The Protocol-Reader Manual

(version 1.0)

10rd April 2007

Preface

Version

The manual is written to support *Protocol Reader* Version 1.0.0

History

Here describe the history of the software version:

Version	*Date*	*Comments*
0.9a	2006.12.20	Internal Test Version
1.0	2007.2.28 2007.4.10	Software Released (Beta Vesion) Document Released

Introduction

Protocol Reader is a powerful protocol-analyser tool for parsing and displaying received binary messages. With an easy-to-use interface and highly advanced decoding engine, it can be used to support all levels of protocols such as IP based protocols (TCP, UDP, etc), Telcomunication protocols(MTP3,ISUP,TUP, etc) and user-defined packets.
Unlike other protocol analyzer tool, we provide advanced features to meet future needs. Instead of waiting long time for a new protocol decoder to be released, you can easily write a script to support new protocols in a incredible short time by yourself.

Objectives of this Manual

This manual describes all the basics and features that *Protocol Reader* provides, you'll get an understanding of what *Protocol Reader* is, what its features are, and how to use it.
This manual is part of documents to improve the usability of *Protocol Reader*.

Operation system Support

Windows XP/2000/NT/ME/98/95

About this Document

We hope you will find it useful, and look forward to your comments and valueble feedback.

1 Overview

1.1 Introduction

Protocol Reader is a powerful protocol analyse tool for parsing and displaying received binary messages. With an easy-to-use interface and programable kernel, it can support variety of protocols such as IP based protocols(TCP,UDP, etc), Telcomunication protocols (MTP3,ISUP,TUP, etc) and user-defined packet.

Unlike other protocol analyzer tool, we provide advanced features to meet future needs, not only for standard protocols, but also for user-defined *protocols*. Instead of waiting long time for a new protocol decoder to be released, you can easily write a script to support new protocols in a incredible short time by yourself.

◆ **Main features and characteristics:**

- Display packets content with detailed protocol information by hiberarchy.
- Excellent protocol-decoding kernel for supporting new or specific types of messages, Script development environment allow you to write a script with simple and familiar syntax, no boring program involved. *Protocol Reader* make it easier.
  Write excellent script and share it with others makes it easy to support even newest protocols.
- Plug-in is supported to deal with complicated protocols or specific requirement.
- *Protocol Reader* can be used as independent data analyzer, or integrated with other application. A binary-packet-accepting interface is provided for accepting raw data from any external applications.
- Smart real-time analyzer.
  You can use manual or automatic decode mode to deal with the constantly coming message.
- Two display mode, three comment mode and some alternation is provided for your preference, easy to use.
- Import and export feature is provided for packets management.

Note: *Protocol Reader* is not developed to be a sniffer tool. We focus our work on analyzing packet, not captureing packet, but you can intergrated it with sniffer softwares through the binary-packet-accepting interface, for details about this interface, See chapter 2.3.2 Communicate with other application

1.2 Who will benefit from this software

Protocol Reader can be used as message monitor or debug tool for software manufacturers, protocol analyzers and program developers.

It is useful when you develop or debug applications which deal with protocol or have data communication with each other. This software is helpful especially for those who want to have a tool to support their own protocol or their own frame format, because no other software can be found to meet such specific requirement.

1.3 Development and maintenance

Protocol Reader was originally developed by James.shen . This is the first released version, and ongoing development and continually improvement will continue. Our aim is to make *Protocol Reader* to be one of the best softwares that are comparable to any other commercial analyzer.

We will create and keep a loose group of individuals to fix bugs, add new functions, improve manual document and develop new protocol scripts.

Technically, the code is still considered beta, so there's always room for improvements. All contributions and feedbacks are welcome. Our team, the users and developers worldwide would benefit from your efforts, and we will merge useful contributions into the newly released software.

If you have any feedback about *Protocol Reader*, please send them through

- Website http://aries-studio.vicp.net/soft/ProtocolReader.html
- Email Address shen_bd@sohu.com

Note: When giving your feedback, it is helpful if you supply following informations

- Subject type (problem, comment or new script,etc.)
  description of your feedback (the error/warning message you get, and the relate script file, etc.)

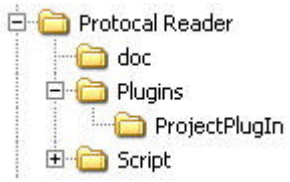
2 Quickstart

2.1 Installation

To install *Protocol Reader* , follow these steps:

1. Download this software from Website
   The most authoritative source for downloads is the *Protocol Reader* download website at www…..com/download.html.
2. For installion package, run setup.exe to install the software to the destination fold.
   For compress package, uncompress the software to the destination fold.
3. After a successful installation, you can begin to use *Protocol Reader*.

In following Figure, you can see a breakdown of the installation directories contained in the *Protocol Reader* distribution.



Note: As a beta version, you don't need to register this software.

2.2 User Interface

Protocol Reader's graphical user interface is easy to use. This chapter covers the main components of its Graphical User Interface (GUI).

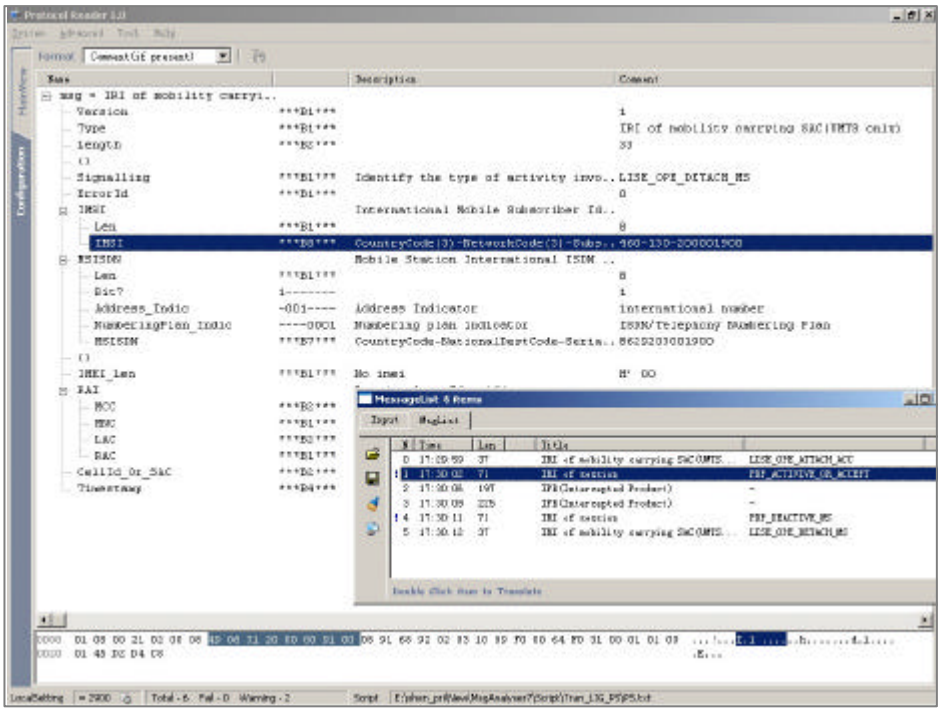


Figure 2.1 Main Window

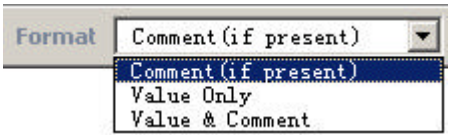
For this version, there are three main components to the window shown in Figure 2.1:

◆ MainView Window

Figure 2.1 shows what a typical MainView window looks like. The main pane is the protocol detail view. We use the *protocol tree* to display and access the details and components of protocols contained inside the binary packet. The tree looks familiar to you as one you might normally use to navigate a file system. The tree on the "Name Column" allows you to navigate around the fields of the protocol.Clicking on various parts of this protocol tree will highlight corresponding hexadecimal and ASCII output in the bottom pane. Both of the main pane and the bottom pane are adjustable in size by clicking on the seperator row between the panes and dragging up or down.

Table 2.1 Default Columns of the main pane

Column Name	*Description*
Name	display the name of protocol item (defined in script file, see…)
Location	display current location information of this item this column is initially hided, If required, you can click and drag the column heading to show it.

BitMask	show how many bits or bytes current item has. 'B' mean bytes, and 'b' mean bits.
Description	description or remark of current item if present (defined in script file, see…)
Comment	display the comment of the current protocol item's decoded value. there are three format for selection :  • Comment (if present) If the definition of value-to-comment dump is present, show comment, otherwise show the value. It is the default selection. • Value Only Just show the value • Value & Comment If the definition of value-to-comment dump is present, show value and comment, otherwise just show the value

■ Other components in MainView window

Display format Toolbar

Toolbar Item	*Description*
Comment Format	select the comment column display mode
TypeOF–protocol item display mode	toggle button for changing status of *TypeOF*-item, we expand the node of *TypeOF*-item when the button is in push-down status. for details about *TypeOF*–protocol item, see Chapter 3.3.2

Status bar

Statusbar Parts	*Description*
LocalSetting	2900 is the local Udp port for packet-accepting interface for details about packet-accepting interface, see Chapter2.3.2 Communicate with other application

Information	realtime statistics
Script	current protocol script file loaded

◆ Configuration Window (To be implemented)

We will implement Configuration Window in next version, and three enhanced feature will be provided:

- Filter Management allows you to enter a filter string restricting which packets are Displayed in the MessageList Dialog when accepting external packets. A filter string is a string defining some conditions that may or may not match a packet. Powerful filter provides a flexible mechanism to deal with specific packets
- Plug-In Management allows you to display and manage the Plug-in List, you can add, Delete a certain Plug-in lib.
- Multi-protocol script will be supported.

◆ MessageInput & MessageList Window

Message Window has two pages, One is Message Input Page for entering hex data, another one is MessageList Window for accepting external binary packets. We will give you an overview of the interface components here, and detail description about how to use it will be given in Chapter 2.3 Using Protocol Reader.

MessageInput Page

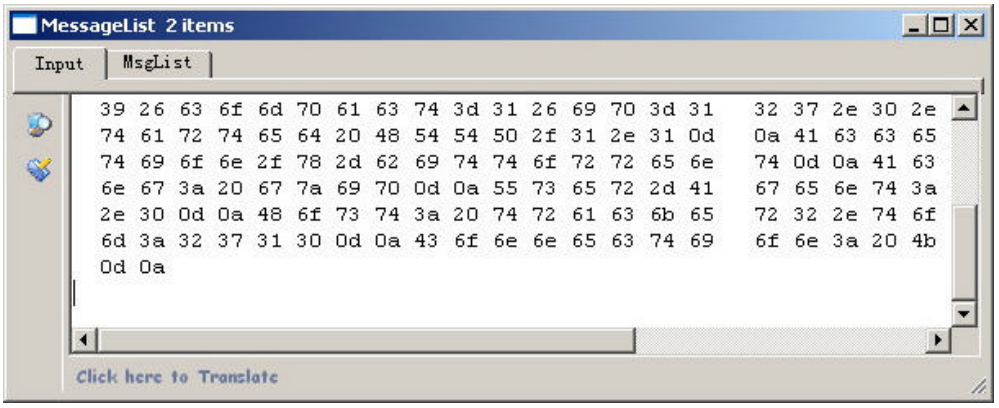


Table 2.2 Message Input Page Item

Menu Item	*Description*
	Translate current hex data
	Add current hex data to Message List directly

MessageList Page

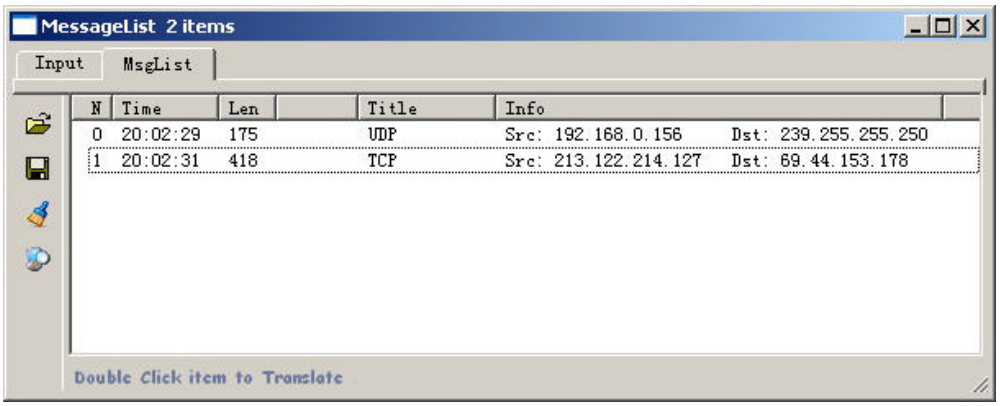


Table 2.3 Message List Page Item

Menu Item	*Description*
	Load a saved data file for viewing
	Save current data in the list to file
	Clear the list
	Toggle button When in push-down status, system will automatically translate external packet when accepting it.

Table 2.4 Default Columns of the Message List

Column Name	*Description*
N	Packet Number
Time	Time to when we accept this packet
Len	Packet length
Source Description	Identify data source (e.g from which PC, or direction …) For more information about how to use this field, see chapter …
Title	Summary of this packet. For more information about how to use this field, see chapter …
Info	

2.3 Using Protocol Reader

Having spent a lot of time discussing *Protocol Reader* 's features and benefits, and now, there may come a time to systematically explore how to use *Protocol Reader*. In this chapter, we will also provide you examples for quick start.

