ICE iPush® Communication Server

Embedded

IOModule Programming Guide

For WinCon-8000

By: ICE Technology Corp., Feb 11, 2004

Ver. 1.3.3

E-Mail \ service@icetechnology.com Tel \ +886-2-23961880 Fax \ +886-2-23961881

Copyright © 2004 ICE Technology Corporation. All Rights Reserved.

iPush Server is the registered trademark of ICE Technology Corporation.

Content

iPush® Embedded System Architecture

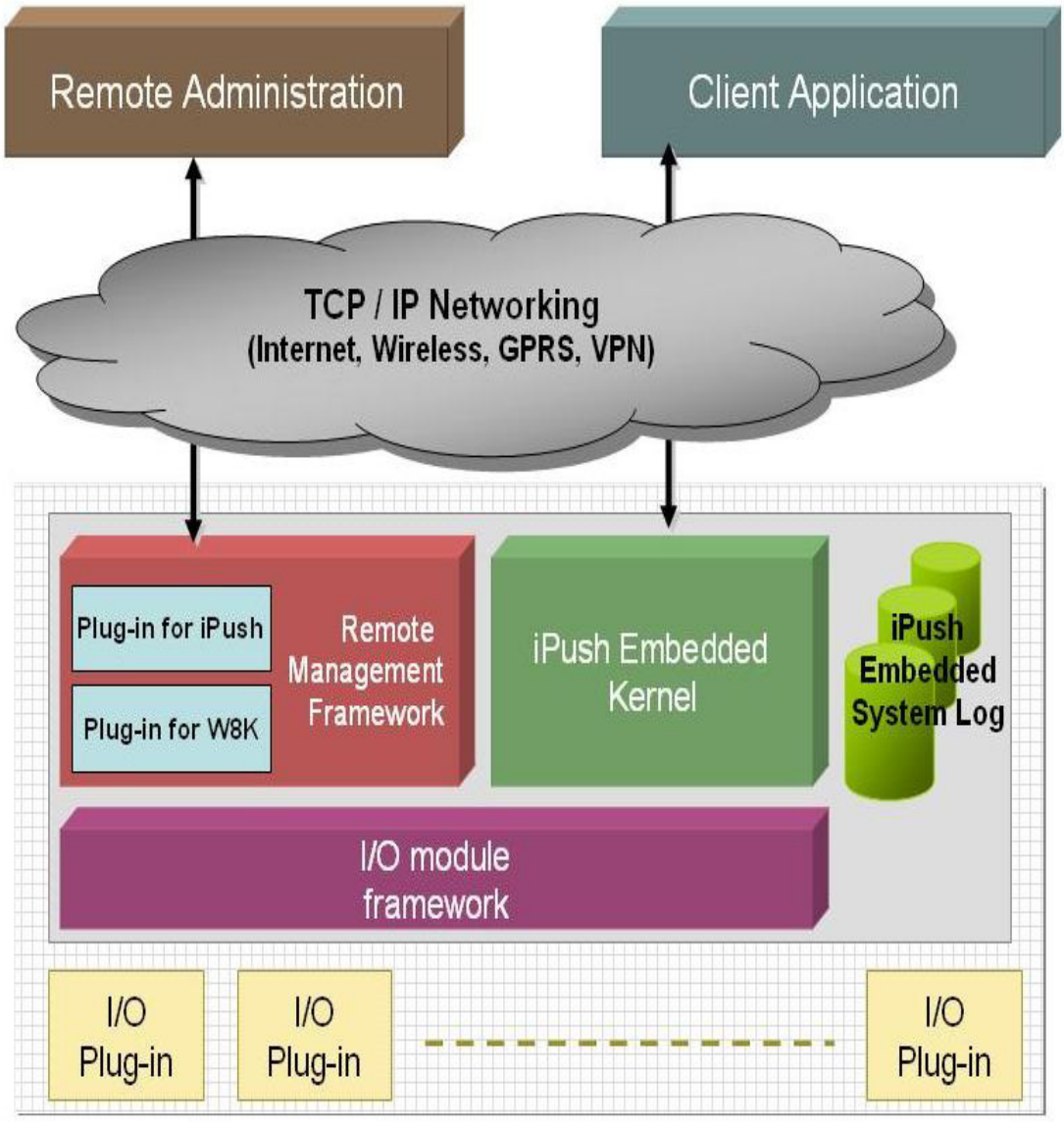

Figure 1. iPush® Embedded System for WinCon-8000 Architecture

Chapter 1 Introduction

IOModule was executed as Service on Windows CE Platform. Under proper configuration, Windows CE automatic loading iPush Embedded to memory and executing program. IOModule will make routine check all modules on WinCon-8000, and automatic loading or removing corresponding Module program.

IOModule can automatic up-link iPush Server at startup, and make data exchange. Even the connection was broken under abnormal situation, IOModule can rebuild connection automatically.

ICE Technology Provides IOModule **Main Program**, **Control Module**, and Basic **Device Modules** for Customer use. Following are **Control Module**, **Device Modules**, **Tag Container, Tag Object** programming guide to help System Developer to create new Device modules for iPush Communication Server Embedded.

PS: When using iPush Embedded, Please **UPDATE BOTH** iPush Server Embedded version and Client side component (Including iPushX.ocx and iceHMsg.ocx) to newest version same time, in order to make sure system work correctly.

Chapter 2 System Architecture

IOModule can be divided into 3 Parts: **Main Program**, **Control Module** and **Device Modules**.

