frontline™

DecoderScript™

Reference Manual and User Guide

Copyright © 2001-2009 Frontline Test Equipment, Inc. All rights reserved.

DecoderScript is a trademark of Frontline Test Equipment, Inc.

UltraEdit is a registered trademark of IDM Computer Solutions, Inc.
Visual C++.NET is a registered trademark of Microsoft Corp.

All other trademarks and registered trademarks belong to their respective owners.

Table of Contents

Chapter 1: Overview 1

Chapter 2: How the Protocol Analyzer Processes Captured Data 2

Protocol Layers 2
Autotraversal 3
The Processing Sequence 4

Chapter 3: Structure of a Decoder 7

What You Need 8
Basic Structure 10
The Header Section 10
The Reference Section 11
DECODE 11
The Table Section 12
The Executable Section 12
Fictitious Protocol 13
The Basic Structure of a Decoder File 16
The FIELD Statement 17
General Syntax Rules 17
General Syntax: 18
Standard Methods 18
VERIFY Field Values 19
Creating Summary Pane Columns using IN_Summary 21
Tables 22
The GROUP Statement 24
Conditionals 27
The BRANCH Statement 28
SUPPRESS_DETAIL and NO_MOVE 32
The GROUP FIELD and RESERVED Statements 33
START_BIT 34
Comments 35
The FP Decoder 36

Chapter 4: Putting It All Together 40

Decoder IDs 40
Encrypted Decoders 40
Plain Text Decoders 40
[ProtocolName].personality files 40
Rules 41
Predefined Stacks 43
Port Assignments 43

Filters.......... 44
What to Send to End-Users.......... 45
Chapter 5: DecoderScript Reference 47
BYTE_STREAM_FRAMER.......... 47
FRAME_AFTER_DECODING.......... 47
FRAME_BEFORE_DECODING.......... 48
NEXT_PROTOCOL.......... 48
NEXT_PROTOCOL_SIZE.......... 48
NEXT_PROTOCOL_OFFSET.......... 48
PAYLOAD_IS_BYTE_STREAM.......... 48
PREPROCESSING.......... 49
POSTPROCESSING.......... 49
RECOGNIZER.......... 49
SUMMARY_COLUMNS.......... 49
FIELD.......... 53
FIELD Order of Statement Processing.......... 64
FIELD field_name;.......... 64
RESERVED.......... 65
GROUP.......... 65
REPEAT COUNT|SIZE [TRUNCATED_FIELD_OK]|UNTIL Method.......... 66
Detail Tag.......... 67
GROUP FIELD.......... 68
BRANCH.......... 70
PARAMETER.......... 71
INCLUDE.......... 73
END_MAIN_PATH.......... 73
Intra-frame and Inter-frame Data.......... 73
Intra-frame Data.......... 73
Inter-frame Data.......... 74
Chapter 6: Methods.......... 77
Boolean Methods.......... 77
Branch Methods.......... 77
Format Methods.......... 77
NextProtocol Methods.......... 78
NextProtocol.......... 78
NextProtocolOffset.......... 78
NextProtocolSize.......... 78
Retrieval Methods.......... 79
Size Methods.......... 79
StartBit Methods.......... 79
Tag Methods.......... 79
Verify Methods.......... 79

Checksum/CRC Methods 80
Field-Validation Methods 80
A Sample Method 80
The Method Context 81
Notation for Constant/Variable Names 82
CString and CArray Objects 82
Method Inputs 83
Redundancies in Input Values 89
Truncated Frames 90
The Current Field 90
Other Resources 90
Miscellaneous Functions 90
Defining a Method 92
Parameter Types 92
Output Values 93
Method Contexts and Examples 94
Boolean Methods 94
Branch Methods 95
Byte Stream Framer Methods 95
Format Methods 96
NextProtocol Methods 97
NextProtocolOffset and NextProtocolSize Methods 97
PreProcessing, Processing and PostProcessing Methods 98
Retrieval Methods 98
Size Methods 98
StartBit Methods 99
Tag Methods 99
Verify Methods 100
Programming Exceptions 101
Common 101
Where do Custom Methods reside? 102
How does a Custom Method get compiled? 102
Chapter 7: Frame Recognizers 103
Recognizer Syntax 103
Input Values 104
Return Values 105
Parameters 106
Error Processing 106
Efficiency 106
The Simplest Recognizer: HalfDuplex 107
A Frame Recognizer for FP 108
Handling Signal-Change Events 111
Handling Error (DRF, or Data Related Flag) Events 111