Protocol Reader is an easy-to use software, it can be use either as an independent protocol analyzer , or as an monitor or debug tool who can communicate with other application.

Note: In this chapter, we assume the protocol script needed is ready for use.

As for how to develop script, see Chapter 3. Developing Guide

2.3.1 Use as an independant protocol analyzer

Using *Protocol Reader* as an independent software, in other words, input hexadecimal message for analyzing is the major feature of typical protocol analyzer.

◆ **To analyze a single message, follow these steps**

- Select the protocol script file by clicking menu item System | Select protocol script file
  Select mainview displaymode by checking menu Advanced | Preference | Display mode
  Select display format from the Display format Toolbar
- Bring up the "MessageInput box"
  You can show it by click menu Tool | Show MessageList Window if it's not visiable.
- Input or copy the hexadecimal message to the "MessageInput box"

  You should input message like this (only hex character, Space and Tab is allowed)

```
00 0F 3D 2C 87 C1 00 0F      EA 57 38 5F 08 00 45 00
CF 0B 06 C4 00 50 6D B6      31 72 00 00 00 00 70 02
20 00 CE F7 00 00 02 04      05 B4 01 01 08 02
```

- Start to analyze the message by click button.  in the "MessageInput box"
- The result will be shown in the mainview window.

2.3.2 Communicate with other application

Accepting binary-packet from other application is one of the major features of *Protocol Reader*. *Protocol Reader* is not a sniffer tool, but it can be intergrated with sniffer softwares. We provide binary-packet-accepting interface, so the permitted source of the data packets are widely increased, you can receive any hex packet to "MessageList Dialog" for further decoding, not only from packet-capture softwares, but also from any user applications.

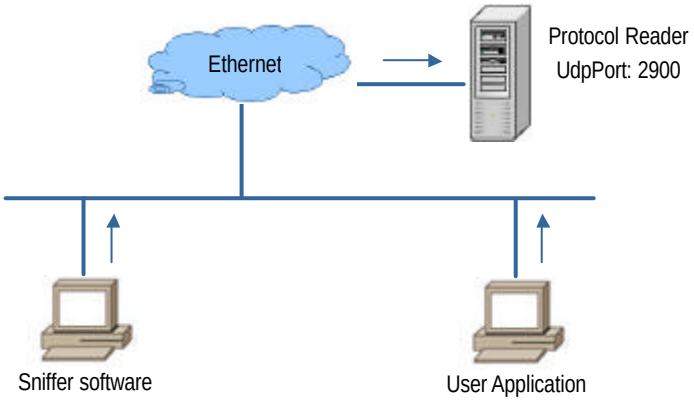


Figure 2.2 Network Architecture of Using *Protocol Reader*

- **Binary-packet-accepting interface**

Because *Protocol Reader* is designed to accept packets from many different applications simultaneously, we use UDP (User Datagram Protocol) as communication protocol, the default udp port is 2900. This message interface is open to any application programs as long as they have following header format:

	Field	Default	Description
Octet 0	FrameHeader	0x05	Header to identify this frame
Octet 1	SourceId	0x00	Identify data source (e.g from which PC, or direction …) For more information about how to use this field, see chapter …
Octet 2	ProtocolId	0x00	Reserved for this software's next version to deal with multi-protocol supporting.
Octet …	RawData		The payload stream need to be sent

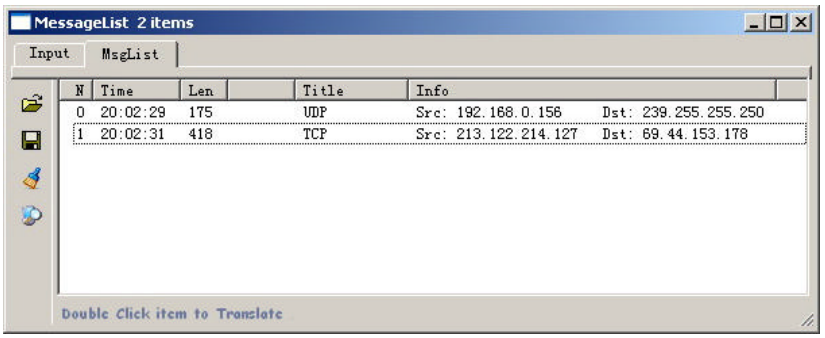
Note: In this version we have 1 megabyte(s) size kernel buffer to keep the accepted packets, if you encounter packets overflow, we will clear all received packets and restart accepting. Feature of configuring the buffersize will be provided in next version.

◆ **To start binary-packet-accept interface , follow these steps**

1. Select the protocol script file by clicking menu item System | Select protocol script file
   Select mainview displaymode by checking menu Advanced | Preference | Display mode
   Select display format from the Display format Toolbar
2. Bring up the "MessageList box"
   You can show it by click menu Tool | Show MessageList Window if it's not visiable.
3. Decide whether you would like to decode received packets in a automatical way while keep on accepting.

   In the "MessageList box" , if the  button is in push-down state, we will decode every packet automatically in real time when accepting it; otherwise, instead of decoding received packet automatically, we just add accepted packets into "MessageList box", you could decode these packets by double clicking corresponding item.

   Note: If the "MessageList box" is in automatical decode status, it will consuming a lot of CPU time, because *real-time analyzing* with large packets can be quite slow. So close automatical-decode-feature, unless it is very necessary to analyze live accepted packets.
4. Start accepting by checking the menu item System | Accept external data to translate
   The accepted packets will be shown in the MessageList box.
   It's a packet summary window, each line corresponds to one packet.

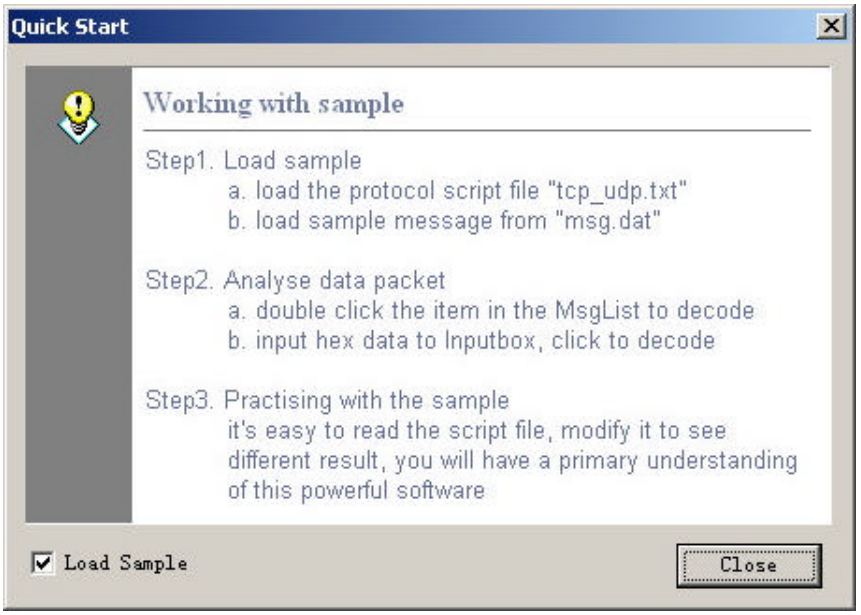


5. Using the "MessageList box"
   Once you have accept some packets, or you have opend a previously saved file, you can view the packets that are displayed in the packet list by simply double-clicking on a packet.

   Double-click one packet to analyze, result will be shown in mainview window.
   Right-click one packet to show the corresponding hexadecimal and ASCII content.
   Click button  to save all the packets in list to file.
   Click button  to load a saved file to list.

2.3.3 Demonstration

When first time use *Protocol Reader*, we will bring up a "quick start dialog" or you can click the menu item Help | Quick start to show the "quick start dialog" as follows:



Do as the dialog shows, it will give you a step-by-step guidance about how to use the main features of *Protocol Reader*.

Here is an example with a TCP pakcet, result is show in figure 2.3

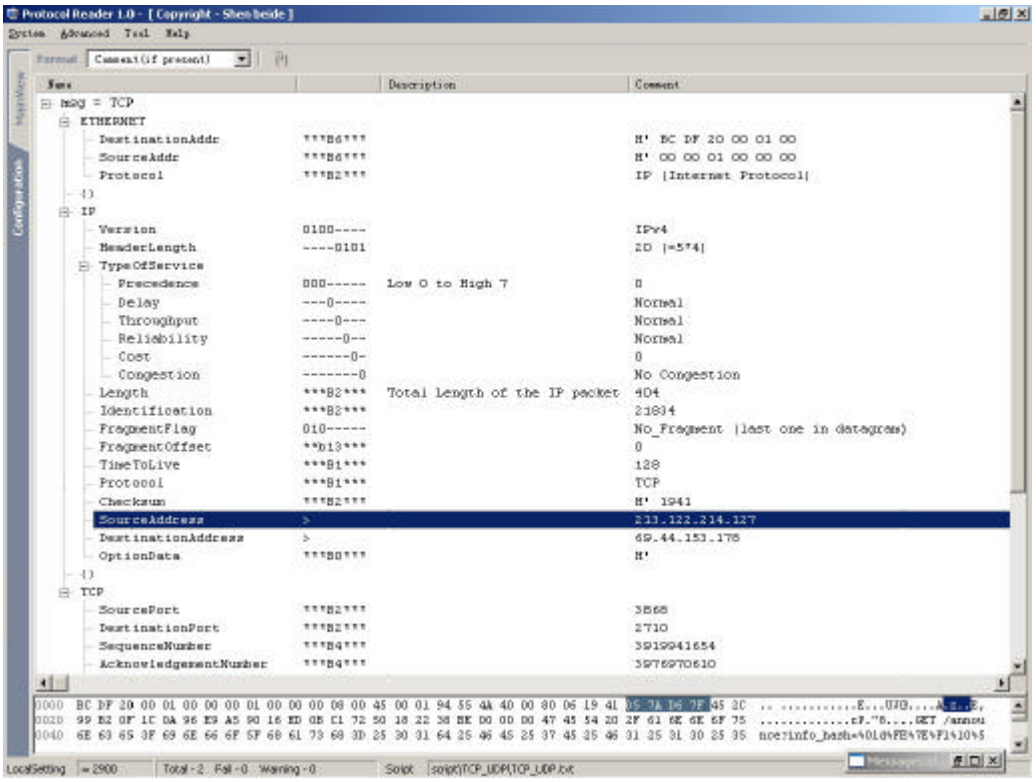


Figure 2.3 TCP packet example

In figure 2.3, Detailed information about Fields of this packet is generated in a tree-style view, we can see that the packet contain TCP inside IP inside an Ethernet packet.

You can then expand or collapse any part of the tree nodes to show more complicated data structures in the protocol, or collapse it to only show the summary.

- Selecting individual protocol fields by clicking on them will highlight corresponding hexadecimal and ASCII output in the bottom pane. The bottom displays the raw data both in hexadecimal and ASCII format.
- Right-click protocol field will bring up an information dialog to show you the definition of this field in script file. It will give you some hint about how this field is decoded or dumped.

  Example 1:

  when you right-click msg.ETHERNET.Destination.Addr, an information dialog is shown:

```
OCTET  DestinationAddr {

     21: void Translate(String strCurIdent, dynamic &curItem, long nIdxInArr)
         {
             23: sysFetchValue(strCurIdent, curItem, now(), 6,  true);
         }
};
```

  From the information box, we know that this field is decoded as a *octet* stream with six Bytes (see procedure *Translate()*, it's the base procedure for decoding, we will discuss it in next chapter) and displayed in "comment column" as hexadecimal string (default display mode for field with *octet* type) .

  Example 2:

  when you right-click msg.IP.FragmentFlag, an information dialog is shown:

```
UINT2  FragmentFlag:3 {

     103: String Dump(dynamic &curItem)
          {
              105: return strFormat("%s %s",
                                    (curItem & 0x02)? "No_Fragment":"Allow_Fragment",
                                    (curItem & 0x01)? " ":"(last one in datagram)");
          }
};
```

  From the information box, we know that this field is decoded as *unsigned short* with 3 bits and displayed in "comment column" as a description string generated by procedure *Dump()* (the base procedure for comment dumping, we will discuss it in next chapter) .

3 Script Developing Guide

This chapter will outlined some of the most important parts of develping *Protocol Reader*, a view of the syntax and development process will be presented. There are two basic component you should master before you could become an expert or a contributor to the *Protocol Reader* project :

- Protocol script
- Plug-in

3.1 Tutorial of syntax

Because *ANSI C* is a widely used programming language, we make it as our script-writing language, majority of the code base for script-implementing is plain *ANSI C*, knowledge about *ANSI C* will be sufficient for *Protocol Reader* development in almost any case.

Since *Protocol Reader* is not intended to be a C compilor, we just bring in the most necesssary parts of ANSI C's syntax to keep a more simple and efficient kernel. Althougth compare with the standard ANSI C language, there may have some tiny differencies, syntax for script is designed to make it easy to compose, analysize the hiberarchy of encoded data.

The tutorial, by being brief, does assume a basic knowledge of ANSI C programming, so our aim is to show only the essential elements of the language , but without getting bogged down in details, rules and specific programming techniques. At this point, we are not trying to be complete or even precise. We want to get you as quickly as possible to the point where you can write useful script.