■ Main Program

Main Program is a Dynamic Linking Library (DLL) compatible with Windows CE Service API. After Main Program loaded and Startup, Main Program will automatically load up Control Module. Then depends on Control Module request, Main Program will load or remove Device Modules. Main Program will handle every module's publish request, then package messages into iPush Message format. If Messages come from iPush Embedded, Main Program will pack messages into message objects first; In order to deliver data to corresponding Device Modules.

■ Control Module

Control Module is a hardware specific (for example: WinCon-8000) design module. Control Module has it own thread and will make routine check for hardware device configuration modification. If hardware device configuration changes, Control Module will acknowledge Main Program to add or remove corresponding Device Module.

Tips: Control Module has it own Subject string command structure. Please refer to later chapter for Detail. User can execute remote administration command by needs, for example: remote stop or start device.

■ Device Module

Device Module is a program module corresponding hardware I/O module (like i8056...etc), it's has his own thread, can routine grab information from hardware I/O modules. Then through Main Program, passing message to iPush Embedded. Device Modules also can take information from Main Program, executing remote administration work (for example: Do_8 function). One Device Module can handle multiple devices. Even at none device situation, Device Module still can function normally. Device Module design is flexible and powerful.

Chapter 3 System Configuration and Startup

■ The Files need to Start iPush Embedded

Module Name	Function
IOModule.dll	Main Program
ConnModuleCE.dll	Connection module for Main Program to iPush Embedded
IOMConfig.xml	Main Program Configuration File
ScanKernelController.dll	Control Module Corresponding WinCon8000 ScanKernel Edition
DeviceInfo.xml	Configuration File for IOMWC8k.dll
ScanKernelModule.dll	Device Module Corresponding WinCon8000 ScanKernel Edition
i8040.xml	i8040 Configuration File (naming convention fits other files)

■ Detail about Installation and Startup, Please refer to iPush Embedded Installation Guide.

Chapter 4 Main Program

ICE Technology did NOT release Main Program design documents, if you have Main Program related programming problem, please contact ICE Technology Service Department at service@icetechnology.com

- The Configuration file used by Main Program, file name is "IOMConfig.xml", puts under directory "\Compact Flash\IceTechnology\Config". The syntax needed to qualify standard XML syntax and file format. When Editing those configuration .xml file, please use standard XML Editor, in order to make sure correct editing result. Wrong .xml format will result unexpected error.

- Following **IOMConfig.xml** file, designated Log File Path, The information needed to Uplink and Log in iPush Server, and subscription setting , Control Module Path and subscription information.

```
<?xml version="1.0" encoding="utf-8" ?>
<root>
<Properties name="IOModule">
<SystemName>Chobits</SystemName>
<LogfilePath>\Compact Flash\IceTechnology\Logs\IOMLog</LogfilePath>
</Properties>
<Properties name="Uplink">
<LoginInfo>WC8K ICE IOM iomeci</LoginInfo>
</Properties>
<Properties name="ControlModule">
<ControlModulePath>\CompactFlash\IceTechnology\IOModule\ScanKernelC
```

ontroller.dll</ControlModulePath>

<Subject>Controller</Subject>

</Properties>

</root>

- The "SystemName" item under IOModule property designated this Machine's name (here is "Chobits"). Then, IOModule will subscribe subject start with WC8K.SystemName. In the Example here is "WC8K.Chobits".

- The "LogfilePath" item under IOModule property designated Log's file name and store path. In this Example, Log will puts under \Compact Flash\IceTechnology\Logs. And, the file name will be "IOMLogDATE.txt". DATE will be replaced with today's date, IOModule will generate different logs everyday.

- The "LoginInfo" tag of "Uplink" item, designated IOModule's login name for upper iPush Server. In this Example, Login information is: Company="WC8K", Product="ICE", Username="IOM", Password="iomeci".

- The "CommonModulePath" tag under CommonModule item designated ControlModule's physical path.

- The "Subject" tag under CommonModule (ControlModule) item designated Subject Postfix that ControlModule will subscribe. In this Example, ControlModule will subscribe "WC8K.Chobits.Controller". Any message start with "WC8K.Chobits.Controller" will be processed by "ControlModule".

In this case, any subject string named start with "WC8K.Chobits" will acknowledge every module to process, including "ControlModule" and "DeviceModule". And subject string start with "WC8K.Chobits.Controller" will only be processed by ControlModule.

Chapter 5 Control Module

Control Module is a Windows standard Dynamic Linking Library (DLL), which will be loaded by IOModule Main Program. Control Module will make decision about which plug-ins should be loaded by IOModule Main Program.

5.1. Plug-in APIs

There are 6 functions that a plug-in should extend. These functions are list as fellow:

5.1.1. Init

extern "C" __declspec(dllexport) int (__stdcall Init)(int nID, CBPublish procCBPublish, CBDeviceJoin procCBDeviceJoin, CBDeviceLeave procCBDeviceLeave, CBWriteLog procWriteLog);

Init function will be called by main program when the IOModule services were initialed. Plug-in should do some init actions, like memory allocating and objects initialing, then return result.

- **nID**
  This parameter is a ID number provided by main program to indicate the difference between plug-ins. Plug-in should remember this parameter, and uses it on some call back functions.
- **procCBPublish**
  Main program provide a CALL BACK function as this parameter, and Control module can use it to publish a message to iPush server. Let's discuss this call back function later in chapter 5.2.
- **procCBDeviceJoin**
  Control module calls this function to command main program to load a specified plug-in module. Let's discuss this call back function later in chapter 5.2.