Handling Timer Events 112
Loose Ends 112
Frame Recognizers Included with the Protocol Analyzer 113

Chapter 8: Frame Transformers 115

Frame Transformers 115
A Frame Transformer for FP 116
Output Variables 118

Chapter 9: Decoder Data Objects 119

Understanding How The Decoding Process Affects Data Objects 119
Compilation Phase 120
Reconstruction Phase 120
Implications Of Phases 121
Ways to share data at different points of the decoding process 121
Decoder Data Object Syntax 122
Variables Available to Decoder Data Objects 123
Using Decoder Data Objects for per-Frame Data 125
A class to make saving and retrieving easier and more compact 127
Error Processing 128
Providing a User Interface to a Decoder Data Object 128
What the User Sees 129
Code Blocks 129
Writing Methods That Use Decoder Data Objects 132
A Simple Example 133
A Complex Example 136

Chapter 10: Tips and Tricks 141

Approach to Decoding a Field 141
Empty Frames 141
Red items in the Summary Pane 141
How Field Values are stored 141
A subtle difference between GROUP and GROUP FIELD 142
Large Fields 142
Error correcting in GROUP REPEAT constructions 142
How to implement a repeating counter. 143
Recalcitrant Summary Columns 144
Overwriting Summary-Pane Column Entries 144
Readable Protocols 145
Using Variables 145
Storing Data 146
Storing Data into Decoder Data Objects 146

Chapter 11: One complete example 147

Chapter 12: A Formal Grammar for DecoderScript 158

Appendix 1: Decoder Summary.................................. 161
Appendix 2: Using the Protocol Analyzer with the Option for Creating Decoders.. 162
Reload Decoders 162
Building Methods 162
How to Insert Fictitious Protocol into a Fictitious Product.. 162
Appendix 3: Keyword List 166
Appendix 4: Creating Custom Frame Recognizer Return Values 171
Appendix 5: Method Reference 173
Group: BOOLEAN 173
Method: Check_CrcCCITTrevDataBytes... 173
Method: CompareFields 173
Method: CurrentDataMatches 174
Method: Dce 174
Method: Dte... 175
Method: FieldIs 175
Method: FieldIsBetween.. 175
Method: FieldLengthIs 176
Method: IndexedPersistentStaticExists 176
Method: IntraframeStaticExists 177
Method: IsInTable 177
Method: IsOdd 177
Method: IsProbablyAtType 178
Method: MatchTableData 178
Method: MoreBytes... 179
Method: MoreBytesInSegment 179
Method: MoreBytesToDecode 180
Group: Branch 182
Method: DTEorDCE 182
Method: FromTable 182
Method: PersistentStaticExists 183
Group: FORMAT 184
Method: Binary 184
Method: Constant.. 184
Method: Decimal 185
Method: DecimalWithFieldLabel 185
Method: DosDate... 186
Method: DosTime 186
Method: Double... 187

Method: Float ... 187
Method: Hex ... 187
Method: IpAddress ... 188
Method: MacAddress ... 188
Method: MillisecondsFrom1970 ... 188
Method: NetBiosName ... 189
Method: Octal ... 189
Method: OtherField ... 190
Method: Scale ... 190
Method: SecondsFrom1970 ... 190
Method: StringOfAscii ... 191
Method: StringOfBCD ... 191
Method: StringOfBinary ... 191
Method: StringOfDecimal ... 192
Method: StringOfHex ... 192
Method: StringOfUTF8 ... 192
Method: StringOfUnicode ... 193
Method: Table ... 193
Method: UUID ... 194
Group: FRAME_TRANSFORMER ... 195
Method: RealignOnByteBoundary ... 195
Method: StripDoubledChar ... 195
Group: NEXT_PROTOCOL ... 196
Method: FromField ... 196
Method: FromPortAssignment ... 196
Method: IsAlways ... 196
Group: NEXT_PROTOCOL_SIZE ... 197
Method: FromField ... 197
Method: FromFieldIfLessThan ... 197
Method: OffsetToEnd ... 197
Method: OffsetToEndTakesFieldParam ... 198
Method: ThisLayerLength ... 198
Group: POSTPROCESSING ... 199
Group: PREPROCESSING ... 200
Method: InitField ... 200
Group: PROCESSING ... 201
Group: RETRIEVING_DATA ... 202
Method: AddField ... 202
Method: AddFieldToField ... 202
Method: AddFieldToInteger ... 202
Method: AddInteger ... 203
Method: AddIntegerToField ... 203
Method: AndMask ... 203