Note: The *Protocol Reader*'s syntax is case-unsensitive.

3.1.1 Basic data types

Generally, every name (identifier) has a type associated with it. This type will determine what operations can be applied to the entity referred to by the name. Based on *ANSI C* standard, many small changes and additions have been made to the basic data types:

Type	Description
`Void`	null data type
`Char`	8 bits integer, capable of holding one character in the local character set
`Short`	16 bits integer
`Long`	32 bits integer
`Double`	double-precision floating point
`Octet`	an octect-stream is a sequence of characters

In addition, there are a number of qualifiers that can be applied to these basic types, we list all the data type supported as follows:

Table 2.4 Data type List

Type	*BasicType*	*Comment*
char, int1	Char	
short, int2, int	short	
long, int4	Long	
byte, uint1 bool	unsigned char	
word, uint2	unsigned short	
dword, uint4	unsigned long	
tchar	char	the type is used mostly for *char* array with fixed size
float	double	
octet	octet	hexadecimal sequence of 8-bit bytes
string	octet	a string is an array of characters with '\0' at the end, usually surrounded by " " or ' '
memo	octet	memo is used for large size octet-stream
void	-	null data type
choice	-	only used as a struct-member, represent there has one or more optional selection here

Note: For *Integral* and *floatingpoint* types, they can be mixed freely in assignments and expressions. Wherever possible, values are converted so as not to lose information.

3.1.2 Complicated data type: structure

A structure is a collection of one or more variables, possibly of different types, grouped together for convenient handling. Structures help to organize complicated data, The keyword *struct* introduces a structure declaration, which is a list of declarations enclosed in braces.

Syntax of structure declaration is as follows:

```
struct name {
    datatype  variablename;  // structure member
    ... ...
};
```

◆ Prefedined structure

A structure cannot be used in definition of instances unless it has been previously declared, but in some cases, we want to use a structure before the entity of structure has been declared, so we bring in the concept of *predefined structrue*. Example is given to show how to use predefined structure:

Example: predefined structure

```
...
struct Location;      // predefine(the entity they refer to must
                      // be defined elsewhere)
...
Location locat;       // use predefined structure
locat.x=1;
locat.y=2;
...
struct Location {     // structure entity
  long x;
  long y;
};
```

◆ Bit-fields

Sometimes it may be necessary to pack several member objects into a *unsigned integer* (e.g.byte,word,dword) . we offer the capability of defining and accessing fields within a *unsigned integer* directly. Structure members with *bit-field*, is a set of adjacent objects share one single data type object.

Example:

```
struct mydata {
   unsigned int nflag : 1;       // the highest 1 bit
   unsigned int value : 15;      // the lower 15 bits
};
```

This defines a variable table called *mydata* that contains two bit-fields. The number following the colon represents the field width in bits. The fields are declared *unsigned int* to ensure that they are unsigned quantities, and its format in octet-stream (bits always in high to low order) is:

Individual fields are referenced in the same way as other structure members: *mydata.nflags*, *mydata.value*, etc. Fields behave like smaller integers, and may participate in arithmetic expressions just like other integers. Padding field may needed if total bits of associated adjacent fields is not reach the data type's bits-size. Any bit-field is not arrays.

3.1.3 Basic syntax components

This chapter discusses the basic syntax components for composing a script file, but there will be no details here, because all the syntax is similar with ANSI C, it's easy for you to understand.

Notice: There will be some reductions and additions in the syntax of *Protocol Reader*. following list show some *ANSI C* syntax components we do not support:

- macros
- union, pointer, typedef
- array with two or more dimensions

1. Const definitions

Similar to *ANSI C*, we offer the concept of a user-defined constant to express the Notion that a value doesn't change directly. This is useful in several contexts. For instance, many objects don't actually have their values changed after initialization, symbolic constants lead to more maintainable code than do literals embedded directly in code. We use the keyword *define* to the declaration of an object to make the object declared a constant.

Form of const definitions	
Syntax	`# define const_identifier number|Plainstring`

Sample	`# define PI 3.14`
	`# define MAX_LEN 20`
	`# define INFO_STR "Welcome"`

2. Enumerations

The notion of enumeration in *Protocol Reader* differs from the enumeration notion in the *ANSI C*. An *enumeration* here is a type that can hold a set of values of constants specified by the user, it's a group of contants used very much like an integer type. The role of the identifier in the enum-specifier is analogous to that of the structure tag in a struct-specifier. In addition, enumerator names in the same scope must all be distinct from each other.

Form of enumerations	

Syntax	`Enum enum_identifier { enumerator-list }`
	
	`enumerator-list:`
	`enumerator`
	`enumerator-list, enumerator`

	enumerator: *identifier=constant-expression // Normal enum-item* *constant-expression // implicit enum-item*
Sample	Example 1: normal enumeration *enum MsgType {* *CMD = 0, // command* *REP = 1, // report* *ACK = 2 // response* *};* *long type;* *type = MsgType::CMD; // use enumerator* Example 2: implicit enumeration *enum MsgType {* *0, // command* *1, // report* *2 // response* *};* Note : Implicit enumeration is defined and used only for the kernel of *Protocol Reader*, for more details see …

3. Include

Include is the preferred way to tie the declarations together for a large program. It guarantees that all the script files will be supplied with the same definitions and variable declarations, and thus makes it easy to handle shared codes. Naturally, when an included file is changed, all files that depend on it must be reload and recompiled.

Form of include	
Syntax	*# include "filename"*
Sample	*# include "sys.h"* *# include "header.h"* Note : *filename* is searched in association with the path of the original source file.

4. Variable Declarations

All variables must be declared before use. A declaration specifies a type, and contains a list of one or more variables of that type, as in

```
int x, y;
char linebuffer[1000];
```

Variables can be distributed among declarations in any fashion; the lists above could well be written as

```
int x;
int y;
char ch;
char linebuffer[1000];  // line buffer
```

The latter form takes more space, but is convenient for adding a comment to each declaration.

A variable may also be initialized in its declaration. If the name is followed by an equals sign and an expression, the expression serves as an initializer. Here are some examples illustrating the diversity of declarations:

```
struct point {
        int x;
        int y;
};
...
char ch='a';
double pi=3.1415926;
int size=MAXLINE+1;
string str[2]={ "welcome", "you" };
point pt= { 320,200 };
```

If the variable in question is not automatic, the initialization is done once only before the program starts executing, and the initializer must be a constant expression. An explicitly initialized automatic variable is initialized each time the function or block it is in is entered; the initializer may be any expression.

5. Statements and program blocks

As that in *ANSI C*, fundamental constructions are privided for well-structured programs. Here are a summary and basic elements of statements supported:

Statements Syntax

```
Statement:
     expression-statement
     If (condition_expr) statement else statement
     If (condition_expr) statement
     while (condition_expr) statement
     for(init_expr;condition_expr;expr) statement
     break
     continue
     return expression

statement-list:
    statement statement-list
    { statement }
```

In our script, the semicolon is a statement terminator. Braces *{* and *}* are used to group declarations and statements together into a *compound statement*, or *program block*, so that they are syntactically equivalent to a single statement. The braces that surround the statements of a function are one obvious example; braces around multiple statements after an *if*, *else*, *while*, or *for* are another. (Variables can be declared inside *any* program block). There is no semicolon after the right brace that ends a block.

7. Functions

Functions break large tasks into smaller ones, and enable people to build on what others have done instead of starting over from scratch. Appropriate functions hide details of operation from parts of the program that don't need to know about them, thus clarifying the whole, and easing the pain of making changes.

Form of function definitions	
Syntax	*Return-type function_name(argument declarations)* *{* *declarations and statements* *}*

◆ Return-type

Functions may return values of any basic types defined in chapter 3.1, and values with array are not allowed.

◆ Argument declaration

Communication between the functions is by arguments and values returned by the function, and through global variables. Type of the argument can be any basic type defined in chapter 3.1 (except *void*) or user-defined structure. And argument can be passed *by value* or *by reference* (It can more efficient to pass a large object *by reference* than to pass it *by value*).

Here are some examples for argument passing:

```
Struct point {
        int x;
        int y;
};
bool setLocation1(point  pt);   // pass by value
bool setLocation2(point& pt);   // pass by reference
                                // the variable'pt' can be altered by the function
void setDatas(long data[2]);    // pass array
```

There are another two argument-passing modes we should mention:

1) Arguments with Unspecified number

For some functions, it is not possible to specify the number and type of all arguments expected in a call. Such a function is declared by terminating the list of argument declarations with the ellipsis(. . .), which means "may has more arguments.''

For example:

```
extern string strFormat(string strFormat, … );      // definition
...
strFormat("Hello, world!\n");
strFormat("My name is %s %s\n", first? ?ame, second? ?ame);
strFormat("%d + %d = %d\n",2,3,5);
```

From these examples, you could know that such a function must rely on information not available to the compiler when interpreting its argument list. Clearly, if an argument has not been declared, the compiler does not have the information needed to perform the standard type checking and type conversion for it.

Note :

In *Protocol Reader* software, this argument-passing mode can only be used for extern function , as shown in example, function should be defined as an external function and be implemented in plug-in library. For more information about how to implement extern function, see chapter 3.3 Developing plug-in.

2) Adaptive arguments

New argument type "*dynamic*" makes it easier to pass data between functions. "*dynamic*" mean adaptive to *any basic type* (not array), in other words, you can pass any data (with *basic type*) to such an argument of the same function. It is useful when you want to use a single function to share one of its argument to deal with different data types. And similar to passing data by reference, it provide another way for the called function to alter a variable in the calling function.

An example for using *adaptive argument* is as follows:

```
void print_data(dynamic& data)       // definition
{
   ...          // to print the data which may be with different type
}
…
double val=1.68;
print_data(val);                  // pass a ' double' value
…
string str="Welcome";
print_data(str);                  // pass a 'string' value
```

◆ Predefined function

A function cannot be called unless it has been previously declared, but in some cases, we want to call a function before the entity of the function has been declared, so we bring in the concept of *predefined function*. Example is given to show how to use predefined

function:

```
long max(long x,long y); // predefine ( the entity must be defined elsewhere )
...
long val=max(2,0);        // call predefined function
...
long max(long x,long y)   // function entity, argument must match that in predefined header
{
      return (x>y)? x:y;    // function body
}
```

◆ External function

A function can be defined as a external function, meaning that the entity of the function is not implemented in the corresponding script files, instead, it might be implemented in plug-in library.

In *Protocol Reader*, external function is an usual way to bring in a library (for more details about bring in an external library, see …). To declare an external function, just add keyword *extern* in front of the function header, for example:

```
extern double sqrt(double v);        // bring in a external function
...
double val=sqrt(2.0);                // call external function
...
```

8. Standard Library

Although similar to *ANSI C*, the kernal compiler of *Protocol Reader* is independent of any other C compilor, so there is no standard library (such as mathematical functions, string functions, etc) initially.

In order to develop a good script, we add build-in functions which are most necessary and declare it in "sys.h". Furthermore, you could create, manage commonly used functions as a shared library (or as a library just for a given protocol) by your own. You also can merge libraries develped by other *Protocol Reader's* user into you project. A well organized library can save you a lot of time in furture development.

And there are two way to create your own library: one is writing the functions in script file directly; the other is developing an external, plug-in function (remember that the latter mode is more efficient, for more information, refer to chapter 3.3 Developing plug-in).

3.1.4 Enhanced syntax components

Section below shows the syntax of the enhanced components supported by *SDE* (script develop environment). Detail about the meaning, where and how to use these syntax components will be illustrated in next chapter.

Syntax

```
TypeOf    struct_Identifier

ContentOf struct_Identifier

DeclareMap(Item|ImplicitItem,declareMap_identifier)
{
     variablename [nTag] basictype
     variablename [nTag] struct_Idendifier
     variablename [nTag] TypeOf struct_Idendifier
}

ChoiceMap declareMap_identifier
```

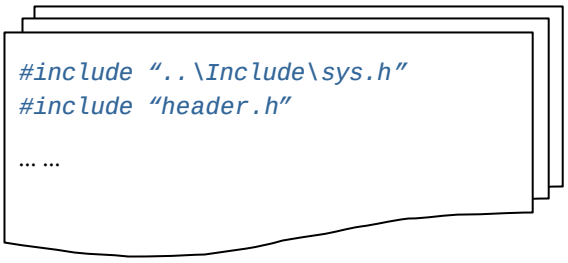
3.1.5 Architecture of a typical script

Before going into the development process, you should have an understanding of how the script files is organized and what a typical script is like.

- **Hiberarchy of the script files**

In order to understand the framework of script files, you could open the sample script provided in *Protocol Reader*'s installation fold. Note that the sample do not necessarily address the entire range of using *Protocol Reader*, it is merely a sample.

Example: \script\TCP_UDP\main.txt



1. File "*sys.h*" must be included at the beginging of the root script file. All the system reserved constants and functions are defined in this file.
2. File "*header.h*" is included to define global constants, global functions and the external functions (implemented in plug-in dll) for the current protocol. initially Content of this file is initially empty, and is left for your enhancement. Of couse, you can organize the included files by your habit.