- **procCBDeviceLeave**
  Control module calls this function to command main program to unload a specified loaded plugin module. Let's discuss this call back function later in chapter 5.2.

- **procWriteLog**
  Main program provide a CALL BACK function as this parameter, and Control module can use it to write a Log message. Let's discuss this call back function in chapter 5.2.

5.1.2. Finalize

extern "C" __declspec(dllexport) void (__stdcall Finalize)();

Finalize function will be called by main program when the IOModule service is going to be unloaded. Contorl module should do resource free actions after this function called.

5.1.3. Start

extern "C" __declspec(dllexport) int (__stdcall Start)();

IOModule Main program will call this function when service start. Control Module should active its threads to work.

5.1.4. Stop

extern "C" __declspec(dllexport) void (__stdcall Stop)();

Main Program of IOModule framework will call this function when service stopped. Control module should hold all threads but don't have to free its resource from system.
IOModule might be unloaded immediately, and Finalize will be called without Stop in this sititution.

5.1.5. Refresh

extern "C" __declspec(dllexport) void (__stdcall Refresh)();

This function will be called by the Main Program of IOModule framework when the IOModule services commanded to be refreshed. This could be happened when a user

uses services refresh command in command line. Or, a user changes IOModule configuration in Remote Management Client from a remote side.

Control Module program should check system new configurations here, and do following actions.

5.1.6. ProcessMsg

extern "C" __declspec(dllexport) void (__stdcall ProcessMsg)(char *strTagPath, MessageObject *lpMsgObj);

When a message is send to IOModule from iPush Embedded, Main Program of IOModule framework will check its destination and then pass this message to the corresponding plug-in with **MessageObject** format.

- **strTagPath**

This parameter indicates the remand destination of the message. For example, if this Control module's subject prefix is **control**, and the destination of the message is **Service.ServerName.Control**. Action, then the **strTagPath** would be Action.

Control Module can check this parameter to indicate which sub-function should process this message.

- **lpMsgObj**

Control Module always put its message into a **MessageObject** data object, after that, publish to iPush Embedded, and when IOModule receives a message, it will always transform the message into a **MessageObject** format for Control Module. We will discuss the **MessageObject** in chapter 5.3.

5.2. Call Back Functions

There were 4 call back functions that Main Program of IOModule framework provided for Control Module. Control Module can get these functions from **Init** parameters.

5.2.1. CBPublish

typedef int (__stdcall *CBPublish)(int nID, int nType, char *strGroupPath, MessageObject *lpMsgObj);

Control Module uses this function to publish a message to iPush Embedded.

- **nID**

Plug-in ID number of the Control Module. Control Module should put this parameter with the **nID** into **Init** function. Some wrong results might occur if Control Module put wrong **nID** in **CBPublish**.

- **nType**

Means Which type of this message would be. IOModule provides subject mode only and this parameter should be MESSAGE_TYPE_SUBJECT.

- **strGroupPath**

This parameter indicates the postfix of the destination of the message. For example, if you fit **Function.Tag** in this parameter, IOModule could complete the full destination of this message as **Service.ServerName.ModuleName.Function.Tag**. If this parameter is **NULL** or empty string, the full destination would be **Service.ServerName.ModuleName**

- **lpMsgObj**

Control Module will package the data for which will be published into an **MessageObject**, then. Then, **MessageObject** will be put into this parameter then published. We will

discuss **MessageObject** object in Chapter 5.3.

5.2.2. CBDeviceJoin

typedef int (__stdcall *CBDeviceJoin)(char *strModuleFilepath, char *strModulePrefix, DWORD dwParam);

Control Module calls this function to command Main Program to load a specified plug-in.

- **strModuleFilepath**

The full file path of the plug-in DLL which Control Module commanded the Main Program to load from.

- **strModulePrefix**

The prefix of the indicated plug-in. For example, **MyPlugin** then the full subject prefix of the plug-in would be **Service.ServerName.MyPlugin**

- **dwParam**

Control Module uses this parameter to pass a value into the plug-in. The plug-in will receive this value from its **Init** function. This parameter is a double word, which can present an integer, a float or a pointer.

- **Return value**

The return value of this CALL BACK function indicates the **nID** of the plug-in. Return –1 means plug-in loading failure.

5.2.3. CBDeviceLeave

typedef void (__stdcall *CBDeviceLeave)(int nID);

Control module calls this function to command main program to unload a specified plug-in.

- **nID**

The plug-in ID number that returned by **CBDeviceJoin**.

5.2.4. CBWriteLog

typedef void (__stdcall *CBWriteLog)(int nID, char *strLog);

Control Module can call this call back function to write a log message into log file.

- **nID**

Plug-in ID number of this Control Module, Control Module should fit this parameter with the **nID** in Init function. Some wrong results could occurred if plug-ins fit wrong **nID** in **CBPublish**.

- **strLog**

A string with NULL ending that plug-ins wish to write into log file.

5.3. MessageObject Object

MessageObject is a Hash Table like object, that users can use **setXXXProperty** to put data into it and use **getXXXProperty** to get data from it easily.

When users would like to use **MessageObject** in their programs, they should include **Mesg.h** and link **MessageObject.lib**.