Method: BitOffset 204
Method: ClearPersistentField 204
Method: DivideField 204
Method: DivideFieldByField 204
Method: DivideFieldByInteger 205
Method: DivideInteger 205
Method: DivideIntegerByField 205
Method: FromFlag 206
Method: FromFlags 206
Method: IndexedPersistentField 206
Method: IndexedPersistentFieldAndDelete 207
Method: IntraframeField 207
Method: ModuloInteger 207
Method: ModuloField 208
Method: ModuloFieldByField 208
Method: ModuloFieldByInteger 208
Method: ModuloIntegerByField 209
Method: MultiplyField 209
Method: MultiplyFieldByField 209
Method: MultiplyFieldByInteger 209
Method: MultiplyInteger 210
Method: MultiplyIntegerByField 210
Method: OrMask 210
Method: PersistentField 211
Method: ReplaceWithExtraData 211
Method: ShiftLeft 212
Method: ShiftRight 212
Method: SignExtension 212
Method: SplitField 213
Method: StoreField 213
Method: StoreIndexedPersistentField 213
Method: StoreInteger 215
Method: StoreIntraframeField 215
Method: StorePersistentField 215
Method: SubtractField 216
Method: SubtractFieldMinusField 216
Method: SubtractFieldMinusInteger 217
Method: SubtractInteger 217
Method: SubtractIntegerMinusField 217
Group: SIZE 218
Method: ArrayOf 218
Method: Bit 218
Method: Byte 218

Method: Fixed 218
Method: FixedUpTo 219
Method: FromExtraData 220
Method: FromField 221
Method: MinFieldOrEndOfLayer 222
Method: OddTerminated 223
Method: OneToken 223
Method: RelativeFromField 223
Method: String 224
Method: ToEndOfLayer 224
Method: ToTerminatingValue 225
Method: UnicodeString 225
Method: UpTo 225
Method: Zero 226
Group: START_BIT 228
Method: FromEnd 228
Method: FromEndSizeFromField 228
Method: FromStart 229
Method: Move 230
Method: MoveAbsoluteFromField 230
Method: MoveRelativeFromField 231
Method: MoveToField 231
Method: MoveToFieldPlusOffset 231
Method: NextByteBoundary 232
Method: NextWordBoundary 232
Method: PreviousByteBoundary 232
Group: TAG 233
Method: Tag 233
Group: VERIFY 234
Method: CheckCrc802153MacHcs 234
Method: Check_CrcCCITTrevDataBytes 235
Method: Check_CrcCRC16 235
Method: Check_CrcCRC16RevAndInvert 235
Method: Check_CrcCRC16rev 236
Method: Check_CrcCRC32 236
Method: Check_CrcHDLC 236
Method: Check_CrcHDLCerr 237
Method: Check_CrcLRC 237
Method: Check_CrcModBus 237
Method: Check_CrcModBus_LRC 238
Method: Check_CrcSum 238
Method: Check_CrcSum1comp 238
Method: Check_CrcSum2comp 239

Method: Check_CrcXOR1comp 239
Method: Check_CrcXOR2comp 239
Method: Fail 240
Method: FieldIs 240
Method: FieldIsBetween 241
Method: FieldLengthIs 241
Method: HdlcFcs 242
Method: IsInTable 242
Method: IsLengthOfLayer 242
Method: MatchTableData 243
Method: TcpChecksum 244
Glossary 245
Index 248

Chapter 1: Overview

The main purpose of this manual is to describe DecoderScript™, the language used in writing decoders. DecoderScript allows you to create new decoders or modify existing decoders to expand the functionality of your FTS protocol analyzer. DecoderScript displays protocol data, checks the values of fields, validates checksums, converts and combines field values for convenient presentation. Decoders can also be augmented with custom C++-coded functions, called "methods", to extend data formatting, validation, transformations, and so on.

A decoder defines field-by-field how a protocol message can be taken apart and displayed. The core of each "decoder" is a program that defines how the protocol data is broken up into fields and displayed in the Frame Display window of the analyzer software.

This manual provides instruction on how to create and use custom decoders. When reading the manual for the first time, we encourage you to read the chapters in sequence. The chapters are organized in such a way to introduce you to DecoderScript writing step- by- step.

