priority = 0x05;
  myPtr->MsgType = 0x00;
  myPtr->Tgt_Adr_H = 0x01;
  myPtr->Tgt_Adr_L = 0x98;
  myPtr->Length = 0x06;
  myPtr->Data[0] = 0x34;    // Function Block.
  myPtr->Data[1] = 0x00;    // Function ID
  myPtr->Data[2] = 0x20;    //
  myPtr->Data[3] = 0x00;    // OP Type
  myPtr->Data[4] = 0x01;    // DataLength
  myPtr->Data[5] = 0x04;    // Data

  CtrlSend(myPtr); // send message
}
```

The system message results will return to the callback function

CtrlTxStatus(byte status, struct Ctrl_Tx_Type *ptbuf)

Status	Name	Comment
0x00	XMIT_WRONGTARGET	Transmission failed. No response under chosen target address.
0x10	XMIT_SUCCESS	Transmission successful
0x11	XMIT_TYPEFAILED	Message was received, but receiving node does not support XMIT message type

0x20	XMIT_CRC	Transmission failed. Incorrect CRC.
0x21	XMIT_BUF	Transmission failed. Receive buffer of target device was full
0x40	XMIT_FIFO	No free entry in the Tx FIFO of the hardware layer. This error is possible only, if using the message based hardware layer interface
0x80	XMIT_TIMEOUT	No MTX event(message transmitted) after timeout

Table 3-3 Transmission results of control messages

The parameter ptbuf points at the MsgEntry in Tx buffer, so the application has access to the message if needed. If transmission failed, there are two possible ways to handle this event. First, delete the message. Second, retry transmission.

For deleting the message, CtrlFreeTxMsg() must be called.

void CtrlFreeTxMsg(void);

For retrying to transmit the message, CtrlTxRetry() must be called.

void CtrlTxRetry(void);

MsgType	Description
0x00	Application message, using any of the various addressing options.
0x01	System message: Remote Read
0x02	System message: Remote Write
0x03	System message: Resource allocation

0x04	System message: Resource De-allocation
0x05	System message: Remote GetSource

Table 3-4 Types of control messages

The following example shows the usage of the above APIs.

```
void CtrlTxStatus(byte status, struct Ctrl_Tx_Type *ptbuf)
{
    If (status != XMIT_SUCCESS)
    {
        If (++prbuf->TxHandle[0] < 5) // allow 4 high level retries
        {
            CtrlTxRetry();
        }
        else
            CtrlFreeTxMsg();
     }
 }
```

For receiving messages via CMS, the callback function CtrlRxOutMsg is provided. After reading the message, the function CtrlFreeRxMsg will be called to delete the message. The following example shows the usage of above functions.

```
byte CtrlRxOutMsg(struct Ctrl_Rx_Type *prbuf)
{
   // …
```

```
    // … read the message
    // …
    CtrlFreeRxMsg(prbuf);
    Return(1);
}
```

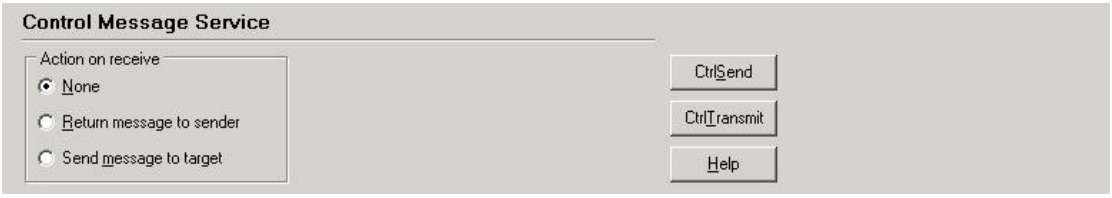

Fig 3-5 CMS application

3.3 AMS(Application Message Service)

This service is based on CMS (Control Message Service). AMS organizes incoming and outgoing messages on a higher level. The CMS limits the message to 17 bytes. AMS extends the length of messages limited by the CMS.

```
Typedef struct Msg_Tx_Type
{
    byte Status;
    byte Priority;
    word Tgt_Adr;
    byte Fblock_ID;
    byte Inst_ID;
    word Func_ID;
    byte operation;
    word Length;
```

```
    byte Data[MAX_MSG_TX_DATA];    // MAX_MSG_TX_DATA := maximum
                                   // number of data bytes
    byte TxHandle[MAX_TX_HANDLE]; // MAX_TX_HANDLE := number of
                                  // add-on bytes (will not be send)
  #ifdef CTRL_FILTER_ID
  byte Filter_ID;
  #endif

  #ifdef AMS_TX_ADD8
  byte *PtrDataExt; // optional pointer at externally stored payload
  #endif
  #ifdef AMS_TX_ADD9
  byte MidLevelRetries;
  #endif
} TMsgTx, *pTMsgTx;

Struct Msg_Tx_Type MsgTxBuf[MAX_MSG_TX_MSG];
// MAX_MSG_TX_MSG := Maximum number of messages to be buffered

Typedef struct Msg_Rx_Type
{
    byte Status;
    byte BlckCnt;
    byte UsageCnt;
    word Src_Adr;
    byte Fblock_ID;
```

```
    byte Inst_ID;
    word Func_ID;
    byte operation;
    word Length;
    byte Data[MAX_MSG_RX_DATA];    // MAX_MSG_RX_DATA := maximum
                                   // number of data bytes
    #if (MAX_EXT_DATA >0)
    byte ExtData[MAX_EXT_DATA];    // additional info, e.g. timestamp
    #endif
    #ifdef CTRL_FILTER_ID
    byte Filter_ID;
    #endif
    byte *DPtrIn;
    byte Rcv_Type;
    #ifdef AMS_RX_ADD5
    byte* PtrData;       // pointer at data field
    word SizeBuffer;                                  // size of data field
    struct Msg_Rx_ExtBuf_Type* PtrNext;           // pointer at next field
    struct Msg_Rx_ExtBuf_Type* PtrCurBufExt; // pointer at current buffer extension
    word CurSizeMsgBuf;                               // current size of buffer
    #endif

} TMsgRx, *pTMsgRx;

Typedef struct Msg_Rx_ExtBuf_Type
{
```

```
    byte* PtrData;                          // pointer at data field
    word SizeBuffer;                        // size of data field
    struct Msg_Rx_ExtBuf_Type* PtrNext;     // pointer at next field
    struct Msg_Rx_ExtBuf_Type* PtrPrev;     // pointer at previous field
} TMsgRxExtBuf, *pTMsgRxExtBuf;

struct Msg_Rx_Type MsgRxBuf[MAX_MSG_RX_MSG];
// MAX_MSG_RX_MSG := Maximum number of messages

Typedef struct Most_Header_Type
{
    Word Device_ID;    // source (on Rx side) or target address (on Tx side)
    byte FBlock_ID;
    byte Inst_ID;
    Word Func_ID;
    byte Operation;
    Word Length;
} TMostHeader, *pTMostHeader;

Typedef struct Msg_TxOpt_Type
{
   byte Priority;
   #if (MAX_TX_HANDLE > 0)
   byte TxHandle[MAX_TX_HANDLE];
   #endif
   #ifdef CTRL_FILTER_ID
```

```
    byte Filter_ID;
    #endif
    #ifdef AMS_TX_ADD9
    byte MidLevelRetries;
    #endif
} TMsgTxOpt, *pTMsgTxOpt;
```

byte MsgCheckTxBuffer(void):

This function will return the number of free MsgEntries.

struct Msg_Tx_Type *MsgGetTxPtr(void):

This function will get the free MsgEntry.

void MsgSend(struct Msg_Tx_Type *ptbuf)

This function will set the AMS MsgEntry.

void MsgFreeTxMsg(void);

This function will delete the MsgEntry;

void MsgTxRetry(void);

This function will retry transmission.

void MsgTxPostpone(void);

Postpone a message to retry the transmission later

void MsgTxStatus(byte status, struct Msg_Tx_Type *ptbuf);

This callback function will report the transmission status.