5.3.1. Put data into MessageObject

Before calls **CBPublish**, a plugin should put its data into a MessageObject object first. MessageObject object provides a group of functions for callers to put data into it.

- **setBooleanProperty**
- **setByteProperty**
- **setShortProperty**
- **setIntProperty**
- **setFloatProperty**
- **setDoubleProperty**
- **setStringProperty**
- **setText**

Those **setXXXProperty** functions, always have 2 parameters, first one is the name of the data and the second one is the value.

The **setText** function has only one parameter, which is a character pointer of a NULL ending string.

Users can call setXXXProperty many times to put some difference data into a **MessageObject**, but only one **setText** can be called.

5.3.2. Get data from MessageObject

After IOModule program received a message from iPush Embedded, it will pack the message into a MessageObject object. Then, Plug-ins can get the data from MessageObject.

- **getBooleanProperty**
- **getByteProperty**
- **getShortProperty**
- **getIntProperty**
- **getFloatProperty**
- **getDoubleProperty**
- **getStringProperty**
- **getText**

Those **getXXXProperty** functions have only one parameter, that is the name of the data, and functions return the value of the data.

Before users calling **getXXXProperty** functions, they should call **isPropertyExist** to check is the data exist in the **MessageObject**.
When calling **getText**, users will get the string pointer that **setText** put into it.

5.3.3. Clear data in MessageObject

Users can call **clearProperties** to clear all data within the object.

5.4 DeviceInfo.xml Configuration File

ICE Technology provides a Control Module (IOMWC8K.dll) corresponds to WinCon-8000 hardware. It use DeviceInfo.xml as a configuration file. The file format is XML. Suggest use standard XML Editor, to avoid data format Error.Format as fellow:

```
<?xml version="1.0" encoding="utf-8" ?>
- <root>
- <BootupDevice name="0">
  <Device>ScanKernal</Device>
  <ModulePath>\Compact
Flash\IceTechnology\IOModule\ScanKernelModule.dll</ModulePath>
  <Subject />
  </BootupDevice>
  </root>
```

■ BootupDevice

When WinCon-8000 Control Module start, Control Module need to read Bootup Device information. Then call CBDeviceJoin to request Main Program to load correspond Device Module. This Bootup Module was loaded by default, not because main program detect Bootup Module.

DeviceInfo.xml attribute **Name** designate loading sequence, start number is 0, maxium number is 9. So iPush Embedded could load up TEN BootupDevice. **ModulePath** item designate the loading Module Path. **Subject** item designate DeviceModule Subject Prefix.

■ ICE XML Property tool

ICE Technology provided a Library, made complex XML easy to use. For simple XML file like DeviceInfo.xml and IOModule.xml, using XML Property tools will making reading XML file just like to read simple Win32 system INI file.

This Function have 4 function Call.

```
int XMLGetIntProperty(char *strTitle, char *strName, char *strProperty, int
nDefault, char *strFilePath);

char* XMLGetStringProperty(char *strTitle, char *strName, char *strProperty, char
*strBuf, int nMaxSize, char *strDefault, char *strFilePath);

void XMLSetIntProperty(char *strTitle, char *strName, char *strProperty, int
nValue,char *strFilePath);
```

```
void XMLSetStringProperty(char *strTitle, char *strName, char *strProperty, char
*strValue, char *strFilePath);
```

For **XMLGetStringProperty** function, If We want to get device 8017's ID from DeviceInfo.xml, the following code should work.

```
char strResult[32];
XMLGetStringProperty("DeviceInfo", "8017", "ID", strResult, 31, "ERROR",
\\Windows\\DeviceInfo.xml);
```

If succeed, function will return string "17", or return default string "ERROR".
Please include < Win32XMLParser.h> header file, and link XMLParser.lib library, before using XML property tools.

Chapter 6 Device Modules

ICE Technology provides basic device module for ICPDAS 8017, 8024, 8040,8041,8053, 8056, 8077. And a flexible IOModule framework for Developer could write new device module based on special needs.

Device Module and Control Module are alike, Dynamic Linking Library, Call Back function, API are all similar.

6.1 CallBack Function for Device Module

Main Program Open 2 Call Back Function for Device Module.

typedef int (__stdcall *CBPublish)(int nID, int nType, char *strGroupPath, MessageObject *lpMsgObj);		
Explanation	Device Module call this Callback function provided by MainProgram for publish message to iPush Embedded.	
Parameter	nID	DeviceModule ID for Main Program
	nType	Message TYPE has 2 different choice, MESSAGE_TYPE_CHANNEL and MESSAGE_TYPE_SUBJECT. But IOModule only provide subject mode.
	strGroupPath	This Parameter designated Subject Postfix, if this command's parameter is "HelloWorld", than the Subject message destination is "WC8K.Chobits.8056_3.HelloWorld".
	lpMsgObj	Including the MessageObject want to send out message. Detail about Class MessageObject will be descript later.
Return Value	1 means success, 0 means fail.	

typedef void (__stdcall *CBWriteLog)(int nID, char *strLog);		
Explanation	Call MainProgram to write log record.	
Parameter	nID	DeviceModule ID
	strLog	The Log string want to be written, do not need ended with CR.