◆ Syntax architecture for script element

In *ANSI C*, structure is an useful component help to organize complicated data, so it can be very suitable to present the hiberarchy of a protocol's fields.
Let's start from a piece of code from example \script\helloworld\main.txt

```
#include "..\Include\sys.h"

#define PROTOCOL_ROOT     msg
#define NETWORK_TRANMODE  H_TO_L     // default value is L_TO_H

……

////////////////////////////////////////////////////////////
struct msg
{
   BYTE      Header;                  // Framing header
   UINT2     PacketType : 2 {         // Type of this Packet
               enum description {
                  0,   // Command
                  1,   // Ack
                  2,   // Report
               };
               string Dump(dynamic& curItem)
               {
                  String str;
                  str=sysGetEnumComment(curItem,"description","Equal","-");
                  sysSetTitle(0,str);
                  return str;
               }
             };
   UINT2     SequenceNumber: 14 ;
   BYTE      ContentType {            // Type of following content
               enum description {
                  1,   // Packet (string)
                  2,   // Packet (ip address)
                  3,   // Packet (bcd stream)
                  4,   // Packet (timestamp)
               };
               string Dump(dynamic& curItem)
               {
                  String str;
                  str=sysGetEnumComment(curItem,"description","Equal","-");
                  sysSetTitle(1,str);
                  return str;
               }
             };
   Choice    Content {
               DeclareMap(ImplicitItem,DataField)
               {
                  StringWithLenAndZeroEnd [1]  tStringWithLenAndZeroEnd
                  IpAddress               [2]  tIpAddress
               };
               ChoiceMap DataField

               void Translate(String strCurIdent,dynamic& curItem,long nIdxInArr)
               {
                   sysTranChoiceItem(strCurIdent,curItem,this.PacketType);
               }
             };
};
```

Figure 3.1 sample script

We will not explain any detail about the script here. Just focus on the architecture!
Is there any thing common between the protocol fields ? Yes, see this script carefully, you could find two important thing:

1) *Structure* is used to organize the protocol hiberarchy

Structure is used for establishing parent/child relationships between protocols and fields, as well as associating data with a particular field so that it can be stored in the logical tree and displayed in the GUI protocol tree.

The structure (include all its members) can be regarded as one level of a protocol.

2) Each of the field has the form

```
struct identifier_name      // struct_description
{
    ……
    datatype  field_identifier {    // field_description

                decode attribute
                dump attribute      } attribute_block(optional)
            };
    ……
}
```

Figure 3.2 form of individual protocol field

Typical script for a certain protocol field is composed of four parts: the field definition, the field description, the decode-attribute and the dump-attribute for the field.

- Field Definition : define the name and data type of the protocol field
- Field Description : a string of remark for this field (usually writed in the same line as the Field Definition), used in display
- Decode Attribute : program block which describes how to decode this field. It's a *optional* script block, if not exist, default decode-rule will be used. For more information, refer to …
- Dump Attribute : program block which descirbes how to comment decoded value of the field. It's a *optional* script block, if not exist, default dump format will be used. For more information, refer to …

We can see this four parts from following result generated by the *Protocol Reader*:

Field Definition DataLen Description Dump Attribute

Name		Description	Comment
⊟ msg = Report			
Header	***B1***	Framing header	H' FE
FlagACK	10------	Flag of Acknowledgement	Report
SequenceNumber	**b14***		520
PacketType	***B1***	Type of this packet	Packet (string)
⊟ Content()			
strData	***B7***	string content	T' welcome

```
0000  FE 82 08 01 07 77 65 6C 63 6F 6D 65 00                .....welcome.
```

Note:
When studing the sample above, it is recommended you open the script source \script\helloworld\main.txt to see the entire script and corresponding protocol definition. It will help you understand the framework of the script file.

◆ Techniques

For most protocol analyse softwares, when it come to trying to support a new protocol, you have to understand the internal framework of the software, write new codes for this specific protocol, merge codes to the main software, and compile the entire software. It is not an easy thing for even the seniror expert, and it will take you a lot of time coding and testing.

We are always thinking about is there any easier way for people to develop a decoder or monitor for a complex protocol ? Now, things get better. The *Protocol Reader* could realeas you from all the annoying task.
We know that the only one basic way of dealing with complexity is: divide and conquer. A problem that can be separated into two sub-problems that can be handled separately is more than half solved by that separation. This simple principle can be applied in amazing variety of ways. it can be the fundamental approach to handling the inherent complexity of a protocol. Similarly, the process of developing a protocol analyser can be broken into distinct activities.

We find that all the protocol's hiberarchy is similar, and the main differences are how to decode specific field and what's the meaning of decoded value. So, in *Protocol Reader*, insteading of dealing with protocol itself, we *focus our workload on individual field* of the protocol, any field is relatively independent and coded like Figure 3.2. Separate the complex protocol into smaller one help to simplify the problem, make it easier to test and save your valuable time.

In addition, default decode and dump rule for specific datatype (see Table3.2) is provided to fit most need; if there are no special requirements, you don't need to write any codes in attribute block, system will do all that for you. That's what is called "Avoiding Program".

Protocol Reader provides you with an more efficient and easier way to develop protocol analyzer. In chapter 3.2, we will show you the development process step by step in details.

3.1.6 System reserved consts and functions

This section provides a comprehensive reference to the reserved keywords and functions of *Protocol Reader*. All the keywords are case-unsensitive.

Table 3.1 System reserved const

Const Item	Type	Description
PROTOCOL_ROOT	M	The root structure for starting decode
PROTOCOL_NAME	O	Name of the protocol
NETWORK_TRANMODE	O	Define network transmit order for byte stream. This const is used when decode a field with integer or unsigned integer datatype. Possible value: *H_TO_L* *// high to low* *L_TO_H* *// low to high* (default value for system)

Example:

```
#define  PROTOCOL_ROOT        msg
#define PROTOCOL_NAME         TCP/IP
#define NETWORK_TRANMODE    H_TO_L
```

NOTE

For *NETWORK_TRANMODE*, scope rule is introduced when there are one or more different declaration exist. Following is an example:

```
#define NETWORK_TRANMODE    H_TO_L          // global declaration for transmit mode
struct msg
{
    UINT2 Data1;                            // use global transmit mode to decode
    UINT2 Data2 {
        #define NETWORK_TRANMODE   L_TO_H  // local declaration hide global one
        …
    };
    UIN2 Data3;                             // use global transmit mode to decode
};
```

Type： M = mandatory
O = optional

Table 3.2 System reserved functions

Type	Functions	Descriptions
Input stream manage	Now	Get current location of current input stream
	TotalLength	Get total length of current input stream
	sysFetchValue	Fetch value (with number, octet or structure datatype) from input stream
String manage	strFormat	Provide formatted output conversion
	Strlen	Return length of octet stream
	strGetAt	Get one character from octet stream
Decode proc	sysTranChoiceItem	Decode from current stream locat using specific structure
	Translate	*Virtual function* for overriding if you have some specific requirement when decoding a protocol field
	Integrate	The decode of a *TypeOf* structure starts by calling this function. Every *TypeOf* structure must contain one function called *Integrate()*
Dump proc	sysSetTitle	Set Title information
	Dump	*Virtual function* for overriding if you have some specific requirement when dumping a decoded data of a field, always return a string for display.
Others	sysGetEnumComment	Get Comment for a value using certain *enum*
	sysAbort	Abort the decode or dump process, and give a description

Note :

For system reserved functions, all of them are declared in "*sys.h*". The entity of these Functions are embedded in *Protocol Reader*. if we want to create new functions, we have two choices:

[1] declare the function as an external function (as that in "*sys.h*") and implement it in plug-in dll, see … for more information.

[2] write the function as an internal function, include header,argument and the entity just as we write a funciton in *ANSI C* program.

- **SysFetchValue**

void *sysFetchValue*(string strCurIdent, dynamic& FetchValue, long locat, dword param, bool bMoveLocatPointer);

Get value from input stream and assign it to *FetchValue*. You can pass any variable with *integer, octet or structure* datatype to *FetchValue*. This function is used when decoding a packet.

Parameters

strCurIdent current protocol field name

FetchValue entity for saving the fetched data

locat current locat of input stream

param only available for octet type, mean length of bytes to be returned

bMoveLocatPointer

true for moving the locat pointer after data fetched, or false with no pointer move

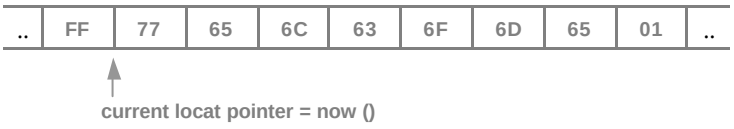
Return

Return fetched data in *FetchValue*.

Example

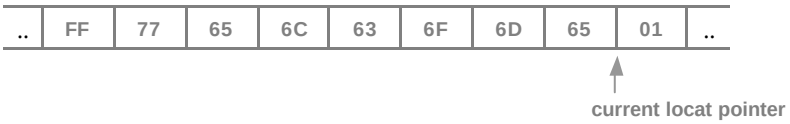
1. Fetching string data, with bMoveLocatpointer=true

Stream status before call fetch function -> transmit direction




```
...
string str;
sysFetchValue(strCurIdent,str,now(),7,true);        // str is "welcome" now
...
```

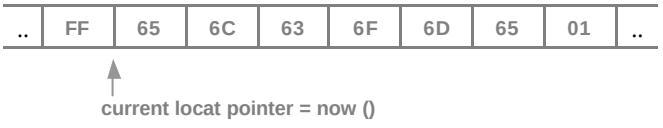
stream status after call fetch function



2. Fetching integer data, with bMoveLocatpointer=false

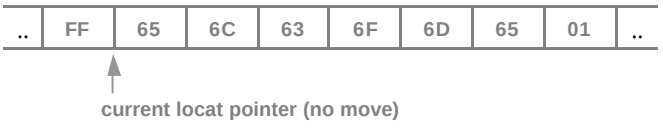
this locat-pointer's control mode is very useful when you want to preview a value but do not want the locat-pointer be moved.

Stream status before call fetch function -> transmit direction




```
...
#define network_tranmode H_TO_L.
word len;
sysFetchValue(strCurIdent,len,now(),0,false);       // len is 0x656C now
...
```

stream status after call fetch function



SysTranChoiceItem

void *sysTranChoiceItem* (string strCurIdent, dynamic& curItem, long ElemId);

Decode from current stream locat using specific struct (structId=elemId), this function is designed to parse optional structure, decoded structure will be added to choice list of *curItem* which must be a *choice* datatype.

Parameters

strCurIdent	current protocol field name
curItem	current protocol field to decode, this field must be a *choice* item
elemId	elementId corresponding to the *nTag* declared in *DeclareMap*, refer to chapter: Field with choice attribute for more information about *DeclareMap*.

Return

Return decoded optional structure, and add it to choice list of *curItem*.

Example

1. Following is a piece of codes from "\script\tcp_udp\main.txt"

```
...
struct tIP
{
  tIPHeader  IP;
  Choice   {          // following is the attribut-block for this field
              DeclareMap(Item, IpData)
              {
                  UDP     [0x11]   tUDP
                  TCP     [0x06]   tTCP
              };
              ChoiceMap  IpData

              void Translate(String strCurIdent,dynamic& curItem,long nIdxInArr)
              {
                  sysTranChoiceItem(strCurIdent, curItem, this.IP.Protocol);
              }
           };
};
...
```

In this example, after decoding IP header (refer to *RFC* for details), we can know from the value of *this.IP.Protocol* what content follows, udp, tcp or other protocol.

Then we create a *Choice* item. In attribute-block, there is one *DeclareMap* that list all the candidate protocol (expressed by structure, for more informations about *DeclareMap*, see …), and *sysTranChoiceItem(…)* is provided to decode such uncertain data. This function compare the value of *this.IP.Protocol* and the value in *IpData* (declared by *ChoiceMap*), if *this.IP.Protocol* equal 0x11, we use struct *tUDP* to decode following bytes, or if *this.IP.Protocol* equal 0x06, we use struct *tTCP* to decode following bytes.

SysSetTitle

void *sysSetTitle* (uint1 TitleN, string strTitle);

This function set *the title information* of current packet, and *the title information* provide you one-line summary that displayed in MsgList_Page. The summary display brief but important information relative to the packet, and allows the user to browse quickly through the packet trace without having to look at each packet decode.

Parameters

TitleN	title index, the value can be 0 or 1
strTitle	text of the content

Remarks

The title information will be passed to MsgList_Page as one-line summary

Example

1. Following is a piece of codes from "\script\tcp_udp\header.txt"

```
struct tIPHeader
{
    ...
    BYTE   Protocol {
                enum  description {
                        ...
                        0x04,     // IP
                        0x06,     // TCP
                        0x11,     // UDP
                        ...
                };

                String Dump(dynamic& curItem)
                {
                    String  str;
                    str= sysGetEnumComment(curItem,"Description","Equal","-");
                    sysSetTitle(0,str);       // result of title0 is shown in Figure 3.1
                    return  str;
                }
            };
    ...
};
```

MsgList Title 0 Title 1

N	Time	Len		Title	Info
0	20:02:29	175		UDP	Src: 192.168.0.156 Dst: 239.255.255.250
1	20:02:31	418		TCP	Src: 213.122.214.127 Dst: 69.44.153.178

Figure 3.1 example of one-line summary

■ SysGetEnumComment

string *sysGetEnumComment* (long value, string strEnumListName, string strMode, string strDefault);

This function provide you a way to get a value's corresponding comment, in other words, it is a value (must be a *integer*) to meanning utility.
Note: The value-comment list is declared in *enumeration* items.