```
{
    Struct Msg_Tx_Type *myMsgPtr;

    If( MsgCheckTxBuffer() == 0)      // Check if there is a free buffer entry
```

```
        return;
    myMsgPtr = MsgGetTxPtr();       // get the free MsgEntry
    // fill up the message
    myMsgPtr->Tgt_Adr = 0x198;
    myMsgPtr->Inst_ID = 0x00;
    myMsgPtr->FBlock_ID = 0x34;
    myMsgPtr->Func_ID = 0x0198;
    myMsgPtr->Operation = 0x00;
    myMsgPtr->Length = 100;
    myMsgPtr->Data[0] = … ;

    MsgSend(myMsgPtr); // send the message
}
void MsgTxStatus(byte Status, struct Msg_Tx_Type *ptbuf)
{
     switch (Status)
     {
          case XMIT_SUCCESS:     // Success: buffer is cleared automatically
             break;

          case XMIT_CRC:
             MsgTxRetry();              // re-send only failed telegram
             break;

          case XMIT_BUF:
             MsgTxPostpone();       // postpone a message
```

```
            MsgSend(ptbuf);         // Re-send whole message immediately again
            break;

        Default:
            MsgFreeTxMsg();         // cancel transmission
            break;

    }
}
```

The following APIs are used for receiving the AMS messages.

byte MsgGetMsgCnt(void):

This function provides information about the number of messages ready to be copied or accessed by the application.

struct Msg_Rx_Type *MsgGetRxPtr(void):

This function returns a pointer to the next message to be read. If no message is available, NIL will be returned.

void MsgFreeRxMsg(struct Msg_Rx_Type *prbuf);

This function will release the MsgEntry immediately.

byte MsgReceive(struct Msg_Rx_Type *prbuf);

This function copies the MsgEntry to the local memory.

byte MsgRxOutMsg(struct Msg_Rx_Type *prbuf);

This callback function is used to receiving the AMS message.

The return value is shown in Table 3-5

Value	Comment
0x00	Message will be read later, or it was already copied by MsgReceive()
0x01	Pointer was grabbed(Acknowledge)
0x02	Request for re-triggering. When for instance the application cannot handle the message at that time. The usage counter is not incremented
0x03	Same as 0x02, but with the difference that the usage counter (UsageCnt) is incremented. This can be used to force a retrigger of the same message to pass it to different applications. Please note, that the buffer entry must be released “n” times, after the message has been read. Otherwise the buffer entry will not be cleared until the usage counter will be zero.

Table 3-5 Results of receiving application message

```
byte MsgRecMethod;
Struct Msg_Rx_Type *MyMsgRxInPtr;

byte MsgRxOutMsg(struct Msg_Rx_Type *prbuf)
{
    switch(MsgRecMethod)
    {
        case 0:    // direct reading
            // …
            // … read the message
            // …
            MsgFreeRxMsg(RxPtr);
```

```
        case 1:    // delayed reading
           // message handling is done by polling outside the callback function
        case 2:    // copying
           // copy the message in a buffer, message handling can be done
           // outside the callback function
           MsgReceive(MyMsgRxInPtr); // copy the message in a buffer.
           MsgFreeRxMsg(MyMsgRxInPtr);
}

void RxInterpreter(void)      // for delay reading
{
    struct Msg_Rx_Type *RxPtr;
    byte msgCount;

    msgCount = MsgGetMsgCnt(void);

    While(msgCount >0)
     {
        RxPtr = MsgGetRxPtr();// get available message
        // …
        // …reading the message
        // …
        MsgFreeRxMsg(RxPtr);
        msgCount--;
     }
}
```

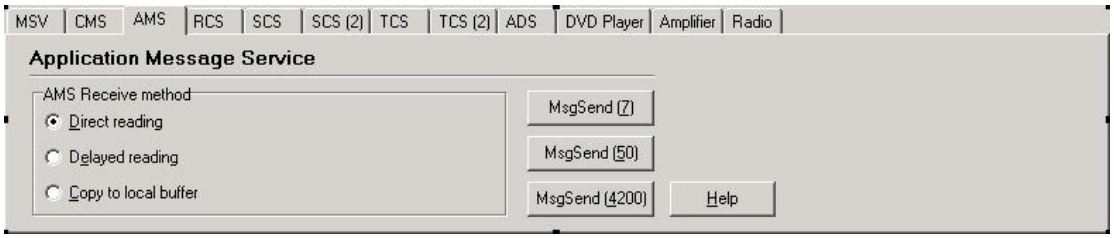

Fig 3-6 AMS application

3.4 RCS (Remote Control Service)

This service provides the sending of Remote System Message. It is divided into the services Remote Read and Remote Write.

byte RemoteWrite(word tgt_adr, byte map, byte *ptbuf, byte num_bytes);

This function will check whether there is space in Control Message TxBuffer available and generates a Control Message. The error mask table is shown in Table3-6.

void RemoteWriteComplete(byte XmitStatus, struct Ctrl_Tx_Type *ptrbuf);

This function provides notification of the application by the MOST NetServices, on termination of the Remote Write process.

XmitStatus	Name	Comment
0x10	XMIT_SUCCESS	Data successful transmitted
0x00	XMIT_WRONGTARGET	Transmission failed. No response under chosen target address
0x20	XMIT_CRC	Transmission failed. Incorrect CRC.
0x40	XMIT_FIFO	No free entry in the Tx FIFO of the hardware layer. This error is possible only, if using the message based hardware layer interface.

0x80	XMIT_TIMEOUT	No MTX event (message transmitted) after timeout.

Table 3-6 Transmit status for Remote Write messages

Element of Structure	Comment
ptrbuf->Tgt_Adr_H	Target Address High
ptrbuf->Tgt_Adr_L	Target Address Low
ptrbuf->Data[1]	MAP (Address to write to)
ptrbuf->Data[2]	n = count of data bytes to be written (max. 8)
ptrbuf->Data[3]	D1
…	…
ptrbuf->Data[n+2]	Dn

Table3-7 Structure of a Remote Write message in Tx buffer

byte buf[8];

byte status;

Buf[0] = …

…

Buf[7]=…

Status = RemoteWrite(0x0198, 0x10,&Buf[0],8);

byte RemoteRead(word tgt_adr, byte map, byte num_bytes, byte *prbuf);

This function will read data bytes from the target device. The maximum number of data bytes is eight.

void RemoteReadComplete(byte XmitStatus, struct Ctrl_Tx_Type *ptrbuf);

This callback function will tells the application the result of remote reading process.

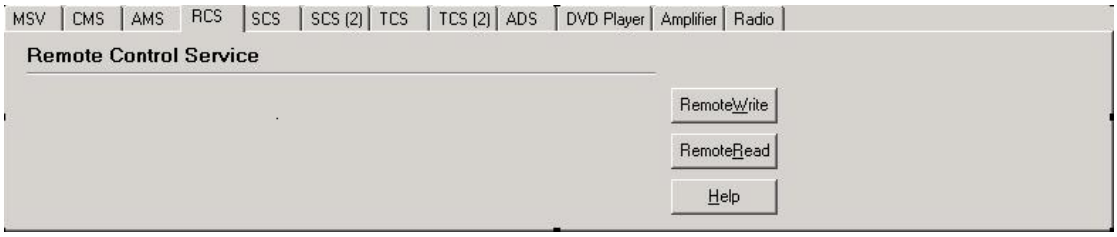

Fig 3-7 RCS Application

3.5 SCS

This service contains for reserving and releasing of Synchronous Source Data Channels and for establishing connections between the inputs and outputs of the MOST Transceiver and Source Data Channels. The connection is done by writing the Routing Engine (RE) of the MOST Transceiver.

Functionality	Parameters For Function Call	Result
Allocate only	Number of channels, Ptr to local result buffer	Status, channels
Allocate + Route	Number of channels, Ptr to local result buffer, Source port info	Status, channels
Deallocate + restore Routing	Number of channels, channels	Status
Deallocate only	Label	Status
Route Source Data To Bus (InConnect)	Number of channels, channels, Source port info	-
Remove Source Data From	Number of channels, channels	-

Bus(InDisconnect)		
Retrieve channels belonging to a certain level	(optional: number of channels), Label	Channels
Route Source Data From Bus (OutConnect)	Number of channels, channels, Source port info	-
Stop Routing Source Data From Bus (OutDisconnect)	Number of channels, Source port info	-
Retrieve channels belonging to a certain label	(optional: number of channels), Label	Channels

Table 3-8 Survey of functions for Source Data sources

byte SyncAllocOnly(byte num_channels, byte *prbuf);

This function reserved channels for Synchronous Source Data, but no source data is routed to the reserved channels.

SyncAllocComplete(byte Status,byte Num_Channels_Req,byte Num_Channels_Free)

This callback will inform the application the availability of channel data or errors.

byte SyncAlloc(byte num_channels, byte *pchrbuf, struct SrcData_Type *ptrsrc);

This function reserved channels for Synchronous Source Data, and routes Data to the reserved channels. The first parameter is the number of channels to be allocated. The number of channels can be allocated at once is limited to eight. The third parameter contains the information of Source Data Port.