6.2 Function call that Device Module must extend

Device Module need provide (extend) 5 Function Call.

extern "C" __declspec(dllexport) int (__stdcall Init)(int nID, DWORD dwParam, CBPublish procCBPublish, CBWriteLog procWriteLog)		
Explanation	Called after MainProgram loaded DeviceModule. This function will do some initial work and check the environment for later execution. But won't execute device immediately.	
Parameter	nID	Designate DeviceModule ID This parameter will be used at CBPublish and CBWritelog later.
	dwParam	The parameter will be passed when CommonModule call procCBDeviceJoin.
	procCBPublish	CallBack Function provides by main program.
	procWriteLog	CallBack Function provides by MainProgram.
Return Value	Return 0 means Fail, other means success.	
extern "C" __declspec(dllexport) int (__stdcall Start)()		
Explanation	MainProgram call this function to acknowlwdge DeviceModule start service.	
Return Value	Return 0 means Fail, other means success.	

extern "C" __declspec(dllexport) void (__stdcall Stop)()		
Explanation	MainProgram call this function to acknowlwdge DeviceModule stop service.	
extern "C" __declspec(dllexport) void (__stdcall Finalize)()		
Explanation	MainProgram call this function to acknowledge DeviceModule will be unloaded. Then Control Module should release memory.	
extern "C" __declspec(dllexport) void (__stdcall Refresh)()		
Explanation	MainProgram will call this function to acknowledge DeviceModule re-read the configuration file. This function will be called when user use Remote Admin program to modify system parameter, or execute service refresh command under command console.	
extern "C" __declspec(dllexport) void (__stdcall ProcessMsg)(char *strTagPath, MessageObject *lpMsgObj)		
Explanation	This function will be called when Main Program received message from iPush Embedded, and decide to send this message to DeviceModule.	
Parameter	strTagPath	If Message Subject Destination is “WC8K.Chobits.8056_3.Superman.HelloWorld” And this parameter will be “Superman.HelloWorld”.
	lpMsgObj	MessageObject needed to be process.

- ICE Technology Provides a sample shell for DeviceModule programming. Developer could use this shell to write codes.
- iPush Embedded use MessageObject When exchange message with Main Program. Detail of MessageObject are discussed in later chapters.
- DeviceModule could recognized message from TagContainer and TagObjects provided by ICE Technology. About how to use these object, please refer to the later chapter.

Chapter 7 Message Object

All Publish, Process Message actions between **Device Modules**, **Control Module** and **Main Program** are using MessageObject. MessageObject is a simple and useful object used for message encapsulation in iPush Embedded.

Using MessageObject, Please including <Mesg.h> header file and link MessageObjects.lib Library file.

7.1 MessageObject Class Definition

Class definition as fellow:

```
class MessageObject {
private:
int msgID;
int priority;
int expire;
LPVOID htProp;
char* sText;
public:
MessageObject();
MessageObject(MessageObject &peer);
~MessageObject();
MessageObject &operator=(MessageObject &peer);

int getMessageID() { return msgID; }
void setMessageID(int id) { msgID = id; }
int getPriority() { return priority; }
void setPriority(int pri) { priority = pri; }
int getExpiration() { return expire; }
void setExpiration(int exp) { expire = exp; }
char* getText() { return sText; }
void setText(char *str);
```

```
// Properties
BOOL isPropertyExist(char *name);
void clearProperties();
BOOL propertyExists(char *name);
BOOL getBooleanProperty(char *name);
BYTE getByteProperty(char *name);
short getShortProperty(char *name);
int getIntProperty(char *name);
float getFloatProperty(char *name);
double getDoubleProperty(char *name);
char* getStringProperty(char *name);
void setBooleanProperty(char *name, BOOL value);
void setByteProperty(char *name, BYTE value);
void setShortProperty(char *name, short value);
void setIntProperty(char *name, int value);
void setFloatProperty(char *name, float value);
void setDoubleProperty(char *name, double value);
void setStringProperty(char *name, char *value);

LPVOID getProperty() { return htProp; }
MessageObject* dup();
};
```

7.2 MessageObject Functions

- You can use **setXXXProperty()** system function put message into message object. And call **getXXXProperty()** to get message from message object. Works just like a hash.

- When device want to publish data, do following procedure:

```
MessageObject MO;
MO.setIntProperty("DigitalInput", nValue);
MO.setBooleanProperty("BitInput", bResult);
ProcCBPublish(nMyIndex, MESSAGE_TYPE_SUBJECT, NULL, &MO);
```

- When device module received **ProcessMessage()** function from Main Program. Just use getXXXXProperty() to extract message.

Tips: If a message is for all Device Module, than every Device Module will process the same MessageObject instance. Recommend not to modify any MessageObject instance received from ProcessMessage.

Chapter 8 TagContainer Object

ICE Technology provides a TagContainer Class. This object could read Tag definition XML files, and manage all Tag Objects. It also capable to read Properties from XML definition files.