Screenshots of the FTS protocol analyzer have been included in the manual to illustrate what you see on your own screen as you develop decoders. But you should be aware for various reasons, the examples may be slightly different from the ones that you create. The differences could be the result of configuration differences or because you are running a newer version of the program. Do not worry if an icon seems to be missing, a font is different, or even if the entire color scheme appears to have changed. The examples are still valid.

A large number of sample files have also been included with examples of decoders, methods and frame recognizers. These files are installed in the FTE directory of the system Common Files directory. The readme file in the root directory of the protocol analyzer installation contains a complete list of included files. Most files are located in My Decoders and My Methods.

We will be updating our web site with new and updated utilities, etc, on a regular basis and we urge decoder writers to check there occasionally.

Chapter 2: How the Protocol Analyzer Processes Captured Data

In this chapter, we describe how captured data gets from where the protocol analyzer collects it to the point where you see each protocol layer decoded on your screen. If you understand how a protocol analyzer works, you can skip this chapter and move to Chapter 3. If you don't understand how data is captured and analyzed or if you want a quick refresher, it would be a good idea to read through this chapter.

Protocol Layers

Now let us look at what lies within a frame.

Generally, network communication is performed using protocol stacks. The most popular stack is the IP or Internet stack. Others include the Netware Stack and IBM's SNA. When data is transmitted, it is passed down through a stack of agents. Each agent is responsible for managing certain communication functions such as making sure the data is sent to the right place, ensuring it arrives intact, ensuring that shipments are delivered in the right order, and so on. In order to accomplish its designated tasks, each agent prepends its own protocol header to the data and then passes it down to the agent at the next layer. Some agents add a trailer as well, typically to hold a CRC or other frame check. Generally, the higher-layer agents are implemented in software and the lower-layer ones in hardware. On receipt, the opposite happens: each layer's agent strips off its own protocol header (and trailer if there is one) and then passes what remains to the appropriate agent at the next layer up.

So what we find inside a frame is a sequence of protocol layers. We have a very specific use for the term "layer": it refers to such a protocol layer within a frame. From this point on in this manual we will use "layer" strictly and solely in this sense. Or, if we do use it in another sense, then we will make that very clear in the context.

If you are familiar with the term layer as used in the ISO seven-layer model then we must emphasize that our meaning, while certainly related, is significantly different. Most important is that we use the word without any implication as to functionality. We do not assume, for example, that if the first layer is a data-link layer then the second layer must be a network layer. If people want to use a data-link protocol on top of a presentation-level protocol, or a zeeblonking protocol over a freezewhacking one, it is all the same. In other words, the protocol analyzer does not care what your protocols do or how they are arranged – it just decodes them. Indeed, thanks to practices like encapsulation and tunneling, it is quite legitimate to see a frame that contains protocols layered like the one on the next page.