```
byte Channel_Id[4];    // Result buffer storing all channels

void Alloc_Example(void)
```

```
{
   Status_1 = SyncAllocOnly(4, &Channel_Id[0]);
   // if status != 0x00, operation not succeeded
}

void SyncAllocComplete(byte status, byte num_channels_req, byte
num_channels_free)
{
   switch (status)
   {
      case ALLOC_XMIT_GRANT:
         //
         //   … Success
         //
         break;
      default:
         // if any error occurred:

         CtrlFreeTxMsg();
         break;
   }
}
```

Status	Name	Comment
0x11	ALLOC_XMIT_GRANT	Xmit success and Allocation grant:

		All requested channels allocated.
0x12	ALLOC_XMIT_BUSY	Xmit success, but Timing Master busy: No channels allocated.
0x13	ALLOC_XMIT_DENY	Xmit success, but not enough free channels available: No channels allocated.
0x20	ALLOC_XMIT_CRC	Xmit failed due to CRC error: No channels allocated.
0x40	ALLOC_FIFO	No free entry in the Tx FIFO of the hardware layer. This error is possible only, if using the message based hardware layer interface.
0x80	ALLOC_XMIT_TIMEOUT	No MTX event(message transmitted) after timeout.

Table 3-9 Status of allocating channels

byte SyncDeallocOnly(byte label);

This function releases channels for Synchronous Source Data, but does not restore Routing Engine(RE). If 0x7F is transferred as parameter, the entire allocation table will be reset (Deallocate All). The return value contains the error mask.

byte SyncDealloc(byte num_channels, byte *pchtbuf);

This function releases the respective channels and restore RE.

void SyncDeallocComplete(byte Status, byte label);

This callback function will inform the application the result of deallocation.

Status	Name	Comment
0x11	DEALLOC_XMIT_GRANT	Xmit success and De-allocation grant: Channels are successfully de-allocated.
0x12	DEALLOC_XMIT_BUSY	Xmit success, but Timing Master busy: No channels de-allocated.
0x14	DEALLOC_XMIT_WRONG	Xmit success, but connection label was wrong: No channels de-allocated.
0x20	DEALLOC_XMIT_CRC	Xmit failed due to CRC error: No channels de-allocated.
0x40	DEALLOC_XMIT_FIFO	No free entry in the Tx FIFO of the hardware layer. This error is possible only, if using the message based hardware layer interface.
0x80	DEALLOC_XMIT_TIMEOUT	No MTX event(message transmitted) after timeout.

Table 3-10 Status of deallocating channels

void SyncInConnect(byte num_channels, byte *ptrch, struct SrcData_Type *ptrsrc);

The function connects the Source Data Ports of a source to already allocated channels. The first parameter contains the number of channels. The second parameter provides a pointer at the array containing the respective channels. The third parameter contains a pointer at the respective Source Port information.

void SyncInDisconnect(byte num_channels, byte *ptrch);

This function restores the RE, which means the Source Data Ports of the source will be separated from the bus.

byte SyncFindChannels(byte num_channels, byte label, byte *pchrbuf);

By calling this function, an application can retrieve which channels belong to a certain connection label. The second parameter contains the respective label.

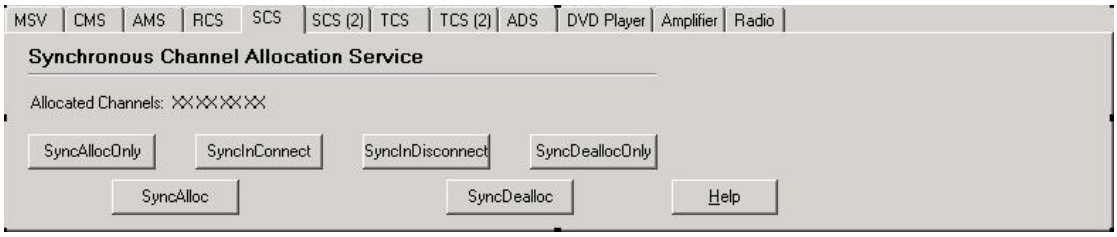

Fig 3-8 SCS application

void SyncOutConnect(byte num_channels, byte *ptrch, struct SrcData_Type *ptrsrc);

This function routes Source Data channels to a sink. The first parameter contains the number of channels. The second parameter provides a pointer at the array containing the respective channels. The third parameter contains a pointer at the respective Source Port Information.

void SyncOutDisconnect(byte num_channels, struct SrcData_Type *ptrsrc);

This function restores routing engine (RE) at a sink.

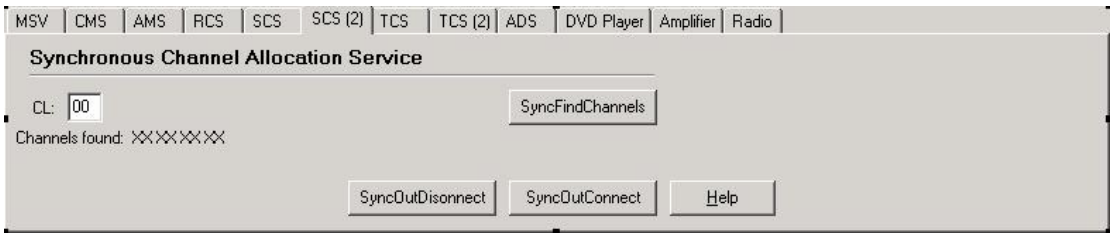

Fig 3-9 SCS application II

3.6 TCS

This service contains functions for reserving and releasing of channels for transporting transparent data, and for connecting inputs and outputs of the MOST

Transceiver. This service is based on SCS (Synchronous Channel Allocation Service).

byte TransAllocOnly(byte num_channels, byte *ptrch);

This function will reserve channels for transparent data transfer.

byte TransAlloc(byte num_channels, byte *ptrch, byte *ptrsrc);

This function will reserve and connect the channels.

byte TransDeallocOnly(byte label);

This function frees channels, RE must be restored by a second function call.

byte TransDealloc(byte num_channels, byte *ptrch);

By calling this function, the respective channels are released again and the RE will be restored.

void TransInConnect(byte num_channels, byte *ptrch, byte *ptrsrc);

By calling this function, the source port of a source will be routed to already allocated channels.

void TransInDisconnect(byte num_channels, byte *ptrch, byte *ptrsrc);

This function restores RE.

void TransOutConnect(byte num_channels, byte *ptrch, byte *ptrsrc);

By calling this function, the Source Port at a sink will be routed.

void TransOutDisconnect(byte num_channels, byte *ptrsrc);

This function resets the routing at a sink.

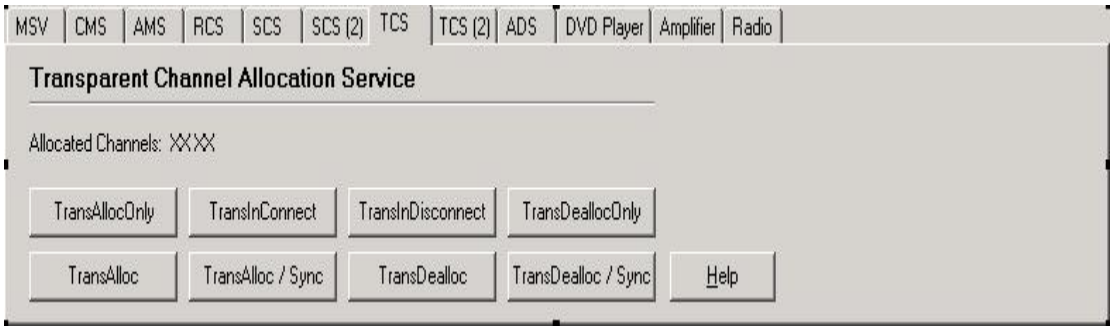

Fig 3-10 TCS application

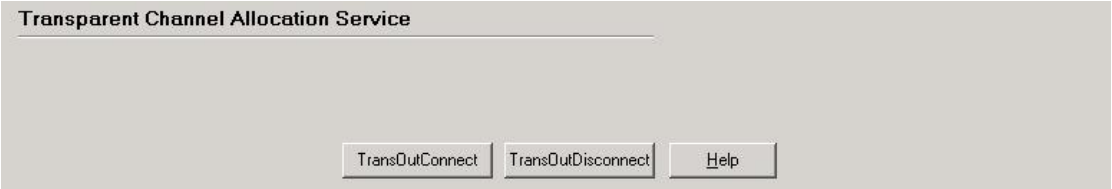

Fig 3-11 TCS application II

3.7 ADS(Asynchronous Data Transmission Service)

This service provides sending and receiving of data packets (Asynchronous Data Transfer). It is divided up into Tx and Rx sections.

```
Typedef struct Data_Tx_Type
{
   byte Status;
   byte Priority;
   byte Tgt_Adr_H;
   byte Tgt_Adr_L;
   byte Data[48];
   byte Length;
   byte TxHandle[MAX_DATA_TX_HANDLE];
} TDataTx, *pTDataTx;
struct Data_Tx_Type DataTxBuf[MAX_DATA_TX_MSG];
// MAX_DATA_TX_MSG := Max. number of packets to be buffered
```

Struct Data_Tx_Type *DataGetTxPtr(void);

This function will return the next free MsgEntry. If there is no free MsgEntry, NIL will be returned.

Void DataSend(struct Data_Tx_Type *ptbuf);

This function releases a MsgEntry for being sent. In that case, status of the entry will be “ready”.

byte DataTransmit(byte priority, word tgt_adr, byte *ptbuf, byte length);

This function directly copies data into a MsgEntry and release it for being sent.

byte DataCheckTxBuffer(void);

This function checks whether there is free space in the buffer. It returns the number of available MsgEntries.