8.1 TagContainer Class Definition

```
class TagContainer {
public:
        TagContainer();
        virtual ~TagContainer();
        static char* AttrGetValue(const char **atts, char *strTarget);
      BOOL LoadProfile(int nMySlot, char *strProfilename, PropertyHandler
procSetProperty, TagCreateHandler procCreateTag);
      void AddTagObject(TagObject *lpTagObj);
        BOOL IsTagExist(char *strTagName);
        void RemoveTag(char *strTagName);
        void RemoveGroup(char *strGroupPath);
        virtual void SetState(LPVOID lpState);
        virtual BOOL GetMessageObject(char *strGroupPath, MessageObject
*lpMsgObj);
        virtual void ProcessMessage(char *strGroupPath, int nSlot, MessageObject
*lpMsgObj);
        int m_nParseState;
        int m_nMySlot;
        char m_strCurrentPropertyName[256];
        char m_strTagName[256];
        TagCreateHandler m_procTagCreatHandler;
        PropertyHandler m_procPropertyHandler;
private:
        BOOL ParseXMLProfile(char *strFilePath);
        CArray<TagObject*, TagObject*> m_arObjects;
};
```

- **Before using TagContainer, please including <TagObject.h> header file and link TagObjects.lib library files.**

8.2 Tag Configuration XML File

Tag configuration xml file format definition as fellow:

```
<?xml version="1.0" encoding="utf-8" ?>
<root>
<Slot slot="1">
    <Properties>
        <Frequency>100</Frequency>
        <DiffVoltage>0.3</DiffVoltage>
        <DiffCurrent>0.4</DiffCurrent>
    </Properties>
    <Tag name="Analog0" type="AnalogInput" channel="0">Description</Tag>
    <Tag name="Analog1" type="AnalogInput" channel="1">Description</Tag>
    <Tag name="Analog2" type="AnalogInput" channel="2">Description</Tag>
    <Tag name="Analog3" type="AnalogInput" channel="3">Description</Tag>
    <Tag name="Analog4" type="AnalogInput" channel="4">Description</Tag>
    <Tag name="Analog5" type="AnalogInput" channel="5">Description</Tag>
    <Tag name="Analog6" type="AnalogInput" channel="6">Description</Tag>
    <Tag name="Analog7" type="AnalogInput" channel="7">Description</Tag>
</Slot>
<Slot slot="2">
    <Properties>
        <Frequency>200</Frequency>
        <DiffVoltage>0.3</DiffVoltage>
        <DiffCurrent>0.4</DiffCurrent>
    </Properties>
    <Tag name="Analog0" type="AnalogInput" channel="0">Description</Tag>
    <Tag name="Analog1" type="AnalogInput" channel="1">Description</Tag>
    <Tag name="Analog2" type="AnalogInput" channel="2">Description</Tag>
    <Tag name="Analog3" type="AnalogInput" channel="3">Description</Tag>
    <Tag name="Analog4" type="AnalogInput" channel="4">Description</Tag>
    <Tag name="Analog5" type="AnalogInput" channel="5">Description</Tag>
    <Tag name="Analog6" type="AnalogInput" channel="6">Description</Tag>
```

```
    <Tag name="Analog7" type="AnalogInput" channel="7">Description</Tag>
</Slot>
</root>
```

- **Slot**

Device could insert into different slot and could have different behavior. Every slot has it's own property when device inserted. When Device Module management device at slot 3, only root information with slot=3 will be loaded.

- **Properties**

Properties record every DeviceModule custom properties, for example: Frequency controls reading frequency.

- **Group**

Group based on Tag teaming, Group could also include other group, no tree level limits.

- **Tag**

Tag could belong under slot, but also included within group. Tag field value is Tag Description, not really used by DeviceModule. And Tag property (name, type, channel) will be use for identify and generated corresponding Tag Object.

8.3 TagContainer Functions

BOOL LoadProfile(int nMySlot, char *strProfilename, PropertyHandler procSetProperty, TagCreateHandler procCreateTag);		
Explanation	Designated a Tag Configuration file for TagContainer to load up, TagContainer will use these information to build up whole Tag object tree system.	
Parameter	nMySlot	Designate slot number will be process now, only information with correspond slot will be loaded.

	procSetProperty	**typedef void (*PropertyHandler)(char *strPropertyName, char *strPropertyValue);** LoadProfile must provide two CallBack functions for TagContainer. TagContainer will read configuration file first, after property read, system will call **procSetProperty** function and passed in **PropertyName** and **PropertyValue** as parameter into function. Your program could execute necessary configuration here.
	procCreateTag	**typedef TagObject*** **(*TagCreateHandler)(const char **atts);** When TagContainer read a Tag data item, will call **procCrateTag**, and passed in correspond attributes as parameter. This time **TagCreateHandler** should generated correspond Tag Object as return value. If return **NULL** means did not corresponding Tag Objects. About Tag Objects description detailed on later chapter.
static char* AttrGetValue (const char **atts, char *strTarget);		
Explanation	**TagCreatHandler**'s parameter **atts** is a 2 dimensional string array. Start from zero, and two attribute team up as a group. If you think it is too much work for analysis **atts** structure. You can call **AttrGetValue** and passed in atts, then query **AttributeName**, **attrGetValue** will return **AttributeName**.	
Parameter	atts	Parameter introduce into **TagCreatHandler**.
	strTarget	The **AttributeName** want to query.
Return Value	Corresponds AttributeValue	
virtual void SetState (LPVOID lpState);		
Explanation	Call **SetState** and passed in a value, TagContainer will use this value as parameter, call all TagObject it within SetState. You can use this call to acknowledge the state change of TagObject, or let TagObjects' pointer point to varible.	