Parameters

value — an integer

strEnumListName — *enumeration* name contain the value-comment list

strMode — the following options are supported for *strMode*,

Value	Meaning
"Equal"	Get the corresponding comment if the *value* 'Equal' one of the *enum-item*
"Or "	Get all corresponding comments if the *value* 'Or' *enum-item* unequal to 0, comments are sperated by '|'

strDefault — this string is returned if the integer is not found in current *enum*

Return

Return a string that show the meanning for this value.

Example

1. In the following example, we show you how to use this function

```
enum FileAttribute {
        0x01,    // Readable
        0x02,    // Writable
};

String str;

str= sysGetEnumComment(2,"FileAttribute","Equal","-");      // sample of 'equal'
// str is "Writable" now

str= sysGetEnumComment(3,"FileAttribute","Or","-");         // sample of 'or'
// str is "Readable|Writable" now

str= sysGetEnumComment(5,"FileAttribute","Equal","-");      // not found
// str is "-" now
```

3.2 Developing protocol script

The purpose of this session is to get the reader started as quickly as possible, example is provided to show you step-by-step how to develop script. We'll start with the made up "HelloWorld" protocol (you can find this sample at \script\helloworld\ )

3.2.1 Definition of the sample protocol

In following tables, the format of the protocol parameters are specified.

	8	7	6	5	4	3	2	1
1	Frame Header							
2	PacketType		Sequence number (MSB)					
3	Sequence number (LSB)							
4	Content type							
5	Content							
..								

Figure 3.3 protocol format

The following codes are used in the *helloworld* protocol parameter field:

a) *Frame Header*

One byte flag to identify this message, always be 0xFE

b) *PacketType*

Type of this packet, 2 bit

The following codes are used in the *PacketType* subfield:

0 0 command

1 0 report

c) *Sequence number*

Sequence number for this packet

d) *Content type*

One byte field, identify the type of following content

The following codes are used in the *ContentType* subfield:

1 octet stream

2 ip address

3 bcd data (positive BCD)

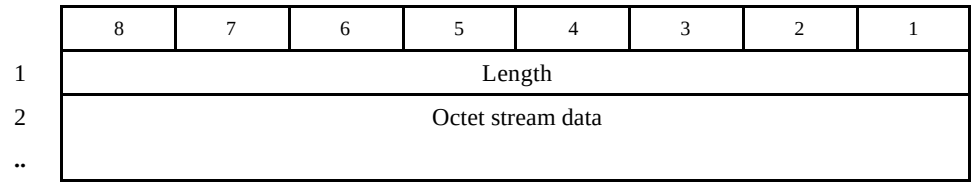
4 timestamp

5 string with *typeof* attribute

6 array with fixed size

7 array with unfixed size

Figure 3.3.1 Format for content field (octet stream)



a) Length: The length of the octet stream

Figure 3.3.2 Format for content field (ipaddress A.B.C.D)

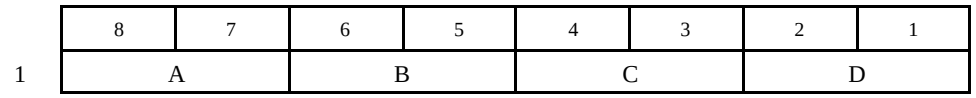
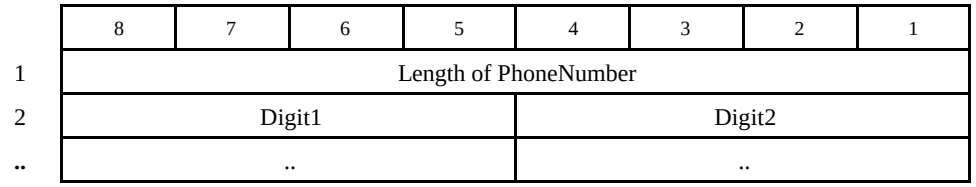
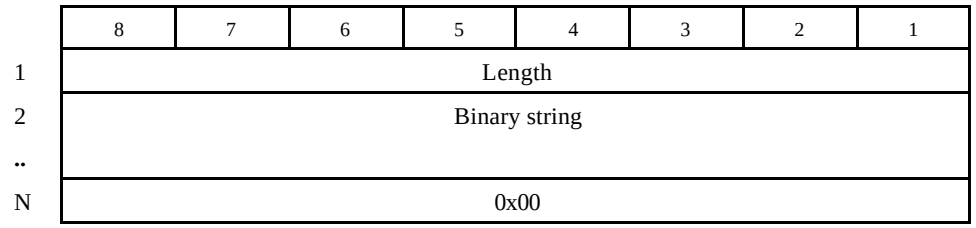


Figure 3.3.3 Format for content field (positive bcd)



a) Length: Number of BCD digits

Figure 3.3.4 Format for content field (string with *typeof* attribute)



a) Length: The length of the string(do not count the zero byte)

Figure 3.3.5 Format for content field (uint array with fixed size = 2)

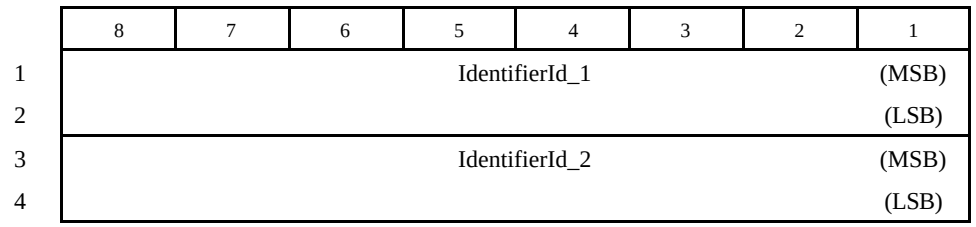
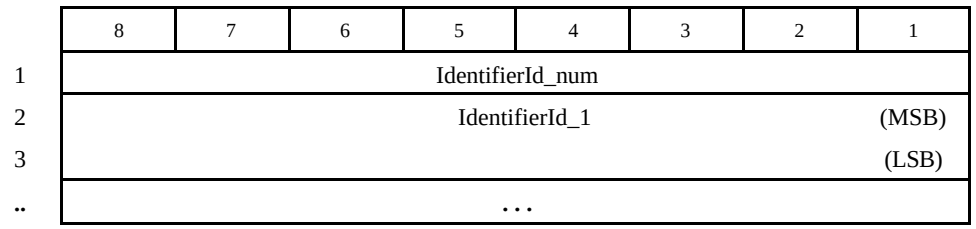


Figure 3.3.6 Format for content field (int array with unfixed size)



a) Identifier_num: The total number of *IdentifierId*

Figure 3.3.7 Format for content field (timestamp)

	8	7	6	5	4	3	2	1
1	Timestamp (MSB)							
2								
3								
4	(LSB)							

■ Aim of this example

By this example, we cover almost every part of the develop technique commonly used.

Each of the subfield give you an speicifc demonstration :

Item	**Demonstration**	**Refer.**
Header	decode and display *hex* field, using default rule	3.2.3.2
PacketType	decode and display corresponding meanning for *number* field with bit definition	3.2.3.2
SequqnceNumber	decode and display *number* field (consider network transmit mode)	3.2.3.2
ContentType	decode and display *byte* field	3.2.3.2
Content	decode *optional* field	3.2.3.2
Content-octet	decode and display octet field	3.2.4.1
Content-ipaddress	decode and display a *user-defined* field, demonstrate overriding system reserved function	3.2.4.2
Content-bcd	decode and display a *user-defined* transformed field, demonstrate calling a global function	3.2.4.3
Content-typeof_string	demonstrate field with *Typeof* attribute (two ways)	3.2.4.4
Content-fixedsize_arr	demonstrate field with fixed size array	3.2.4.5
Content-unfixedsize_arr	demonstrate field with unfixed size array	3.2.4.6
Content-timestamp	decode and display a *user-defined* complex field, demonstrate how to implement external function (plug-in development)	3.2.4.7

3.2.2 begin

To write a protocol script, first, you should follow these steps:

- Include "sys.h" at the beginning of your script file, it is needed for system reserved functions
- Set the root structure name for starting protocol decode by *PROTOCOL_ROOT*
- Set the name of the protocol by *PROTOCOL_NAME*, (optional in this version)
- Set the network transmit mode for byte stream by *NETWORK_TRANMODE* , if this const is not present, default value *L_TO_H* will be used

  Note: you can make a declaration anywhere for any specific protocol field if needed, the declaration accords with the *scope rule* of *ANSI C*, that is, it can be redefined to refer to a different entity within a program block, after exit from the block, the const resumes its previous meaning.

```
# include "..\Include\sys.h"

# define PROTOCOL_ROOT          msg
# define NETWORK_TRANMODE       H_TO_L      // for this protocol
```

3.2.3 Define root node of this protocol

Root node is the access structure for decoding a stream. In this session, we will give you an example. Note that we use structure to establish and organize parent/child relationships between protocol and fields, as well as associating data with a particular field so that it can be stored in the logical tree and displayed in the GUI protocol tree.
There are two steps you should follow: the first step in the development process is to create high-level hiberarchy which should accord with the format of the protocol. Then, we should implement every individual field (write decode-attribute and dump-attribute if needed).

3.2.3.1 Create hiberarchy structure

According to the "HelloWorld" protocol format, the hiberarchy of *root level* is shown as follows:

```
...
struct msg
{
    byte        Header;
    uint2       PacketType:2;
    uint2       SequenceNumber:14;
    byte        ContentType;
    choice      Content;
};
```

First we define a struct named "*msg*", it is the root node of this protocol. Then, we add field memebers to the structure according to Figure 3.3 protocol format. For the last member of the structure named "*Content*", it's a *choice* item composed of optional parts of the protocol, and the number and attribute of these optional parts are lie on the value of "*ContentType*" (refer to Figure 3.3 protocol format).

3.2.3.2 Implement every individual field

In this session, we write decode-attribute and dump-attribute (if needed) for every field.

1) Field: Header

The field *Header* is decoded and dumped using default dump rule, attribute block is unnecessary, only field description is added. This field is coded as follows:

```
struct msg
{
    ...
    byte        Header;             // Framing header
    ...
};
```

2) Field: PacketType

The field *PacketType* is a *number* field with bit definition, it can be decoded by default decode process (refer to Table 3.1) . Because we want to show corresponding comment of the value when display, virtual function *Dump()* should be overrided to meet our need (do not use default dump rule).

This field is coded as follows:

Step 1: write codes for decoding (do not need here, use default process), add field description

```
...
uint2       PacketType: 2;          // Type of this packet
...
```

Step 2: override virtual function *Dump()* to implement our display requirement.

```
...
uint2       PacketType: 2 {         // Type of this packet
                        enum description {
                            0,  // Command
                            1,  // Ack
                            2,  // Report
                        };
```

```
        string Dump(dynamic& curItem)
        {
            String str;
            str=sysGetEnumComment(curItem,"Description","Equal","-");
            sysSetTitle(0,str);
            return str;
        }
    };
```

- Here we define a value-to-comment list named "*description*" at first, then override *Dump()* and call *sysGetEnumComment()* to translate the value of *curItem* to its comment, the result string is returned for display.
  And *sysSetTitle()* is called to set the *Title0* of one-line-summary information.

3) Field: SequenceNumber

The field *SequenceNumber* is a *number* field with bit definition, it is decoded by default decode process (refer to Table 3.1) and dumped by default dump rule. Because the datatype occupy more than one byte, the *NETWORK_TRANMODE* is considered when fetch data from byte stream, we define this constant at the begin of the script here, or you can define it in its own attribute-block of this field for specific needs.
This field is coded as follows:

```
...
uint2       SequenceNumber;
...
```

4) Field: ContentType

The field *ContentType* is a *byte* field, it is decoded by default decode process (refer to Table 3.1) , and virtual function *Dump()* is overrided to get corresponding comment of the value when display. This field is coded as follows:

```
uint2       ContentType {           // Type of following content
                        enum description {
                            1,    // Packet (octet stream)
                            2,    // Packet (ip address)
                            3,    // Packet (bcd stream)
                            4,    // Packet (string with typeof attribute)
                            5,    // Packet (array with fixed size)
                            6,    // Packet (array with unfixed size)
                            7,     // Packet (timestamp)
                        };
                        string Dump(dynamic& curItem)
                        {
                            String str;
                            str=sysGetEnumComment(curItem,"Description","Equal","-");
                            sysSetTitle(1,str);
                            return str;
                        }
                    };
```

5) Field: Content

The field *Content* is a *choice* item, representing that there has one or more optional selection here. To deal with field with this kind of type, you should follow these steps:

Step 1: Use *DeclareMap* to define all possiable choices

```
…
Choice      Content {
                DeclareMap(ImplicitItem,DataField)
                {
                    item   [1]   tOctetStream
                    item   [2]   tIpAddress
                };
                ChoiceMap DataField
            };
…
```

- Here we define a *DeclareMap* named "*DataField*" , listing all possiable Child branch, the attribute of the *DeclareMap* is *ImplicitItem* (for more details about *DeclareMap*, see chapter 3.3.3).
  Every choice item has one IdentifierID, in this demo, we set the ID according to the definition of *Content type*.
  Then keyword *ChoiceMap* is declared to link "*DataField*" with this field.