```
Typedef struct Data_Rx_Type
{
   byte Status;
   byte Tgt_Adr_H;       // Received Target Address
   byte Tgt_Adr_L;
   byte Src_Adr_H;
   byte Src_Adr_L;
   byte Data[48];
   #if (MAX_DATA_EXT_DATA > 0)
   byte ExtData[MAX_DATA_EXT_DATA]; // extend data (e.g. timestamp)
   #endif
} TDataRx, *pTDataRx;
Struct Data_Rx_Type DataRxBuf[MAX_DATA_RX_MSG];
  // MAX_DATA_RX_MSG := Max. number of packets to be buffered
```

byte DataGetMsgCnt(void);

This function returns the number of packets which are ready to be read.

struct Data_Rx_Type *DataGetRxPtr(void);

This function returns a pointer pointing to a MsgEntry containing a packet. After finishing the access, the MsgEntry must be released explicitly.

void DataFreeRxMsg(struct Data_Rx_Type *prbuf);

This function explicitly releases the single MsgEntry.

byte DataReceive(struct Data_Rx_Type *prbuf);

This function copies data from a MsgEntry in the Rx buffer, to a local memory area and releases the entry immediately.

byte DataRxOutMsg(struct Data_Rx_Type *prbuf);

This Callback Function informs the application that there is a packet which can be read. The function must return 0x00 if the packet will be polled later on, or if it was copied via DataReceive(). If the data was read, 0x01 must be returned.

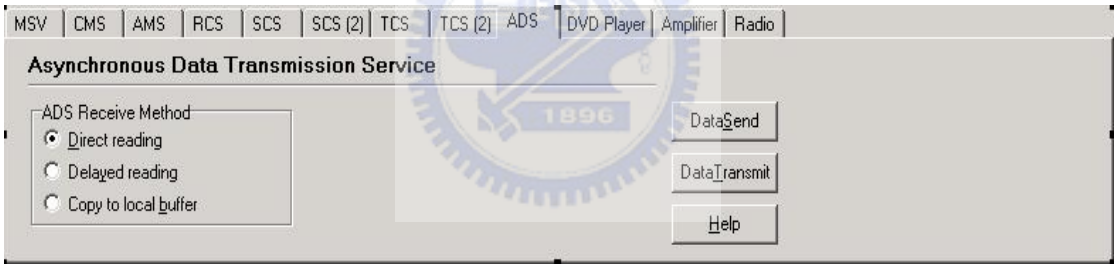

Fig 3-12 ADS application

3.8 Transceiver Control Service

This service provides functions for adjusting certain properties of the MOST Transceiver like its logical address or group address. These functions are provided that are used in timing masters only.

void MostSetNodeAdr(word adr);

This function provides setting of the logical address of the MOST Transceiver.

void MostSetGroupAdr(byte group_adr);

This function provides setting of the group address of the MOST Transceiver.

byte MostGetNodePos(void);

This function provides reading of the current node position of the MOST Transceiver.

Chapter 4

Network Multimedia Applications

In this chapter, we will develop multimedia applications of a MOST network.

4.1 Multimedia Control

As shown in Fig4-1, there are Radio Tuner 4 MOST, DVD Player 4 MOST, MOST PCI Board, and Amplifiers 4 MOST on the MOST network. According to the MOST Protocol, DeviceID.FBlockID.InstID.FktID.OPType.Length(Data), we can implement our control programs with AMS NetServices API.

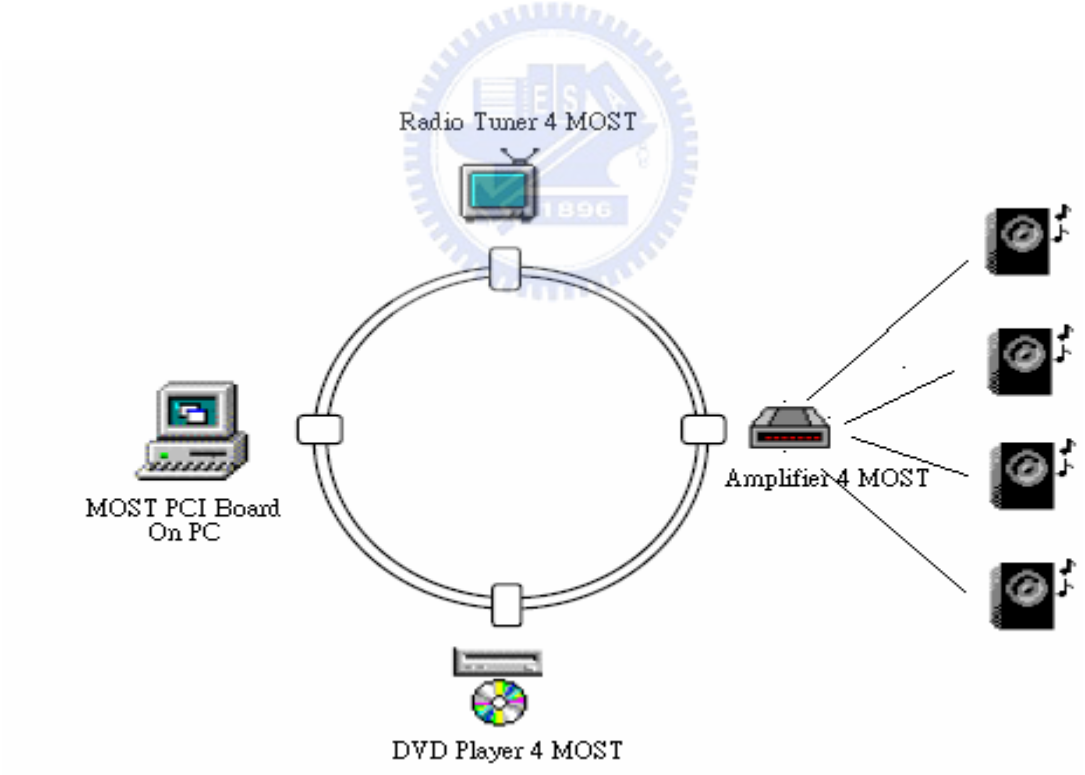

Fig 4-1 MOST network topology

Fig 4-2 Picture of MOST network

4.1.1 DVD Player 4 MOST Control Program

As shown in Fig 4-3, the DVD Player from OASIS SiliconSystems is a MOST device offering a real-time interface to distribute Video and Audio streams from a broad range of media including CDDA and DVD via a MOST network.

Fig 4-3 Picture DVD Player 4 MOST

The Table 4-1 and Table 4-2 below are parts of MOST DVD Player Manual.

FBlock	Function	OPType	Parameter
DVDVideoPlayer (0x34)	DeckStatus (0x200)	Set	DeckStatus
		Get	
		Status	DeckStatus
		Error	ErrorCode, ErrorInfo

Table 4-1 Function Format [10]

Basic datatype	Range of values	Code	Description
Enum	0x00..0x23	0x00	Play
		0x01	Stop
		0x02	Pause
		0x03	Load
		0x04	Unload
		0x05	Search Forward
		0x06	Search Backward
		0x20	Slow Motion Forward
		0x21	Slow Motion Backward