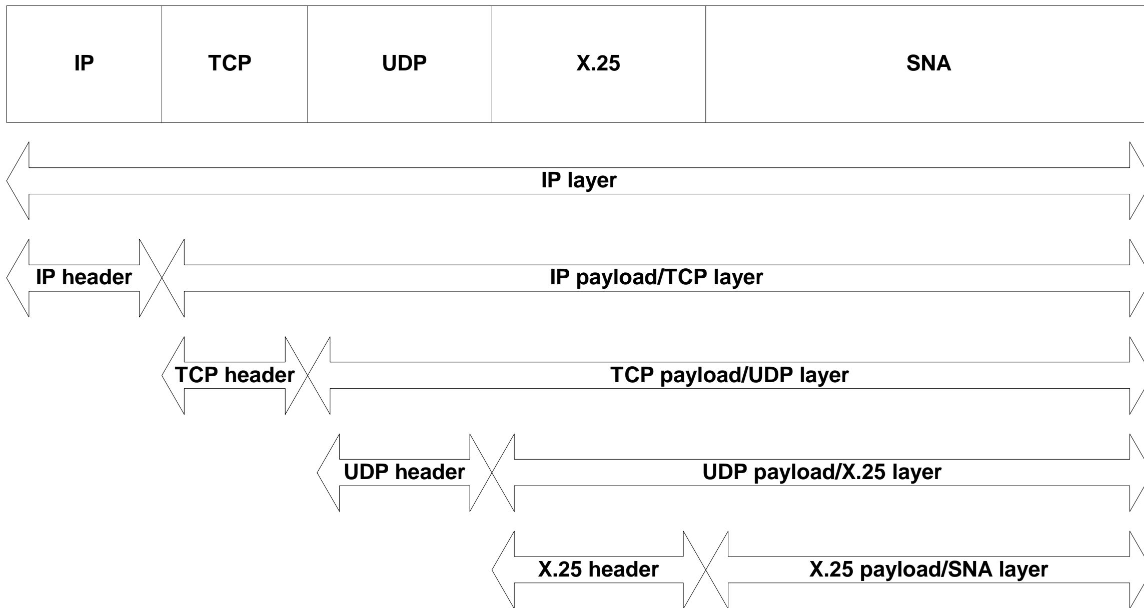


Figure 1 - Protocol Layers

How protocols are layered is not important to the protocol analyzer. The analyzer needs to know the beginning and ending of each layer and the protocol used at each layer. Knowing where each layer begins and ends is easy because, in most cases, one layer begins immediately after the header of the previous layer and continues to the end of the frame. We have ways of handling exceptions such as when there is padding in front of a layer.
Determing the protocol used at each layer is generally easy as well. For a start, the first layer is usually fixed. As for the rest, in some cases there is a fixed sequence. In most other cases the header of one layer includes information that identifies the protocol at the next layer.

The diagram above also serves to illustrate what we mean by “layer”, “header” and “payload”. A protocol header contains control information and is what the decoder for the protocol primarily decodes. The data carried by the protocol is called its payload. Generally, the payload of one layer comprises the next layer. Matters get a little more complicated when trailers are involved; then the payload of a protocol starts after the header and continues only up to the trailer, not to the end of frame.

There are communication links on which only one protocol is used. This would be the case when a protocol such as ZMODEM or Kermit is used in isolation. Then, we would say, the stack is one-layer deep and that a frame and a layer are essentially the same (unless there happens to be padding at the end of the frame).

Autotraversal

Autotraversal is the ability of the protocol analyzer to automatically manage the transition from one protocol layer to another. It depends on a number of factors including predefined stacks and elements in decoders that identify which protocol comes next as the sequence of layers is decoded.

We have said a great deal about layers because it is important to gain a perspective on how the protocol analyzer operates. In practice though, when writing a decoder you are working within the realm of a single protocol and only need to be concerned about other layers in a couple of small and well defined contexts.

How data is divided up into frames depends on the type of communication source involved. With Ethernet, the job is done by the hardware; the Ethernet adapter sends the data to the protocol analyzer a whole frame at a time. (That, by the way, explains why every byte in an Ethernet frame ends up with the same timestamp). With bit-synchronous protocols like HDLC, the hardware also does the work, but in a different way: The receiving hardware passes data to the protocol analyzer a byte at a time with other signals interspersed to denote where frames begin and end. (In some cases, both with Ethernet and bit-synchronous links, the hardware also verifies the CRC on the frame). Finally, there are cases where the hardware is oblivious to framing and the job must be done by software. Examples are byte-synchronous protocols such as Bisync and asynchronous protocols such as PPP and SLIP.

If this business of splitting captured data into frames remains unclear to you, then picture the data bytes as shipping containers. Then, in place of frames, think of shipments – a shipment is a batch of containers that are associated in some way and sent together. Just as with data, containers can be shipped in a variety of ways: by ship, truck/lorry, and railway at least. Then the Ethernet case is similar to containers being sent by rail. From an external viewpoint at least, each frame is like a train. It is very clear where one train ends and the next begins – even if trains run together, you have a locomotive on the front and a caboose/guard's van on the end of each. The other cases are analogous to containers being shipped by road – let us make it a one-lane road because the bytes/containers must arrive in the same order that they were sent in. Then how do we tell where one shipment ends and another starts? In the bit-synchronous case, we insert some special containers between shipments to serve as markers, or flags. These could literally have flags attached to them or they could be painted a unique color. In any case the shipper will have no trouble sorting out the shipments. In the asynchronous case, there are no external clues – one has to look inside the containers to divide up the shipments. Therefore, the job typically falls to the customer (the software) rather than to the shipper (the hardware).

There are many reasons why this is far from a perfect analogy. We just hope it will help you comprehend different aspects of protocol decoding. In some cases, special software usually protocol-specific software, is required to take care of what we call "frame recognition". When the task of frame recognition falls to software, we call that software a Frame Recognizer.

The Processing Sequence

We outline the major processing sequence the protocol analyzer uses to process data. An appreciation of these steps is vital to understanding certain aspects of protocol decoding. The diagram on the next page will illustrate this sequence.

At the top, captured data is fed through a Frame Recognizer if required. Once that is done we can view the capture as a sequence of frames awaiting decoding. The rest of the diagram traces the processing of a single frame.