Parameter	lpState	Want to acknowledge TagObjects' value, any value type is acceptable, but must be recognized by Tag Objects.
virtual BOOL GetMessageObject (char *strGroupPath, MessageObject *lpMsgObj);		
Explanation	Query all TagObject, is any data want to send out. After call GetMessageObject, TagContainer will call every TagObject's GetMessageObject method within it's control region. If any TagObject GetMessageObject return **TRUE**, and TagContainer return **TRUE**. You can continuous call GetMessageObject on a WHILE loop, in order to get all MessageObject.	
Parameter	strGroupPath	If any TagObject Return **TRUE**, This parameter will get TagObject's GroupPath.
	lpMsgObj	If any TagObject Return **TRUE**, This parameter will get MessageObject from which TagObject will send out.
Return Value	**TRUE** means there will have TagObject to send out. **FALSE** means there will be no TagObject to send out.	
virtual void ProcessMessage (char *strGroupPath, int nSlot, MessageObject *lpMsgObj);		
Explanation	Call TagObjects within same region to handle MessageObject.	
Parameter	strGroupPath	When DeviceModule's ProcessMessage is called, Message GroupPath and it's parametet will passed in. Only TagObject with identical GroupPath will handle MessageObject.
	nSlot	Passed in Device Module Slot, some TagObject will need this parameter.
	lpMsgObj	MessageObject needed to be process.
BOOL IsTagExist(char *strTagName);		
Explanation	Is Designate Tag within it's region.	

Parameter	strTagName	Designate Tag Name, including GroupPath. For example: "SuperGroup.SubGroup.TagA"
Return Value	If exist return TRUE, others return FALSE.	
void AddTagObject(TagObject *lpTagObj);		
Explanation	Add a TagObject into TagContainer.	
Parameter	lpTagObj	If want to add a TagObject into container, the m_strName must have complete path name, for example "SuperGroup.SubGroup.TagA"
void RemoveTag(char *strTagName);		
Explanation	Remove TagObject.	
Parameter	strTagName	Designate the TagObject Name that want to remove.
void RemoveGroup(char *strGroupPath);		
Explanation	Remove all TagObjects under GroupPath.	
Parameter	strGroupPath	Designate remove GroupPath.

8.4 Example of TagCratehandler

```
TagObject* MyTagCreateHandler(const char **atts)
{
    char *strName=TagContainer::AttrGetValue(atts, "name");
    char *strType=TagContainer::AttrGetValue(atts, "type");
    char *strChannel=TagContainer::AttrGetValue(atts, "channel");
    if(strName && strType && strChannel)
    {
        if(strcmp(strType, "DigitalInput")==0)
        {
            TagDigitalInput *lpDigitalTag=new TagDigitalInput;
            lpDigitalTag->SetName(strName);
            return lpDigitalTag;
        }
        else if(strcmp(strType, "BitInput")==0)
        {
            TagBitInput *lpBitTag=new TagBitInput(atoi(strChannel));
            lpBitTag->SetName(strName);
            return lpBitTag;
        }
        else
        {
            char strLog[512];
            sprintf(strLog, "Unknown Tag type=%s\r\n", strType);
            g_procCBWriteLog(g_nMySlot, strLog);
        }
    }
    else
        g_procCBWriteLog(g_nMySlot, "Error Tag profile data.\r\n");
    return NULL;
}
```

The TagObject return by TagCreateHandler, must be a TagObject or inheritance class from TagObject. Parameter is a array made by **atts** string array, and have to use **NULL** at end of array. TagContainer provided a static **AttrGetValue** to read this parameter.

8.5 Example of PropertyHandler

```
typedef void (*PropertyHandler)(char *strPropertyName, char *strPropertyValue);
{
  if(strcmp(strPropertyName, "Frequency")==0)
  {
        g_nCheckFrequency=atoi(strPropertyValue);
        char strLog[512];
        sprintf(strLog, "Change Frequency property value=%d\r\n",
g_nCheckFrequency);
        g_procCBWriteLog(g_nMySlot, strLog);
  }
}
```

8.6 Example of SetState and GetMessageObject

Device Module can use it's own thread to read newest device state. Then passed in the state to every Tag Object through TagContainer's SetState. Then call GetMessageObject to test every TagObject is still have message wait to be sent. Here is The Example:

```
void CheckDevice()
{
      unsigned long ulValue=(unsigned long)~DI_8(g_nMySlot);
      ulValue&=0x000000FF;
      if(ulValue != g_ulLastValue)
      {
            g_lpTagContainer->SetState(&ulValue);
            MessageObject MO;
            char strPostfix[256];
            while(TRUE)
            {
                  if(g_lpTagContainer->GetMessageObject(strPostfix, &MO))
                        g_procCBPublish(g_nMyIndex,
MESSAGE_TYPE_SUBJECT, strPostfix, &MO);
                  else
                        break;
            }
```

```
            g_ulLastValue=ulValue;
        }
    }
```

- **Tag's Postfix**

  If a Tag directly belong to a slot, then the **TagPostfix** is **TagName**. If a Tag belong to SuperGroup's SubGroup, and the **TagPostfix** will be SuperGroup.SubGroup.TagName.

  After called **TagContainer::GetMessageobject()**, if any Tag need to transfer message, then the **TagObject** will fill data into the **Messaging Object**. Then put **TagPostfix** into **strPostfix**, then return result.

  When **TagContainer** call **procCBPublic** to send this message, last information will be send to destination **ICE.WC8K.8056_6.SuperGroup.SubGroup.TagName**

- **Process Message**

  When Device Module received ProcessMessage from Main Program. Device Module can choice either handle this message, or call TagContainer ProcessMessage function, Let ProcessMessage function to handle MessageObject to correspond TagObject.

```
void DoProcessMsg(char *strTagPath, MessageObject *lpMsgObj)
{
  g_lpTagContainer->ProcessMessage(strTagPath, g_nMySlot,
lpMsgObj);
}
```

  Any TagObject's TagPostfix identical to sub-group of strTagPath, the TagObject will be called to handle that MessageObject. If strTagPath="SuperGroup.SubGroup",Then TagObject with TagPostfix "SuperGroup.SubGroup.TagName" will be called.