		0x23	PreStop (play resumes at old position)

Table 4-2 Parameter of DeckStatus

Each control button in Fig4-4 can be implemented with NetServices AMS API. The Table 4-3 contains the mapping of control buttons and corresponding MOST

protocols.

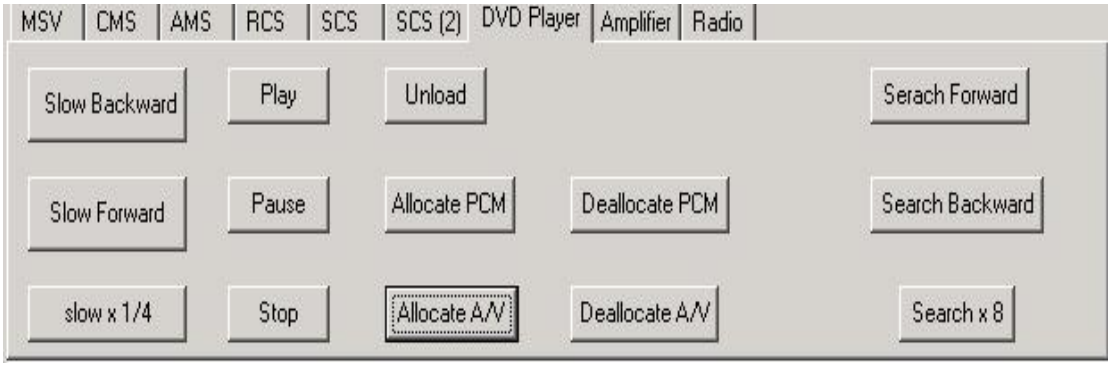

Fig 4-4 DVD Player 4 MOST Control Program

action	DeviceID	FBlock	InstID	FktID	OPType	Length	Data
Play	0x198	0x34	0x00	0x200	0x00	1	0x00
Stop	0x198	0x34	0x00	0x200	0x00	1	0x01
Pause	0x198	0x34	0x00	0x200	0x00	1	0x02
Load	0x198	0x34	0x00	0x200	0x00	1	0x03
Unload	0x198	0x34	0x00	0x200	0x00	1	0x04
Search Forward	0x198	0x34	0x00	0x200	0x00	1	0x05
Search Backward	0x198	0x34	0x00	0x200	0x00	1	0x06
Slow Motion Forward	0x198	0x34	0x00	0x200	0x00	1	0x20
Slow Motion Backward	0x198	0x34	0x00	0x200	0x00	1	0x21

Allocate A/V	0x198	0x34	0x00	0x101	0x02	1	0x02
Deallocate A/V	0x198	0x34	0x00	0x102	0x02	1	0x02
Allocate PCM	0x198	0x34	0x00	0x101	0x02	1	0x01
Deallocate PCM	0x198	0x34	0x00	0x102	0x02	1	0x01
Slow x 1/4	0x198	0x34	0x00	0x456	0x00	1	0x04
Search x 8	0x198	0x34	0x00	0x457	0x00	1	0x08

Table 4-3 Protocol Mapping of DVD Control Program [10]

4.1.2 Radio Tuner 4 MOST Control Program

The Radio Tuner 4 MOST from OASIS SiliconSystems is implemented using a single OS88350 Radio Controller 4 MOST to support all control functions and signal processing necessary to build a high quality AM and FM stereo car radio tuner. The Fig 4-5 shows the picture of Radio Tuner 4 MOST.

Fig 4-5 Picture of Radio Tuner 4 MOST

Each control button is shown in Fig4-6. The “AM” and “FM” buttons are used to switch the wavebands. The user can adjust the frequency by “Incr” and “Decre” buttons. The “Allocate2” and “Deallocate2” buttons are used to allocate or de-allocate the real-time channels for radio voice. The “Search Up” and “Search Down” buttons can be used to auto-scan the frequency upward or downward. Each button can be mapped to the MOST Protocols as shown in Table 4-4.

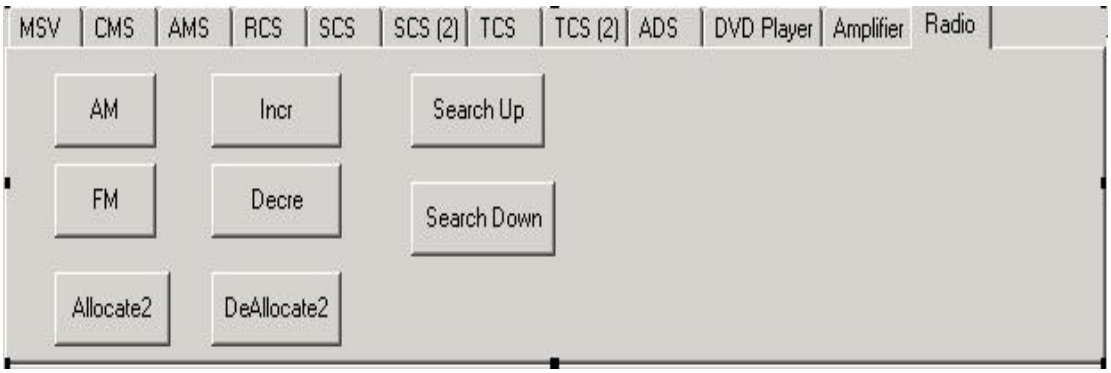

Fig 4-6 Radio Tuner 4 MOST Control Program

action	DeviceID	FBlock	InstID	FktID	OPType	Length	Data
AM	0x180	0x40	0x00	0x200	0x02	3	0x010002
FM	0x180	0x40	0x00	0x200	0x02	3	0x010001
Allocate2	0x180	0x40	0x00	0x101	0x02	1	0x01
Deallocate2	0x180	0x40	0x00	0x102	0x02	1	0x01
Incr	0x180	0x40	0x00	0x200	0x03	1	0x01
Decre	0x180	0x40	0x00	0x200	0x04	1	0x01
Search Up	0x180	0x40	0x00	0x204	0x00	1	0x04
Search Down	0x180	0x40	0x00	0x204	0x00	1	0x0A

Table 4-4 Protocol Mapping of Radio Tuner 4 MOST Control Program[11]

4.1.3 Amplifier 4 MOST Control Program

The Amplifier 4 MOST is implemented using a single OS88558 amplifier controller to do all processing necessary to transport audio information between a MOST network and external speakers. The Fig 4-7 shows the picture of Amplifier 4 MOST.

Fig 4-7 Picture Amplifier 4 MOST

The control program is shown in Fig 4-8. It can be used to connect the real-time audio channels on a MOST network and adjust the volume of external speakers. The mapping MOST Protocols are shown in Table 4-5.

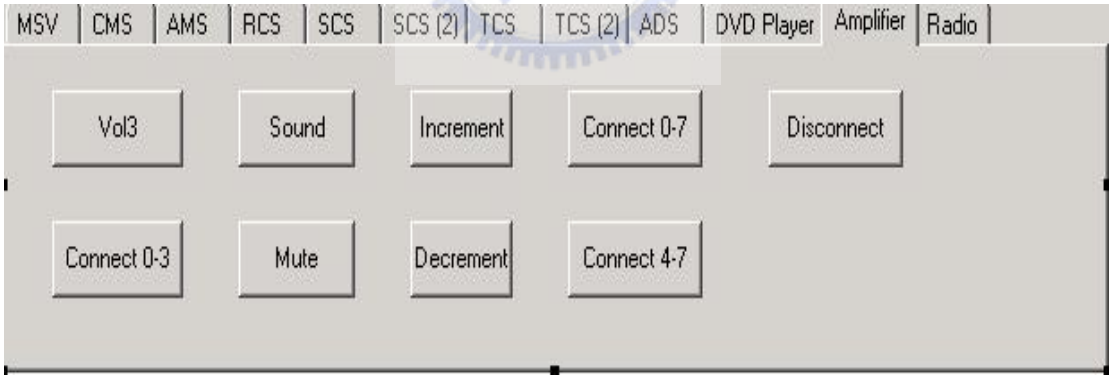

Fig 4-8 Amplifier 4 MOST Control Program

action	DeviceID	FBlock	InstID	FktID	OPType	Length	Data
Vol3	0x188	0x22	0x00	0x400	0x00	1	0x32