The first step in processing a frame is to feed it through the Frame Transformer to prepare it for decoding, if such is required.

The second step, which like the first is optional, is to pass the frame to a Preprocessor Method. We have not mentioned these before because preprocessors are seldom used. Preprocessors can do two things. They can set up inter-frame data and initialize fields.

The next step is to run the decoder and display the results in the the protocol analyzer Frame Display window.

After decoding, a Postprocessor method may be run. Like preprocessors, postprocessors are rarely used. One example is provided by the decoder for Van Jacobson compressed headers (VJC.DEC) which uses a postprocessor to store inter-frame data.

The next step in the flow may involve the invocation of one or more methods that help set things up for working with the next protocol layer. Specifically they may determine what protocol is used, where the layer starts and how long it is. When the next protocol can vary (i.e. when it is not fixed in the stack definition that the user has specified) then a NextProtocol Method must be provided. This dynamically figures out what protocol comes at the next layer. If the next protocol is fixed in the stack definition then any NextProtocol Method specified in the decoder is ignored. By default, the next layer is assumed to start immediately after the current one and either extend to the end of the frame or define its own length. If there is padding or a gap before the next layer then a NextProtocolOffset Method must be supplied; its job is to determine how many bits must be skipped. Finally, when the next layer does not extend to the end of the data, a NextProtocolSize Method is required to calculate the length of the next layer. Note that if a NextProtocolOffset or a NextProtocolSize method is defined then it is always invoked even if the NextProtocol Method is not.

The next step is the optional processing of the frame by another Frame Transformer method. It rarely happens that frame modification is needed at this stage but it can be appropriate if the current decoder needs to make any adjustments before the next decoder gets hold of it. You can see examples in the decoders for Van Jacobson headers (VJC.DEC and VJU.DEC). VJU replaces the byte that specifies that the frame should be saved for VJC with a value indicating that the next protocol is TCP. This allows the VJU frame to look just like an IP frame so the standard IP decoder can be invoked. VJC takes the saved IP header from the VJU frame decoded earlier, modifies it, and then creates an entirely new frame with that and the rest of the payload from the current frame, and passes that back as the new frame to use.

The final job is to check if the end of the frame has been reached. If it has then the decoding of the frame is complete. Otherwise we loop back and start processing the next layer.