Step 2: override virtual function *Translate()* to implement how to process optional items.

```
struct tOctetStream;    // predefine structure before use
struct tIpAddress;
…

struct msg
{
    ...
    byte        ContentType;
    Choice      Content {
                    DeclareMap(ImplicitItem,DataField)
                    {
                        item   [1]   tOctetStream
                        item   [2]   tIpAddress
                    };
                    ChoiceMap  DataField

                    void Translate(String strCurIdent,dynamic& curItem,long nIdxInArr)
                    {
                        sysTranChoiceItem(strCurIdent,curItem,this.ContentType);
                    }
                };
    ...
};
```

- *Translate()* is overrided, and *sysTranChoiceItem()* is called to decide which item in the *ChoiceMap* should be select and add to *Content* field (*sysTranChoiceItem* is called only once, because only one optional branch is valid here).
  For example, if this.ContentType=1, then tOctetStream will be selected to decode

following binary stream and added as a new branch, or if this.ContentType=2, then tIpAddress will be selected … (here, *this.Member* mean *Member* of parent structure)

Note: this is one of the samples about *choice* item, for details about *choice* item, refer to chapter 3.3.3

3.2.4 Define every child branch of this protocol

After defining root node, we will implement every child branches of this protocol.

3.2.4.1 Protocol branch: octet stream

1) Create hiberarchy structure

According to the Format of content field (octet stream), the hiberarchy of this protocol branche is shown as follows:

```
...
struct tOctetStream
{
    string strData;     // octet stream content
};
```

2) Implement fields

The field *strData* is a *string(octet)* field, unlike field with number type, we do not know the length of the *strData* in the beginning , so it should be decoded by overriding the virtual function *Translate()* (refer to …). And we dump it by default dump rule.

This field is coded as follows:

```
string strData {  // octet stream content
                  void Translate(String strCurIdent,dynamic& curItem,long nIdxInArr)
                  {
                      uint1    len;

                      sysFetchValue(strCurIdent, len, now(), 0, true);
                      sysFetchValue(strCurIdent, curItem, now(), len, true);

                      return;
                  }
              };
...
```

- *Translate()* is overrided. First, according to the format, we get the size of *curItem* (*strData*), then we fetch the character stream and store it in *curItem* (*strData*).

3.2.4.2 Protocol branch: ip address

1) Create hiberarchy structure

According to the Format of content field (IpAddress), the hiberarchy of this protocol branche is shown as follows:

```
...
struct tIpAddress
{
    octet    strIpAddress;
};
```

2) Implement fields

The field *strIpAddress* is a *octet* field with four bytes ipv4 address. We don't use *array* to declare this field, because we want to dump the address as one single element (for field with *array* attribute except *Tchar*, every element in the *array* is dumped seperately). Because the software do not know the length of the *strIpAddress* in the beginning , we should override the virtual function *Translate()* and *dump()*.
This field is coded as follows:

```
string strIpAddress {
        void Translate(String strCurIdent,dynamic& curItem,long nIdxInArr)
        {
            sysFetchValue(strCurIdent,curItem,now(),4, true);
        }
        String Dump(dynamic& curItem)
        {
            int   i;
            BYTE Address[4];

            for(i=0;i<4;i=i+1)
            {
                strGetAt(this.strIpAddress, i, Address[i]);
            }
            return strFormat("%d.%d.%d.%d",
                             Address[0],Address[1],Address[2],Address[3]);
        }
    };
...
```

- *Translate()* is overrided. According to the format, we get four bytes (ipv4 address), and store it in *curItem* (*strIpAddress*).
  Dump() is overrided to display the data of *strIpAddress* as the format we usually show (such as "127.0.0.1").

3.2.4.3 Protocol branch: BCD Stream

1) Create hiberarchy structure

According to the Format of content field (Positive BCD), the hiberarchy of this protocol branche is shown as follows:

```
...
struct tBCDStream
{
    string   strPhoneNumber;
};
```

2) Implement fields

The field *strPhoneNumber* is a *user-defined* transformed field, the string of the *strPhoneNumber* is transferred as BCD-octet in the binary stream. When decoding, we should convert the BCD-octet to phone number string. In this demonstration, we override the virtual function *Translate()* and call a global function *BCD2STR()* to implement the convertion.

This field is coded as follows:

```
string strPhoneNumber {
        void Translate(String strCurIdent,dynamic& curItem,long nIdxInArr)
        {
            UINT1     PhoneNumberLen;
            OCTET     strBCDData;

            sysFetchValue(strCurIdent,PhoneNumberLen, now(),0,  true);
            sysFetchValue(strCurIdent,strBCDData,now(), (PhoneNumberLen+1)/2, true);

            curItem=BCD2STR(strBCDData, true, (PhoneNumberLen%2)? 0:1);
            return;
        }
    };
...
```

- *Translate()* is overrided. According to the format, first we get the length of the phone number (total digits number), calculate the length of the BCD-octet that equal to *(PhoneNumberLen+1)/2*, then fetch the BCD-octet and store it in *strBCDData*. After that we call *BCD2STR()* to convert BCD data to visiable string.
  Sample binary stream is given in the demonstration, you can load it and see the result generated by the *Protocol Reader*.

3.2.4.4 Protocol branch: string with typeof attribute

1) Create hiberarchy structure

According to the Format of content field (String with TypeOF attribute), the hiberarchy of this protocol branche is shown as follows:

```
...
struct tTypeOFString
{
    string  strData  { TypeOF  tStringWithLenAndZeroEnd };  // TypeOf string
};
```

2) Implement fields

The field *strData* demonstrate *Typeof* attribute (For details about why and how to use syntax element ' *TypeOf* ', please see Chapter. Struct with TypeOF attribute). We provide two solutions to implement *tStringWithLenAndZeroEnd*, they are in common use in script development.

This field is coded as follows:

Solution 1: In this solution, we first decode all the members of '*tStringWithLenAndZeroEnd*', then call *Integrate()* to assemble and calculate these members to generate a single value which represent the structure itself.

When displaying, by toggling the button in Display format Toolbar, all the detailed members in *TypeOF*-structure could be shown (such as *Len*, *strData* and *Zero*).

```
struct tStringWithLenAndZeroEnd = String
{
    UINT1       Len;
    String      strData {     // string content
                  void Translate(String strCurIdent,dynamic& curItem,long nIdxInArr)
                    {
                        sysFetchValue(strCurIdent,curItem,now(),this.len,true);
                    }
                };
    UINT1       Zero;
};
attribute name='tStringWithLenAndZeroEnd'
{
    String Integrate(String strCurIdent)
    {
        if (this.Zero!=0)
          sysAbort(strCurIdent,"string not end with zero");

        return this.strData;
    }
};
```

Solution 2: In this solution, the struct '*tStringWithLenAndZeroEnd*' is null, all the decoding process is in function *Integrate()* .

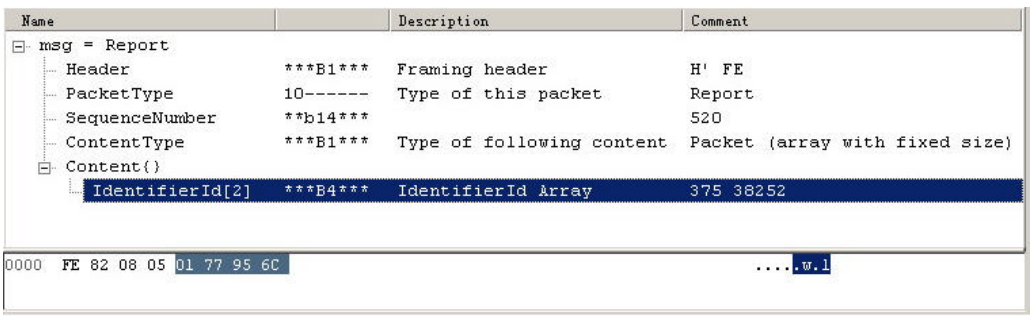
```
struct tStringWithLenAndZeroEnd = String {};

attribute name='tStringWithLenAndZeroEnd'
{
    String Integrate(String strCurIdent)
    {
        uint1   Len;
        String  strData;
        uint1   Zero;

        sysFetchValue(strCurIdent,len,      now(),0,   true);
        sysFetchValue(strCurIdent,strData,now(),len, true);
        sysFetchValue(strCurIdent,Zero,   now(),0,   true);

        if (Zero!=0)
          sysAbort(strCurIdent,"string not end with zero");

        return strData;
    }
};
```

3.2.4.5 Protocol branch: fixed size array

According to the Format of content field (Fixed size array), this protocol branche is coded as follows:

```
...

struct tFixedArray
{
    uint2   IdentifierId[2];          // IdentifierId Array
};
```

The field *IdentifierId* is a *number* field with array attribute, it is decoded by default decode process (refer to Table 3.1) and dumped by default dump rule. Because the datatype occupy more than one byte, the *NETWORK_TRANMODE* is considered when fetch data from byte stream (we define this constant at the begin of the script here, or you can define it in its own attribute-block of this field for specific needs).



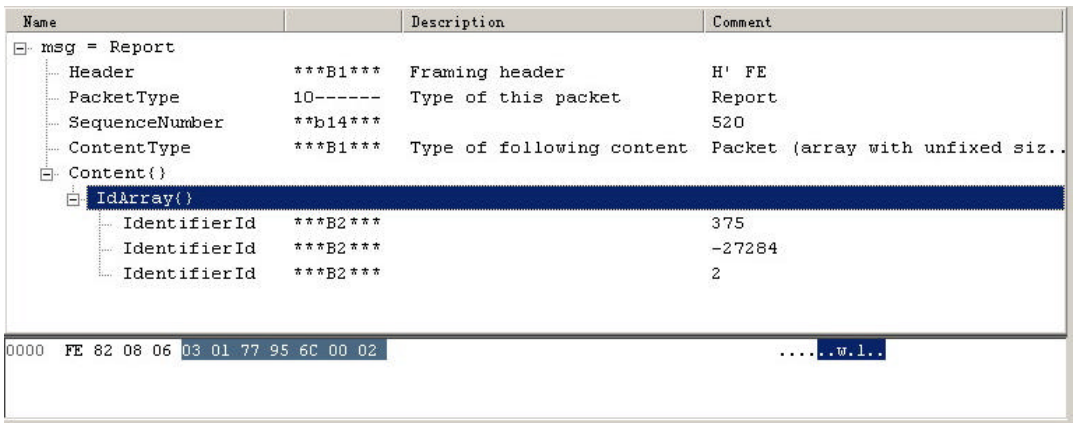
3.2.4.6 Protocol branch: unfixed size array

According to the Format of content field (unfixed size array), this protocol branche is coded as follows:

```
...
struct tUnFixedArray
{
    choice  IdArray {
                DeclareMap( Item , DataField )  {
                    IdentifierId  [0]  int2
                };
                ChoiceMap  DataField

                byte  Num;
                sysFetchValue("IdArray", Num, now(), 0, true);

                void Translate(String strCurIdent,dynamic& curItem,long nIdxInArr)
                {
                    int k=0;
                    while (k<Num) {
                        sysTranChoiceItem(strCurIdent,curItem,0 );
                        k=k+1;
                     }
                }
    };
};
```

The field *IdArray* is a *number* field with unfixed array attribute, and could be implemented as a *Choice* field. According to the format, fisrt, get the identifier number and store it in *Num*, *Translate()* is overrided and *while* statement is called to add certain number (specified by *Num*) of IdentifierId to *IdArray*.

Following is a sample result generated by *Protocol Reader* :



3.2.4.7 Protocol branch: timestamp

This field (see Format of content field: timestamp) is a long integer that store the timestamp value in *timer* represents a date from midnight, January 1, 1970. When dumping, we want to display the value using the format like "yyyy-mm-dd hh:mm:ss".

How to implement it ?

Since *Protocol Reader* is not intended to be a *ANSI* C compilor, we just bring in the most necesssary parts of *ANSI* C's syntax to keep a more simple kernel and we do not support standard C library at the beginning (besides essential functions we declare in the "*sys.h*"), furthermore, it is unnecessary to support all standard C library in *Protocol Reader*. So, the problem is how to implement new functions ?
Considering furtune use, we give two fundamental ways of implement new functions:

- if the function is simple, you can declare and implement it as a global function (refer to sample of *BCD2STR()* to create new functions).
- if the function is complex and it is difficult to code it directly in *Protocol Reader* (such as convert the value of timestamp to format "yyyy-mm-dd hh:mm:ss"), we use plug-in techneque to create such function: declare it as an external function in script and implement it in the plug-in project. By this way, we can share most of the ready-made functions and librarys that C or C++ compilors provide instead of implementing it by ourself.
  There are two kind of plug-in library in *Protocol Reader*, one is *public library* which could shared by any protocol script and is loaded when the software is launched, the other is *private library* only called by the specific protocol script and is loaded when certain protocol script is loaded.
  For details about plug-in development, see Chapter Plug-in Developing Guide, and template project is also provide in software installation fold.