Connect 0-3	0x188	0x22	0x00	0x111	0x00	6	0x01 0x01

							0x00 ~ 0x03
Sound	0x188	0x22	0x00	0x113	0x02	2	0x0100
Mute	0x188	0x22	0x00	0x113	0x02	2	0x0101
Increment	0x188	0x22	0x00	0x400	0x03	1	0x01
decrement	0x188	0x22	0x00	0x400	0x04	1	0x01
Connect 4-7	0x188	0x22	0x00	0x111	0x02	6	0x01 0x01 0x04 ~ 0x07
Connect 0-7	0x188	0x22	0x00	0x111	0x02	10	0x04 0x01 0x00 ~ 0x07
Disconnect	0x188	0x22	0x00	0x112	0x02	1	0x01

Table 4-5 Protocol Mapping of Amplifier 4 MOST Control Program [12]

4.2 Listen to the Radio

The Fig 4-9 shows the procedures to listen to the radio on a MOST network. First, allocate synchronous channels and connecting the Radio Tuner 4 MOST as input, as shown in Fig 4-10. Second, connect to MOST 4 Amplifier as output. This step is shown in Fig 4-11. Then we can turn on volume and listen to radio by Radio 4 MOST Control Program as shown in Fig 4-12 and Fig 4-13.

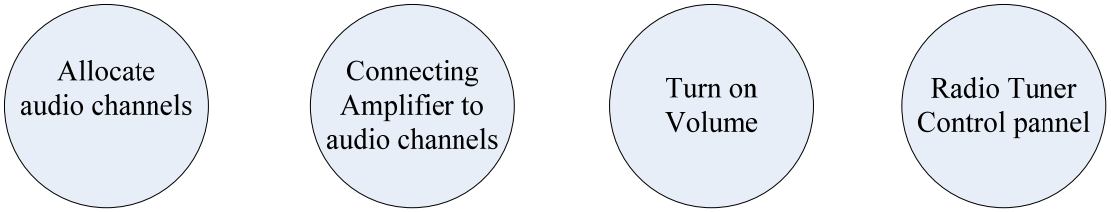

Fig 4-9 Flow diagram of "Listen to the Radio"

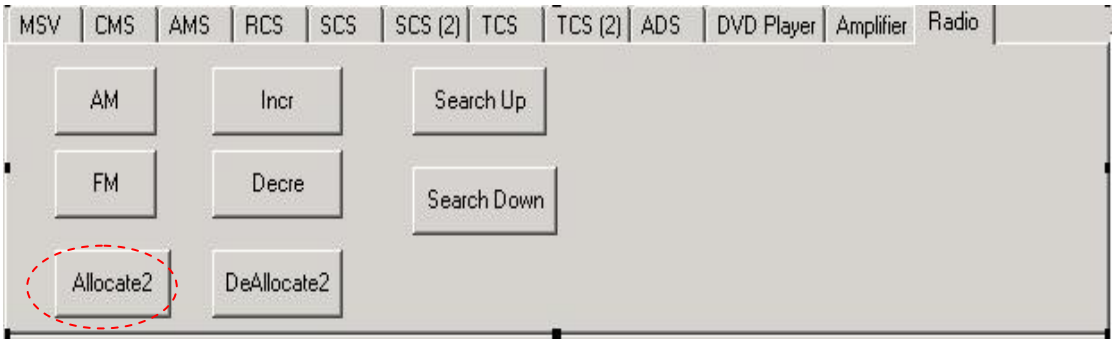

Fig 4-10 Application 1-1

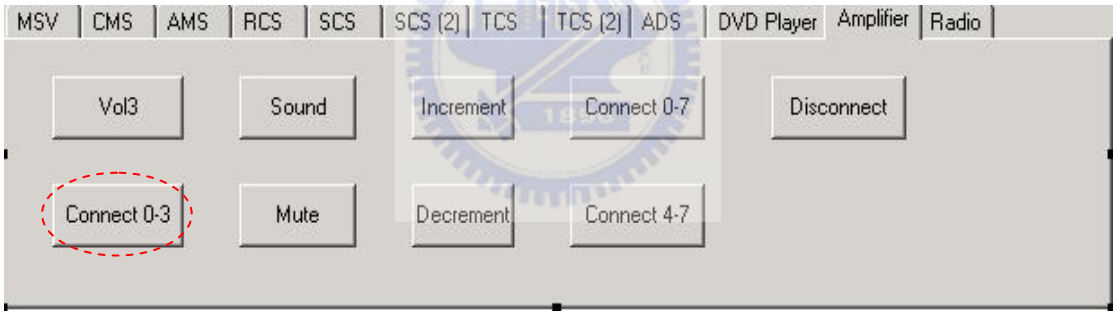

Fig 4-11 Application 1-2

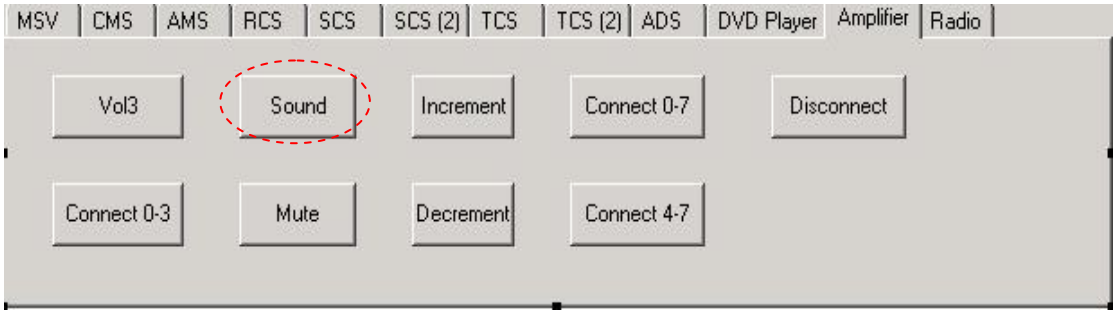

Fig 4-12 Application 1-3

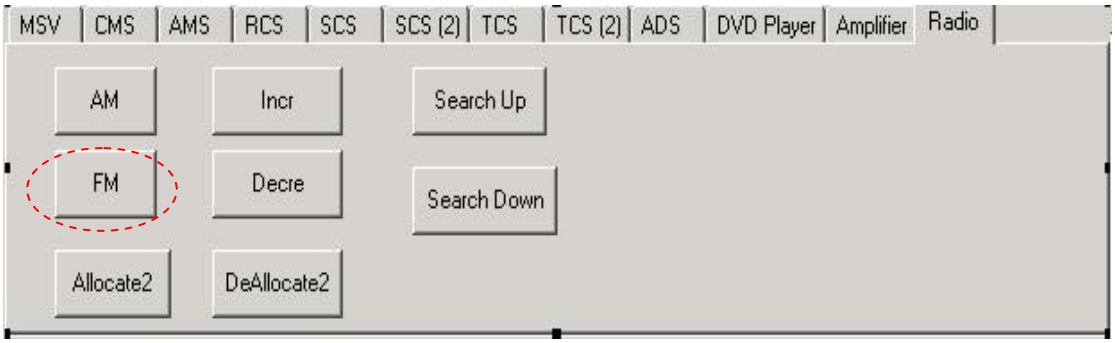

Fig 4-13 Application 1-4

4.3 Playing Wave File on a MOST Network

This application transports wave stream from the PC side to a MOST network. As shown in Fig 4-14, the wave file is transformed to the wave stream by Wave Parsar, and feeds to a MOST network. The Fig 4-15~ Fig 4-18 shows the steps to play the wave stream on a MOST network.

Fig 4-14 Transport wave steam to a MOST network [13]

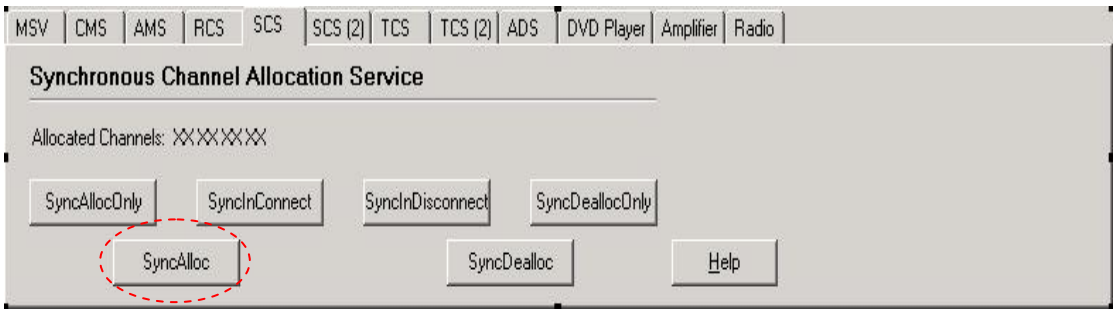

Fig 4-15 Application 2-1

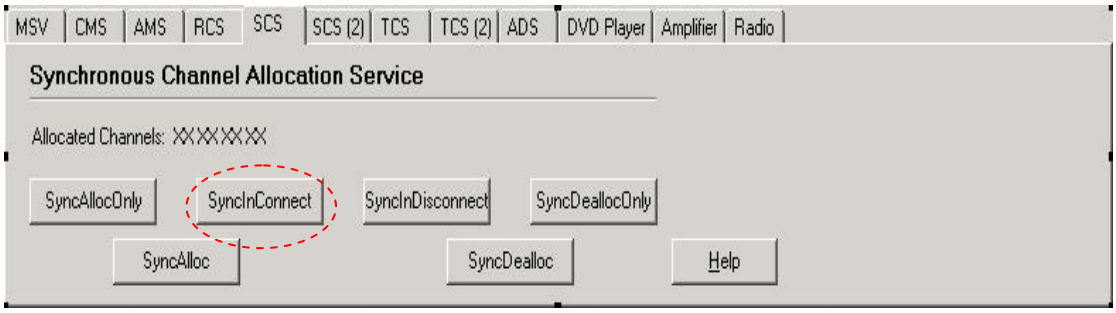

Fig 4-16 Application 2-2

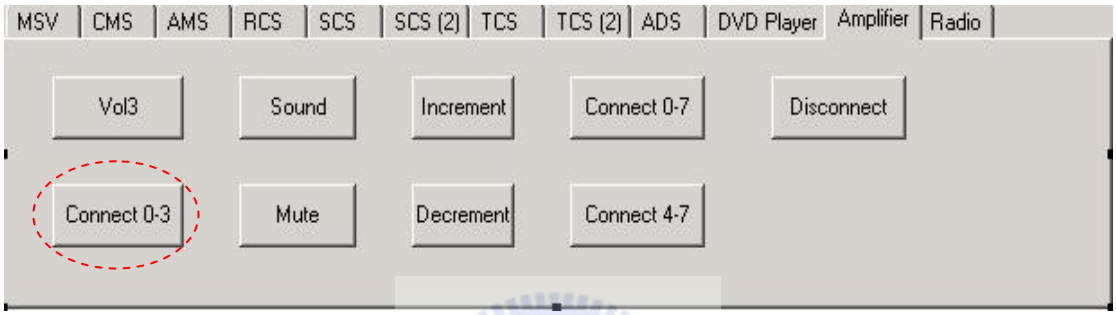

Fig 4-17 Application 2-3

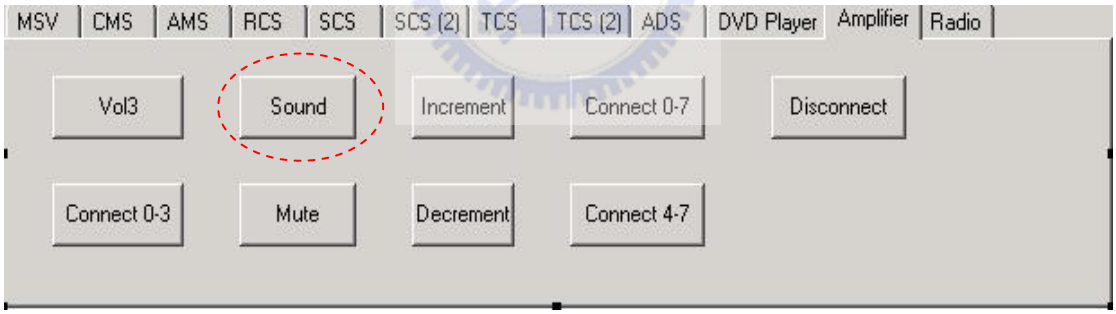

Fig 4-18 Application 2-4

4.4 Capture Mpeg-2 Stream by MOST Capture

In this section, we will introduce how to get the Mpeg-2 program stream from the DVD Player 4 MOST to the PCI board on PC and playing the mpeg-2 video on the PC Windows. There are two procedures. First, allocate the A/V Composite channels for DVD Player 4 MOST, then connect the output channels to the MOST PCI board. This part is achieved by MOST NetServices API. The procedure is shown in Fig 4-5.

Second, decode the Mpeg-2 stream and play it on PC Windows. This part can be achieved by the Microsoft NetShow API.

4.4.1 Mpeg-2 Transportation

To transport the Mpeg-2 stream from DVD Player 4 MOST to MOST PCI Board, the sixteen synchronous channels should be allocated by the DVD Player. After channels are allocated, the program should connect the output channels to MOST PCI board. This can be done by SCS NetServices API. After these steps, the mpeg-2 stream is transported from the DVD Player to MOST PCI Board. The Fig 4-19 shows the flow diagram of Mpeg-2 transportation. Fig 4-20 ~ Fig 4-23 shows each step in the control program.

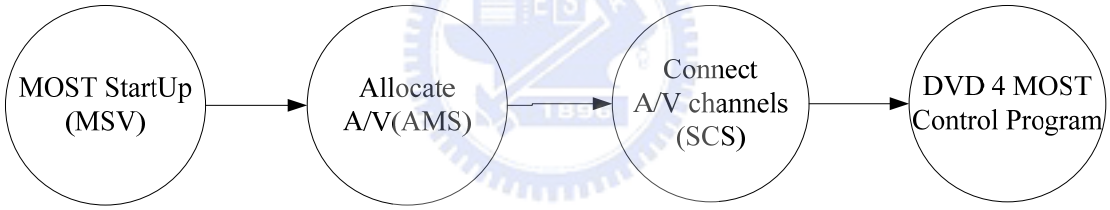

Fig 4-19 Transport Mpeg-2 stream from DVD Player 4 MOST to MOST PCI Board

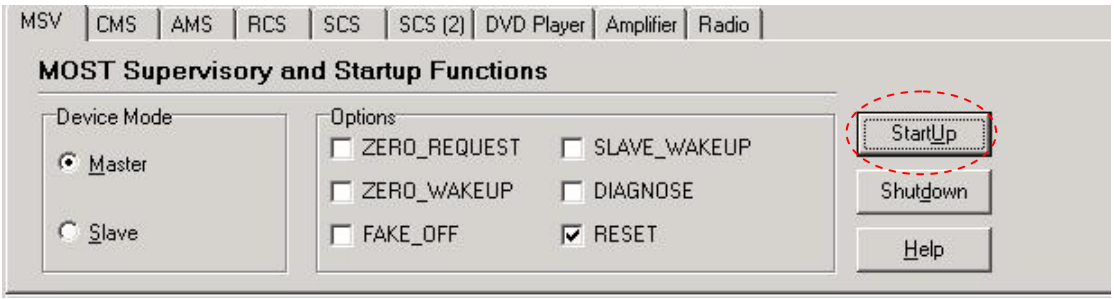

Fig 4-20 Application 3-1

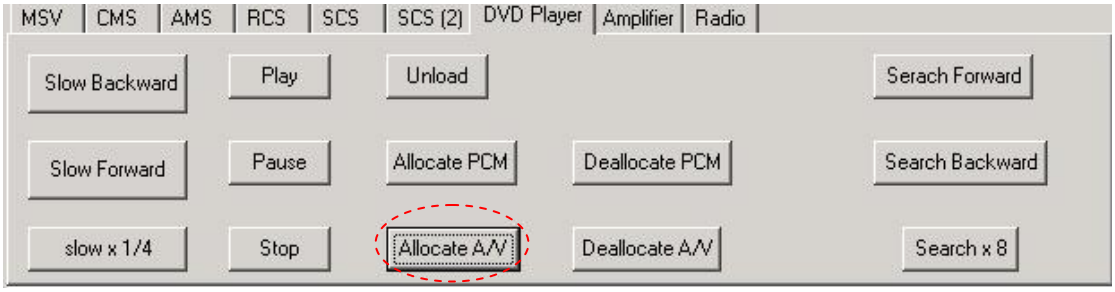

Fig 4-21 Application 3-2