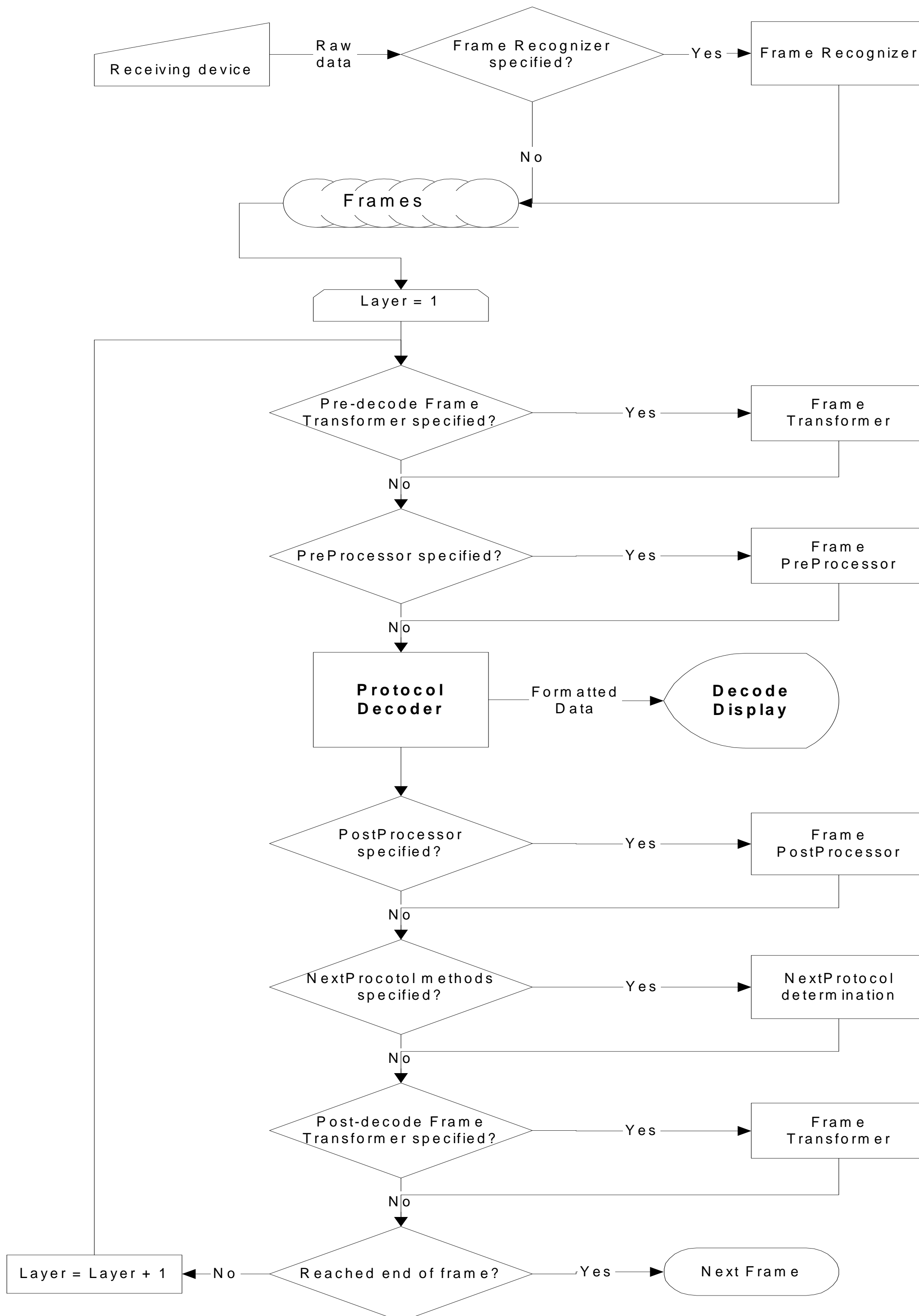


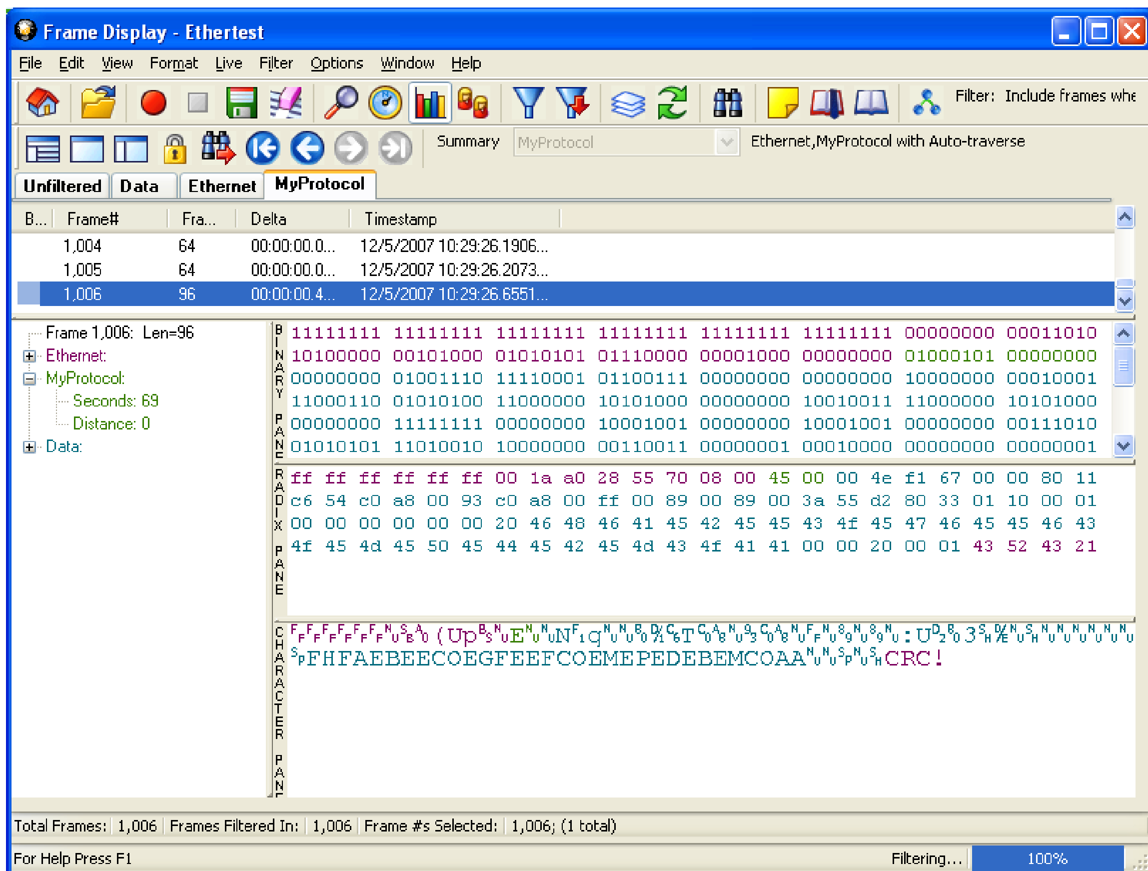
Figure 2 - Frame Processing Flow Chart

Chapter 3: Structure of a Decoder

Our aim is to acquaint you with the most essential elements of DecoderScript and leave you with sufficient expertise so that you can write decoders of your own.

We will begin with a short basic overview of DecoderScript:

FTS turns this…
Stream of bytes
…into this:



To do this, FTS uses a set of instructions called a *decoder.* A decoder is written in a language called DecoderScript.

Here's the decoder, written in DecoderScript, that creates the Frame Display screen shown above:

```
// Copyright 2007 Frontline Test Equipment, Inc. All rights reserved.
"MyProtocol" 0x7f00fff7
DECODE
FIELD secs (Fixed 1Byte) (Decimal) “Seconds”
FIELD distance (Fixed 1Byte) (Decimal) “Distance”
END_MAIN_PATH
```

This is a very simple example which illustrates some very fundamental capabilities of DecoderScript.

The FIELD statements define the contents of the Frame Display. The first statement breaks down like this:

Item	Description
Field	Keyword that identifies the beginning of a field statement.
secs	Identifier of field definition statement.
(Fixed 1)	Fixed is a size method and defines the size of the field.
(Decimal)	Formats the output to be displayed in decimal format.
"Seconds"	Display the value in the detail pane (lower-left corner of Frame Display) with the label "Seconds".

Description of the first FIELD statement in the example

The decoder starts with a name ("MyProtocol" in this name that appears on the tab in Frame Display) and a value called a decoder ID. Each decoder must have a unique decoder ID.
To see the decoder above in action, do the following:

1. Type the entire decoder into a file called mp.dec.
2. Put file mp.dec into directory C:\Program Files\Common Files\FTE\My Decoders.
3. Run FTS and do either a live capture or open a capture file.
4. Open the Frame Display and do the following:
   - Select the Options menu
   - Select Protocol Stack…
   - Select __Build Your Own__
   - Press the Next button twice
   - Remove all entries from the Protocol Decode Stack list by selecting them and pressing the left arrow button.
   - Add the contents of mp.dec by selecting MyProtocol in the box on the left and pressing the left arrow button.