In this session, we will describe how to add a shared function to the *public library*.

i . We should first declare the shared function as an external function, and include the function header in the script.

```
// Filename: lib.h
# include "..\Include\sys.h"
……
extern string TimestampToDateStream(long timestamp);
```

ii . Second, load the template project: \ProjectPlugIn with Visual C++ software, open the File \ProjectPlugIn\PlugInObject.h, register the procedure and declare the procedure as one of the *CPugIn* member.

```
class CPlugIn: public CPlugInbase
{
      ……
      // Add your Function Header here
      static DWORD TimestampToDateStream(void* pParent, unsigned int params_num,
                                         DWORD params[] );
};

// Register your PlugIn Function
static struct FuncLookupTableEnt Tbl_FuncList[] =
{
    { "TimestampToDateStream",      &CPlugIn::TimestampToDateStream },
    { 0,                            NULL }
};
```

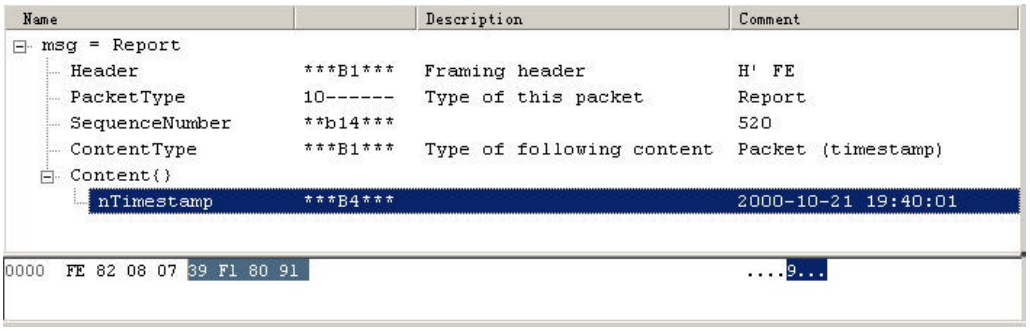
iii. Then, open the File \ProjectPlugIn\PlugInObject.cpp, add implement codes for the plug in procedure "*TimestampToDataStream()*"

```
// Add your function implement codes here !

DWORD CPlugIn::TimestampToDateStream  (void* pParent, unsigned int params_num,
                                       DWORD params[] )
{
      if (params_num<4) return 0;

      ExecuteEnv&    Env        =*((ExecuteEnv*)params[1]);   // current environment
      Procedure*     pProcedure =  (Procedure* )params[2];    // current procedure
      BinStream*     pstream    =  (BinStream* )params[3];    // current binary stream
      RunTimeStk&    curStk     =*(Env.GetTopStk());          // current stack in environment

      //-----------------------------------------------
      bool   bRT=true;

      //-----------------------------------------------
      // GET PROCEDURE PARAMETER

      long   nTimestamp;

      bRT=bRT & curStk.get("timestamp", nTimestamp);
      if (!bRT)
      {
          Env.SetLastError("%s() get param fail",pProcedure->GetProcName().GetBuffer());
          return 0;          // error
      }

      //-----------------------------------------------
      // EXECUTE PROCEDURE

      if (nTimestamp<0)
      {
            Env.SetLastError( "%s() value of timestamp error",
                              pProcedure->GetProcName().GetBuffer());
          return 0;  // error
      }
```

```
        OTSTR   str;
        time_t sec     = nTimestamp;
        tm* cur_time = localtime(&sec);       // Use standard C   function
        str.Format( "%04d-%02d-%02d %02d:%02d:%02d",
                    cur_time->tm_year+1900 ,
                    cur_time->tm_mon+1,
                    cur_time->tm_mday,
                    cur_time->tm_hour,
                    cur_time->tm_min,
                    cur_time->tm_sec);

        //-----------------------------------------------
        // SET RETURN VALUE

        CVarient Varient;
        Varient.set(str);

        if (!pProcedure->setReturn(Env, Varient))
        {
              Env.SetLastError("%s() set return fail",pProcedure->GetProcName().GetBuffer());
              return 0;    // error
        }

        return 1;
  }
```

iv. After coding, compile the source code, generate plug-in library file named *.dl_, and copy it to the installation fold\PlugIns, it will go into effect next time the *Protocol Reader* loaded. It is strongly recommended that you do some test for this newly added function before you release this plug-in.

Following is a sample result generated by *Protocol Reader* :



Note: For un-licensed version, plug-in feature (number of plug-in, total number of functions in every plug-in) is limited

3.2.5 Summary

In this session, we have presented an approach about how to develop a script step-by-step, we hope that it's helpful to you and could speed up the learning process. But please remember that what we provide you is a powerful develop *platform*, there is no "one right way" to design and build script on that *platform*, it's all up to you. Since we believe strongly that the way to learn new techneque is to do more practice , more experiments is needed before you could create an excellent protocol script concisely and quickly.

3.3 Advanced topics

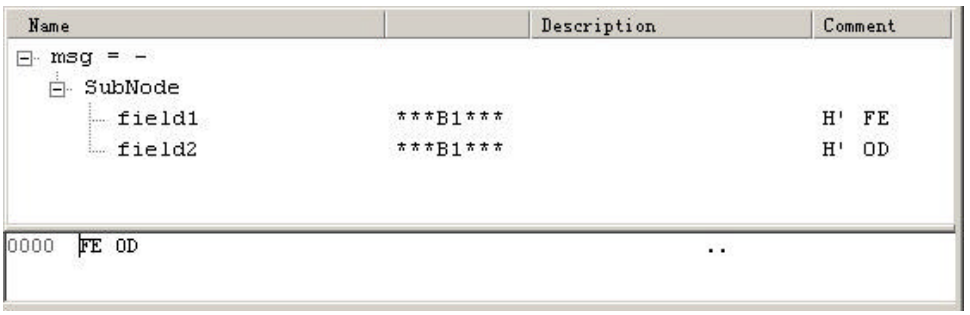
In this chapter, we will discuss some of the important component when developing a script.

3.3.1 Field declared as structure

Most protocols are multi-level. How can we express the parent/child relationships between protocols and fields ? we introduce *structure* to organize the hiberarchy of the protocol, every structure node represent one level or a child branch in the protocol. The syntax is the same as that in *ANSI C*. Let's see an example:

```
// definition

Struct    strucA
{
      byte    field1;
      byte    field2;
};

// sample 1:
Struct msg
{
       strucA    subNode;
       ……
};

// sample 2: array
Struct msg
{
       strucA    subNode[2];
       ……
};
```

We give a sample result generated by the *Protocol Reader* for sample1. you can see the tree of protocol hiberarchy from the figure:



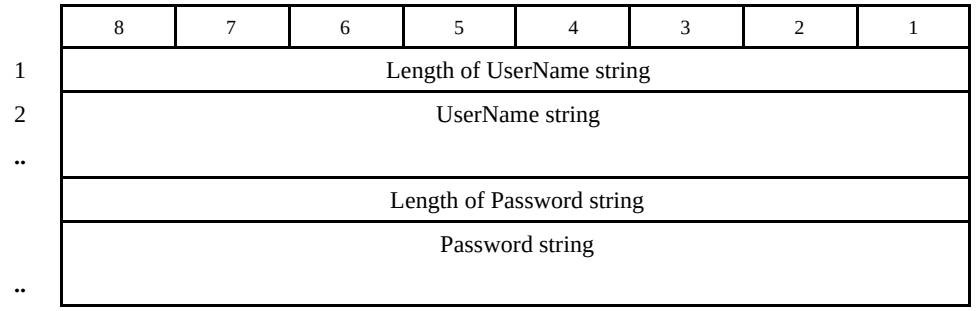
3.3.2 Structure with TypeOf attribute

The *TypeOf* is a special syntax element in the *Protocol Reader*, it is very useful when dealing with shared components or some complicated data.
Let's see its syntax first:

```
// predeclaration if needed
struct struct_Identifier = returntype;
……

// declaration
struct struct_Identifier = returntype
{
    // Note: for structure with typeof attribute, field definition
    // is not mandatory
    datatype  field_identifier {  // field_description

                decode attribute
                dump attribute      } attribute_block(optional)
            };
    ……
};
attribute name = 'struct_Identifier'
{
    returntype Integrate(string strCurIdent)
    {
        ……
    }
    string Dump(dynamic& curItem)  // optional
    {
        ……
    }
};


// usage
struct struct_name
{
    ……
    datatype  field_name {TypeOf struct_Identifier };  //description
              // Note: here, the datatype = returntype
    ……
};
```

The structure with *TypeOf* attribute return a single value (only *basic type* is allowed) to represent itself. All the members of the structure will be organized and calculated to generate a single value.
It is useful and make your script more compact and easy to maintenance. The following example will help you to understand the benefit of *TypeOf*.

Figure 3.4.2 Message format



1) Script A: do not use TypeOf element

```
Struct   loginInfo
{
      string    strUserName{
                        void Translate(String strCurIdent,dynamic& curItem,long nIdxInArr)
                        {
                              byte length;
                              sysFetchValue(strCurIdent, length, now(), 0,     true );
                              sysFetchValue(strCurIdent,curItem,now(),length, true);
                        }
                  };
      string    strPassWord{
                        void Translate(String strCurIdent,dynamic& curItem,long nIdxInArr)
                        {
                              byte length;
                              sysFetchValue(strCurIdent, length, now(), 0,     true );
                              sysFetchValue(strCurIdent,curItem,now(),length, true);
                        }
                  };
};
```

2) Script B: use TypeOf element

First code shared component, and save it to file "*share.h*".

```
# include "..\Include\sys.h"

Struct   tString = string ;  // define shared component
{
      byte       length;
      string   strData {
                        void Translate(String strCurIdent,dynamic& curItem,long nIdxInArr)
                        {
                              sysFetchValue(strCurIdent,curItem,now(), this.length, true);
                        }
                  };
};
attribute name = 'tString'
{
      string Integrate(string strCurIdent)
      {
            return  this.strData;        // return a value for this structure
      }
};
```

Then code our main protocol script:

```
...
# include "share.h"

Struct  loginInfo
{
    string  strUserName  {  TypeOf  tString  };
    string  strPassWord  {  TypeOf  tString  };
};
```

Compare with ScriptA and ScriptB, we can see that the later one is more compact and the shared component is more easy to maintenance. We recommend you to make the shared components more independent when developing script just as the example show you.

When displaying element with *TypeOf* attribute, there is a toggle in Display format Toolbar. If the button is in push-down status, all the detailed members in *TypeOF*-structure is showed, otherwise they will be hided to make the display more compact. *Protocol Reader* does this for convenience, when display, it keep the protocol tree from having unwanted details.

Here, we will give you another way to implement *tString*, and you could use any one of them for your Preference. Inputing a byte stream and using *Protocol Reader* to display the result will show you the difference between them.

```
# include "..\Include\sys.h"

Struct  tString = string ;  // define shared component
{
                        // no  member  introduced here !
};
attribute name = 'tString'
{
    string Integrate(string strCurIdent)
    {
        byte        length;
        string      strData;
        sysFetchValue(strCurIdent,  length,  now(), 0,     true );
        sysFetchValue(strCurIdent,  strData, now(),length, true);
        return strData;      // return a value for this structure
    }
};
```

3.3.3 Field with choice attribute

The *Choice* is a frequently used syntax element in the *Protocol Reader*, it provide another way to organize the hiberarchy of the protocol. If the datatype of a field is *Choice*, meaning that there has one or more *optional* selection here.

To implement field with this kind of type, you should follow these steps:

Step 1: Use *DeclareMap* to list all possiable choices (branches)

Step 2: override virtual function *Translate()* to implement how to process optional items.

We have shown you a sample about the implement of *Choice* field (refer to Field:Content), and here, further discussion will be given:

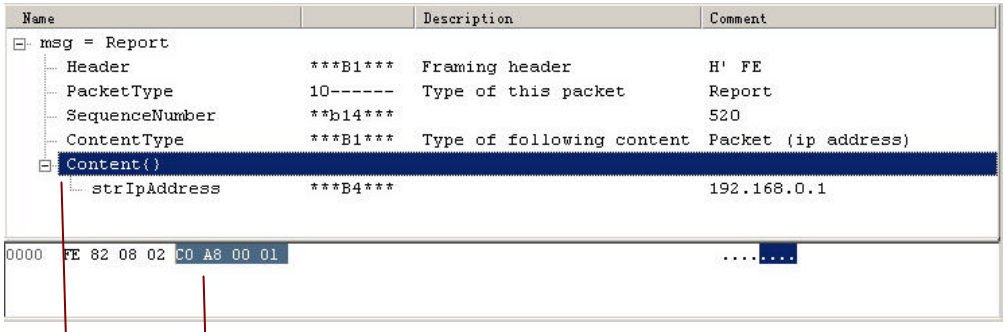
1) Name of *choice* field

The syntax is as follows:

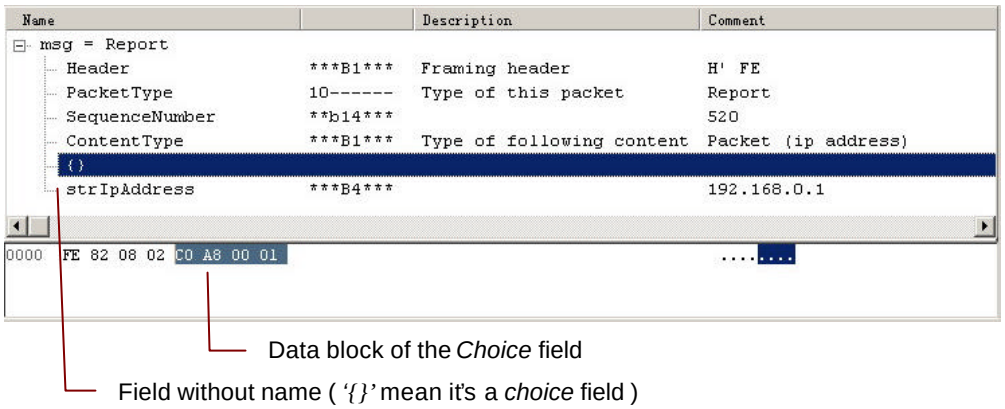
```
Struct structName
{
    ……
    Choice [fieldname] {  // field_description (begin)
                ……        //  attribute-block (must be exist)
          };   // field_description (end)
    ……
}
```

For *Choice* field, the field name is optional. It don't need to be an entity of the protocol, and when the name of the *Choice* field is not exist, it's only used to organize all optional data, just as a virtual container.