Fig 4-22 Application 3-3

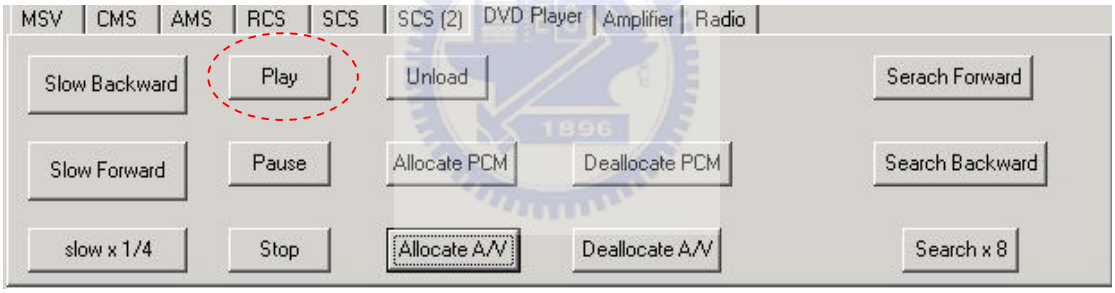

Fig 4-23 Application 3-4

4.4.2 Decode and Playing Mpeg-2 Stream on PC Windows

After receiving the Mpeg-2 stream from the MOST network, the program should demultiplex the Mpeg-2 stream, which will separate the video and audio stream. The program feeds the audio stream to Mpeg-2 Audio Decoder and the video stream to Mpeg-2 Video Decoder, then output audio data to the DirectSound filter to play the sound and output video data to the Video Render filter to show the video on PC Windows. The block diagram is shown in Fig 4-24.

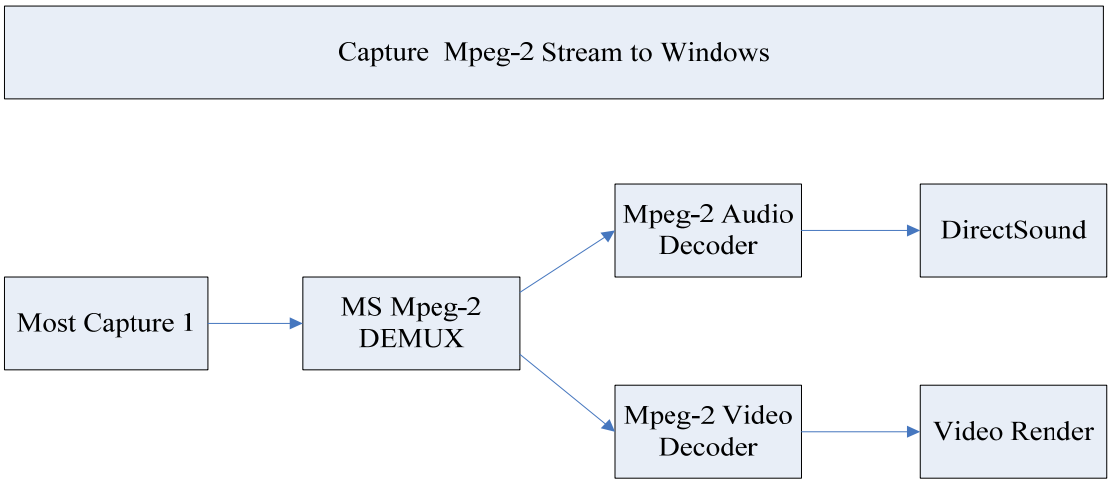

Fig 4-24 Decode and Play Mpeg-2 Streaming

The property of MOST Capture should be properly configured. As shown in Fig 4-25. The channel number should be set to the value as the same as allocated channel number. The channel width should be set to 16 Bytes since the number of allocated channels is sixteen. Also the property of Mepg-2 Demultiplexer should be properly configured as shown in Fig 4-26. "stream_id" and "Content" of the output pins should follow Mpeg-2 standard [14] [15]. Fig 4-27 shows the picture of captured Mpeg-2 video.

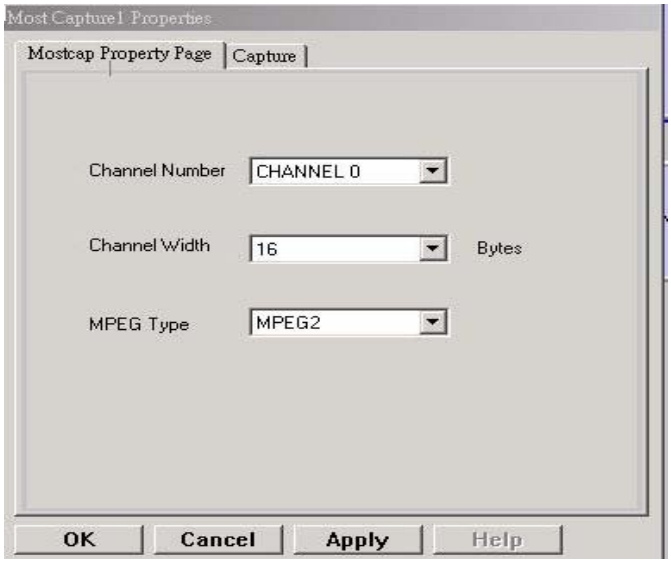

Fig 4-25 MOST Capture Property Setting

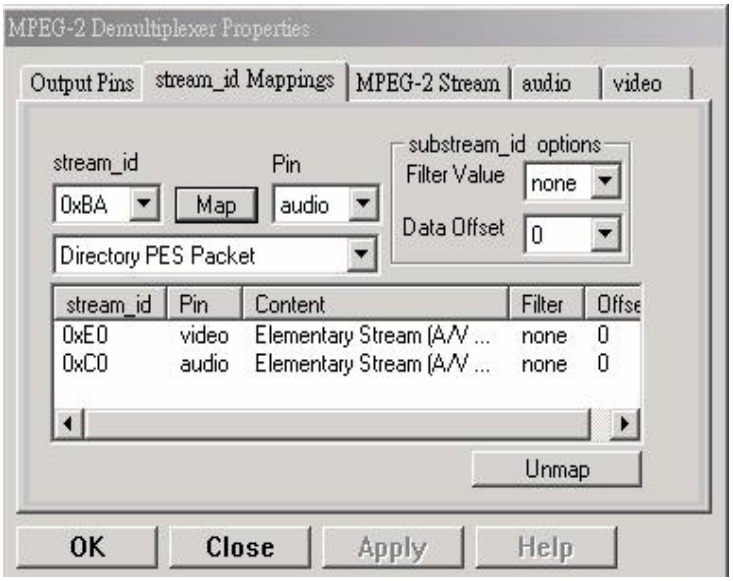

Fig 4-26 Mpeg-2 Demultiplexer Property Setting

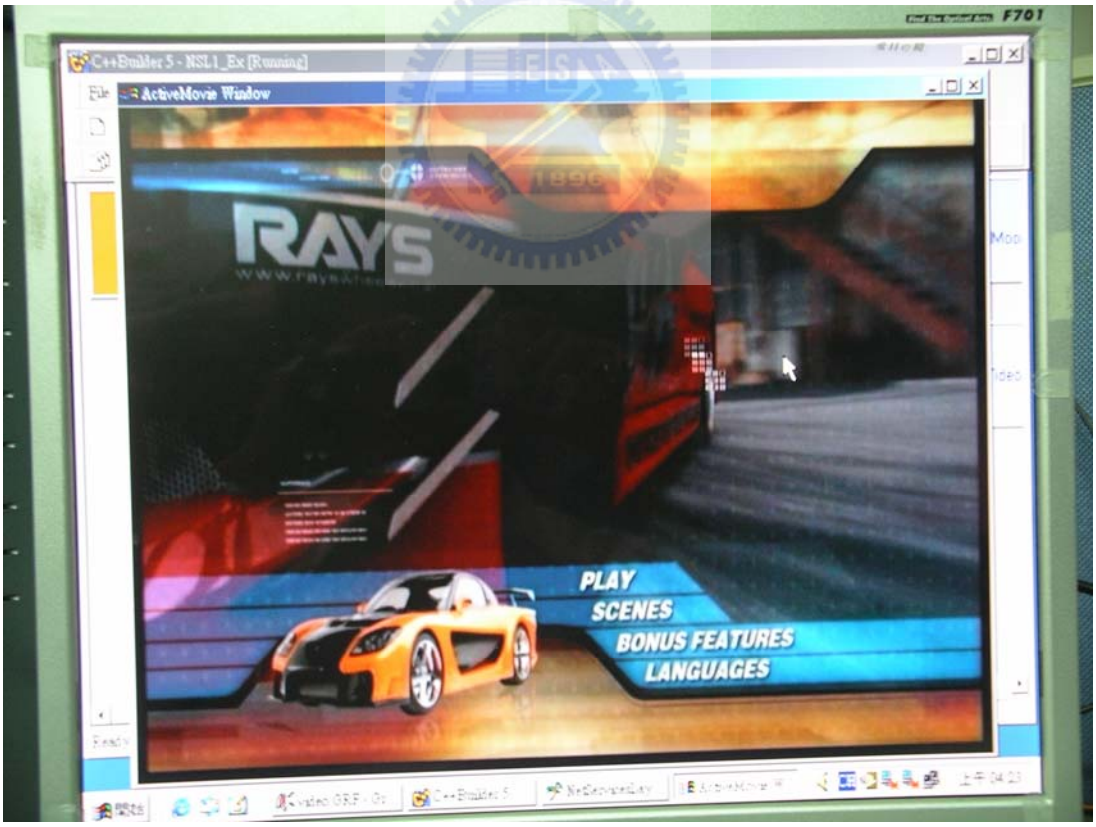

Fig 4-27 Picture of Mpeg-2 video

Chapter 5

Conclusion

The application of MOST technology in multimedia automobile network has been explored in this thesis. There are many benefits for adopting MOST technology instead of traditional vehicle bus system. Due to the high bandwidth requirement of the multimedia and integration of different protocols in car electronics, MOST defines a standard from the lowest hardware layer to the highest application layer to help car vendors to develop their own MOST devices. For speed consideration, MOST supports two physical transmission interfaces which are Plastic Optical Fiber (POF) and Unshielded Twisted Pair (UTP). The speed can reach 24.8Mbps with POF and 50Mbps with UTP. MOST also has flexibility in network topologies. It supports direct point-to-point, token ring, star, rings incorporating splitters connections. There can have up to sixty-four devices on each MOST network. This greatly reduces the connection overhead. In each MOST network, there is a timing master which generates a synchronous bit stream to provide the synchronization of the other timing slaves. To avoid the transmission overhead, MOST supplies resource share service. Every node on the MOST network can simultaneously share resource from the single source. For example, a CD player can send the audio stream to two or more amplifiers to play the sound at the same time. MOST also supports Plug and Play, this provides the convenience of expansion and maintenance. MOST system supports a variety of data types such as control data, packet data and synchronous stream data to accommodate different transmission demands. The control data contains the control message which often indicates some operation of controlled device. The packet data contains the non-real-time information. The synchronous stream data is available for the real-time information like the audio or video stream. The asynchronous data and