  If Device Module designed **NOT** to use TagContainer and TagObjects, Device Module will have to handle message transfer and receive process by

itself.

Chapter 9 Tag Objects

One Tag Object is corresponding to a Tag, and a Tag could be assign to a group, or directly under slot.

9.1 TagObjects

ICE Technology provides common TagObjects, like: DigitalInput, DigitalOutput, BitInput, BitOutput, AnalogInput and AnalogOutPut. Developer can direct use or inheritance TagObject, to design the exact Tags you needed.

```
class TagObject {
// Construction
public:
TagObject();
virtual ~TagObject();

char* GetName();
void SetName(char *strName);
virtual BOOL IsMatched(char *strTagName);
virtual void SetState(LPVOID lpState);
virtual BOOL GetMessageObject(char *strPath, MessageObject *lpMsgObj);
virtual void ProcessMessage(int nSlot, MessageObject *lpMsgObj);
protected:
char m_strName[256];
int m_nNameLen;
BOOL m_bStateModified;
};
```

When design your own TagObject. Overloading **SetState**, **GetMessageObject**, **ProcessMessage** function after inheritance.

- **IsMatched**

All other function will call this function, to decide strTagName is including this TagObject's

Postfix, return TRUE if postfix is identical.
Developer **DO NOT** have to overloading this function, and let TagObject make judgment by system default rules. If any TagObject want to handle all events, just let this function return TRUE.

It is risky to let one object handle all events. For example, if object call RemoveTag, this TagObject will be removed, too.

9.2 SetState

When **TagContainer::SetState()** is called, with set all TagObjects state within TagContainer by sequence.The parameter is non-specify type pointer, so Device Module and TagObjects have to define the parameter means.

For a Input Type Tag, this parameter could be newest state value. TagObject will compare this value with least value, to verify state changed or not.

For a Output Type TagObject, this value better means a pointer. If Tag changes Device state, update the pointer content. After, other TagObjects can get the newest state.

9.3 GetMessageObject

Device Module will call TagContainer's GetMessageObject method in a WHILE loop. Everytime when **TageContainer::GetMessageObject()** is called, every **TagObject::GetMessageObject** will also be called by sequence. If TagObject GetMessageObject return **TRUE**, means: This TagObject itself has data to send. Before return data, TagObject have to fill in the MessageObject content, then return **TRUE**.

If **FALSE** is returned, means no data need to transfer, the next TagObject's **GetMessageObject()** will be called.

After all TagObject return FALSE, the WHILE loop for **TagContainer::GetMessageObject()** should be finished, means no more data will be sent.

CAUTION: If any TagObject design defect, let GetMessageObject return TRUE consistently, the system will into infinite loop and send message out.

9.4 Process Message

When Device Module called **TagContainer::ProcessMessage()**, all TagObject with same strPostfix will be called. If TagObject don't inheritance this function, means no message will be processed.

After TagObject created, first call **SetName()** function to set it's name. For Example: SetName("MyName"). After **TagCreatHandler** return, TagContainer will add **GroupPath** right before it's name. Then when **GetName()** function called, string maybe looks like: **SuperGroup.SubGroup.MyName**

- **Using TagObject, all have to do is including <Mesg.h> header file and link TagObjects.lib.**

Ch10 Hierarchy Subject Naming Rule

Within IOModule system, **iPush Embedded**, **Main Program**, **Control Module**, **Tag Object** all fellow same naming rule for data transmission. Therefore, you have to understand the Subject naming rule.

10.1 Messages Subject Destination

Every message flow in the iPush network had a Subject Destination, looks like WC8K.Chobits.8056_6.SuperGroup.TagName.

- **WC8K** : The name of device model
- **Chobits**: Current system Name
- **8056_6**: Means model number 8056 at slot number 6. The i8056.xml configuration file with slot = 6 setting will apply to this module.
- **SuperGroup**: Tag group, Tag within same group will receive Information same time.
- **TagName**: The name of individual Tags.

Subject Destination's naming rule is: Top of level is fixed **WC8K**, fellow with **SystemName** (Defined in **IOMConfig.xml**), then **DeviceName**, End with **TagGroupPath** and **Tag Name**.

10.2 Subscription

IOModule will subscribe a subject from iPush Embedded, for Example: WC8K.Chobits. Any WC8K.Chobits and the sub-subject message will received by **IOModule**. Including WC8K.Chobits, WC8K.Chobits.8056_6, WC8K.Chobits.Controller, WC8K.Chobits.8053_7.SuperGroup..,etc. WC8K.Chobits will call correspond Modules to handle the Events.

WC8K.Chobits.Controller message will pass to **Control Module** to handle it.

Message for WC8K.Chobits.8056_6 will transfer for 8056 Module at slot 6. If this Module use **TagContainer**, all **TagObject** with same **Tag Container** will receive the message.

Tags under SuperGroup will receive the message destination WC8K.Chobits.8053_7.SuperGroup (If other TagObject's slot belong to SuperGroup, it will receive message, too.)

PS: WC8K.Chobits.CM or WC8K.Chobits.Controller is depends XML file setting discussed in previous Chapter.