Sample1: *Choice* field with name



Sample2: *Choice* field without name



Note: To refer to the item in Choice content, the form *structure-name.member* could be used. For example, if we want to refer to *strIpAddress* in the sample (see the figure above):
For sample1: expression will be *msg.Content.strIpAddress*
For sample2: expression will be *msg.strIpAddress*

2) DeclareMap

DeclareMap is another special syntax element we add to declare and organize all the optional items of *Choice* field. The syntax is as follows:

```
DeclareMap(Item|ImplicitItem, MapName)
{
     variablename [nTag] basictype
     variablename [nTag] TypeOf struct_Idendifier
     variablename [nTag] struct_Idendifier
}
```

The attribute of a *DeclareMap* can be *Item* or *ImplicitItem*. For each choice item, a relevant and exclusive Identifier ID (*nTag*) is given.

- When *DeclareMap* has *Item* attribute
  If an optional item is exist, the corresponding node or leaf (named *variablename*) would be added to current *Choice* field of the protocol tree.
- When *DeclareMap* has *ImplicitItem* attribute
  This attribute is only valid for variable with structure attribute, for variable with other data type, it is equal to *Item*.
  If an optional item is exist, only the members of the corresponding structure (named *struct_Idendifier*) would be added to current *Choice* field of the protocol tree.
  In this circumstance, the structure is only used to organize the sequence of optional items which appear at the same time, and the name of the structure item itself (*variablename*) is hided, because it is not an element of the protocol tree (only the members of the structure is needed for the protocol tree).

Here, we will give you two example to see the difference:

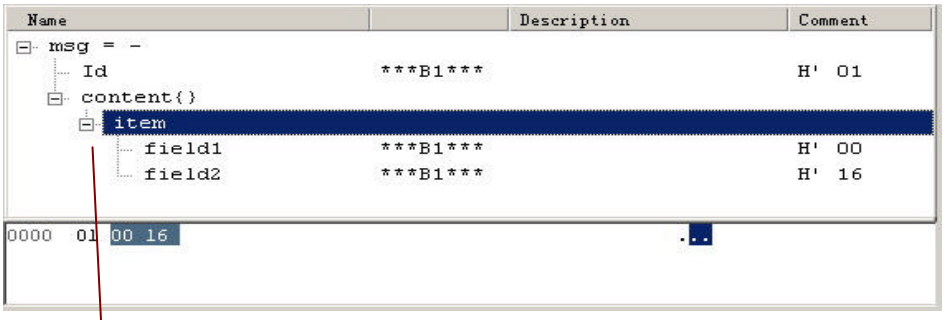
Sample1: *DeclareMap* with *Item* attribute

a) demo script

```
… …
struct  strucA
{
     byte    field1;
     byte    field2;
};

struct msg
{
     byte      Id;
```

```
    choice   content {
                    DeclareMap( Item , DataField )
                    {
                        item   [1]   strucA
                    };
                    ChoiceMap  DataField

                    void  Translate(String strCurIdent,dynamic& curItem,long nIdxInArr)
                    {
                        sysTranChoiceItem(strCurIdent,curItem,this.Id);
                    }
    };
};
```

b) demo result generate by *Protocol Reader*

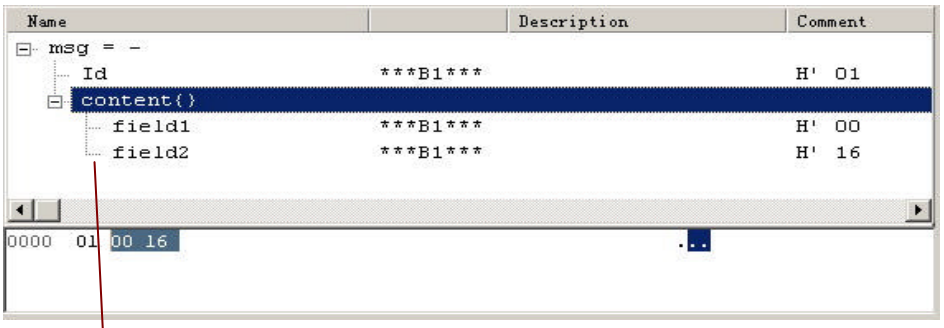


Sample2: *DeclareMap* with *ImplicitItem* attribute

a) demo script

```
... ...
struct  strucA
{
    byte    field1;
    byte    field2;
};
struct msg
{
    byte      Id;
    choice   content {
                    DeclareMap( ImplicitItem , DataField )
                    {
                        item   [1]   strucA
                    };
                    ChoiceMap  DataField
                   void  Translate(String  strCurIdent,dynamic&  curItem,long  nIdxInArr)
                    {
                        sysTranChoiceItem(strCurIdent,curItem,this.Id);
                    }
    };
};
```

b) demo result generate by *Protocol Reader*



Optional content with *ImplicitItem* attribute

(Only the members *field1*, *field2* of *strucA* are added to *content*)

3.3.4 Field with array attribute

To support field with array attribute is necessary in some cases. The process for this kind of field has some differences from that of others.

1) The form of array with fixed size is as follows:

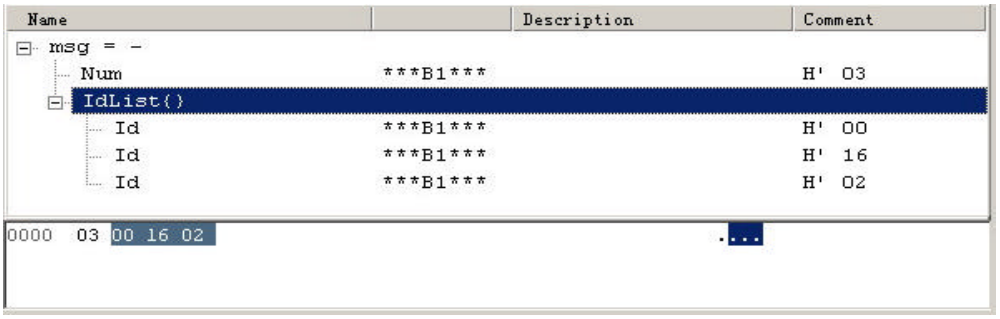
```
-------------------------------------------------------------
struct identifier_name          // struct_description
{
    ……
    datatype  field_identifier[N] { // field_description

              decode attribute   }attribute_block(optional), would be
              dump attribute     }executed for each element of the array
          };
}
-------------------------------------------------------------
```

2) The array with unfixed size could be implemented as a *Choice* field

Let's see a piece of code demonstrate array with unfixed size:

```
struct msg
{
      byte      Num;
     choice   IdList {
                    DeclareMap( Item , DataField )   {
                          Id   [0]   byte
                    };
                   ChoiceMap  DataField
                    void Translate(String strCurIdent,dynamic& curItem,long nIdxInArr)
                    {
                          int k=0;
                          while (k<this.Num) {
                               sysTranChoiceItem(strCurIdent,curItem,0 );
                               k=k+1;
                           }
                    }
      };
};
```

In this sample, we would like to get a *Id* list with dynamic size (determined by *Num*), Following is a result generate by *Protocol Reader*.



3.4 Debug the script

A script that has not been tested does not work. The ideal of designing and/or verifying a piece of codes so that it works the first time is unattainable for all but the most trivial programs. ‘‘How to test?’’ is a question that cannot be answered in general. ‘‘When to test?’’ however, does have a general answer: as early and as often as possible.

- When to test

Testing should begin as early as possiable, so, you don't need to write whole hiberarchy nodes or codes at the very begin when protocol is complex, unless you have a good understanding of using this software.

Actually, you could implement the protocol level by level or field by field, this method is often used when you are not sure about the correctness of your script and try to debug the newly writed code.

- How to test

In *Protocol Reader*, insteading of dealing with protocol itself, we *focus our workload on individual field* of the protocol, any field is relatively independent (coded like Figure 3.2). To separate the complex protocol into smaller one help to simplify the problem, and make it easier to test and save your valuable time. So, the testing of a field would be the primary problem.

When debug a newly coded field, you could input a hexadecimal message, decode it and check if the result is correct. Because there is no “one right way” to design and build a script, you could compare different approaches of implementing a field to see different result generated by the *Protocol Reader*, it will help you to understand this software and get a better way of implementing a field. Usually, compilers will warn of most errors.

Write a field → test it → write another field …, this will help you write a correct script.

Since we believe that the way to learn new techneque is to do more practice , more experiments is needed before you could create an excellent protocol script concisely and quickly. Besides, it is strongly recommended that you open and study the sample script, it will give you a lot of help.

3.5 Appendix

In *Protocol Reader*, default decode (see Table3.1) and dump rule (see Table3.2) for specific datatype is provided to fit most need; if there are no special requirements, you don't need to write any codes in attribute block, system will do all that for you. Or you can override system reserved virtual function (see Table 3.3) to meet certain needs.

3.5.1 Default decode rule

Table 3.1 Default decode process

Type	*Default process*	*Remark*
Int1, uint1 char, unsigned char, byte, bool	Get data (1 byte)	
Int, int2,uint2, short, unsigned short, word	Get data (2 byte)	1. The order of byte stream is defined by *NETWORK_TRANMODE*, if this const is not present, default value *L_TO_H* will be used
Int4, uint4, long, unsinged long, dword	Get data (4 byte)	
Float, double	-	
String	-	
Octet	-	
Tchar	Get byte stream	
Memo	-	
Choice	-	

- When there are some special requirements or *default process* is not valid for certern data-type, one of the *system reserved decode process* should be overrided (shown in Table 3.3)

3.5.2 Default dump format

Table 3.2 Default dump format

Type	*Internal Type*	*Default dump format*
Int, int1, int2, int4, char, short, long	signed dec integer	%ld

Uint1,uint2,uint4, unsigned char, unsigned short, unsinged long, bool	Unsigned dec integer	%lu
Byte, word , dword	Unsigned hex integer	H' %x
Float, double	Double	%.3f
String	Ascii string	T' %s T mean null
Octet	octet stream	H' %02x %02x …
Tchar	Fixed array of char	T' … (character '\0'->'.')
Memo	Large octet stream	M' Length=...

- If there are some specific requirements when dumping a decoded data of a field, the virtual function ***string Dump()*** could be overrided to meet you need.

3.5.3 System reserved process for override

Table 3.3 System reserved process for override

Type	*Valid decode process*	*Corresponding dump process*
All number type	1. Default process 2. *TypeOF* 3. *Translate ()*	Default process or *Dump()*
Float, double		-
String		Default process or *Dump()*
Octet		
Tchar		
Memo	1. *Translate ()* *ContentOF (optional)*	- Default process or *Dump()*
Choice	1. void *Translate ()*	-
Struct with *TypeOf* attribute	1. *Integrate ()*	Default process or *Dump()*

There are three decode process mode you can override

1) ***Translate()***

This function is commonly used in most circumstances, you can write you own decode process in this function, and there are two return mode: if the funciton is void, the returned data is stored in *curItem*; or if the function has return-type (must be same as the type of current field), you can use *return* to return decoded data.

Form of this function is as follows:

```
type Translate (string strCurIdent,dynamic& curItem,long nIdxInArr);
```

Parameters

strCurIdent	current protocol field name
curItem	current protocol field to decode if it's an array field, curItem pointer to ThisField[nIdxInArr]
nIdxInArr	index in ThisField (always equal to 0 for un-array field)

Return

If the return type is not a *void* data-type, return decoded data.

2) ***TypeOf***

The *TypeOf* is a special syntax element in the *Protocol Reader*, it is very useful when dealing with shared components or some complicated data. We will give you detailed information in …

3) ***ContentOf***

This keyword is only used for the field with *memo* attribute. For this kind of field, function *Translate ()* is mandatory, and after *Translate ()* return the decoded byte-stream, you can use *ContentOf* to further decode returned byte-stream, the syntax is as follows:

```
ContentOf struct_Identifier
```

The usage of this keyword is somewhat like that of *sysTranChoiceItem*(), the difference between them is when you use *ContentOf*, the further decoding is limited in current stream returned by *Translate ()*, if overflow occur, system will give you an error message.

4 Plug-in Developing Guide

TO BE CONTINUED ……