synchronous data share the same bandwidth. The arrangement of bandwidth is controlled by the timing master. In the application level, MOST defines Function Blocks (FBlock) and Function (FktID)…etc. Function is used to control the device or access the property of the device. A function block contains a set of functions. A MOST device may contain several function blocks. Every device contains a mandatory function block called NetBlock which contains the functions that affects the entire device. For communication between devices, MOST defines MOST protocol. By MOST protocol, every device can exchange messages and data via the network services provided by the network layer. MOST defines several network services for management of the entire MOST network and data transmission. The MOST Supervisor Service (MSV) is used to manage the states of MOST device. The Control Message Service (CMS), Application Message Service (AMS), and Remote Control Service (RCS) are used to handle the control messages. The Synchronous Channel Allocation Service (SCS) and Transparent Channel Allocation Service (TCS) can manage the routing path of stream data. The Asynchronous Data Transmission Service (ADS) supports the transmission of packet data. The Transceiver Control Service provides the access the low level MOST Transceiver.

The application of MOST technology in this thesis uses only one part of MOST NetServices API. For other similar applications, we can extend our program by adding the PDA or the Bluetooth device like Wireless Mobile Hand Free Car Kit and applies it in the automobile environment.

REFERENCES

[1] Website of MOST Cooperation: http://www.mostcooperation.com/technology/index.php

[2] 郭長祐, "車用電子之多媒體傳控網路：MOST 技術," April 04, 2004 Website: http://tech.digitimes.com.tw/ShowNews.aspx?zCatId=135&zNotesDocId=682AC814E4CE9A048257156005C846C

[3] MOST Cooperation, "MOST Specification Framework Rev. 1.1," 1999.

[4] MOST Cooperation, "MOST Specification Rev 2.2," Nov. 2002.

[5] MOST Cooperation, "MOST NetServices: Application Socket, User Manual and Specification Rev1.10," Jan. 23, 2004.

[6] MOST Cooperation, "MOST Function Catalog, V2.0," Oct. 2000.

[7] MOST Cooperation, "OS 8104 MOST Network Transceiver, Final Product Data Sheet," Sep. 2006

[8] Oasis SiliconSystems, "MOST NetServices Licensing Policy," May 17, 2005. http://www.smsc-ais.com/files/mostnetservices/PFL-MOSTNetServicesLicensing_V01_00_XX-1.pdf

[9] Oasis SiliconSystems, "MOST NetServices Layer 1 User Manual/Specification Rev. 1.10.x," Jan 23, 2004.

[10] Oasis SiliconSytems, "DVDPlayer 4 MOST User Manual, V1.2.0," Oct 13, 2003.

[11] Oasis SiliconSystems, "RadioTuner 4 MOST User Manual, V1.0.0," August 22, 2003.

[12] Oasis SiliconSystems, "Amplifier Controller 4 MOST Advanced Product Data Sheet," Jun. 2003.

[13] Microsoft, "DirectX 9.0 Documentation for C++," 2002.

[14] ISO-13818, “Information Technology - generating coding of moving pictures and associated audio information - Systems, Second edition,” Dec. 1, 2000.

[15] Dr. Gorry’s web site: http://erg.abdn.ac.uk/research/future-net/digital-video/mpeg2-trans.html.