   - Press the Finish button.

In the Frame Display you will now see the packets decoded by your mp.dec decoder. It will appear on MyProtocol tab with the data decoded

To futher build on our short version of decoderscript writing, we will now begin to explain in a little more detail-

What You Need

In order to write a decoder, you need these things:

1. You need the protocol specification. The specification helps you identify how the protocol should work.

2. You need data. Data can be captured live using your FTS protocol analyzer, or it can be created using CFA Maker, a utility for creating capture files. The raw material that the protocol analyzer works with is the captured data. We refer to the contents of a capture file as events. These events comprise not only the actual data bytes, but also start- and end-of-frame markers, and indications of changes in control signals (e.g. DTR, RTS, etc., in the RS-232 case). An event may contain several pieces of information including a timestamp and a record of errors (parity, framing, etc.).

3. You need to be able to separate your data into frames. By “frame”, we mean the fundamental protocol package that is sent and received. There are three ways to frame data:

   Alternative 1

   Hardware can frame the data for you. In most cases where the hardware is able to frame the data, your protocol is likely not the lowest layer protocol in the stack, but is somewhere higher with other protocols beneath it. Examples of protocols with this scenario would be Ethernet and some kinds of synchronous serial data

   Alternative 2

   A soft frame recognizer can also frame data. The recognizer inserts flags at the beginning and end of frames. Asynchronous and some synchronous serial data are examples of data that will need to use a frame recognizer. A frame recognizer is provided by the protocol analyzer

   Alternative 3

   If you need to write a custom recognizer, then you need Microsoft's Visual C++.NET installed on your PC and a text editor in which to write your method. We strongly recommend that you use version 7.1 of MS Visual C++.NET. This product uses the Visual C++ compiler to compile your recognizer – you do not need to compile it yourself.

4. Decode your data. You will need a text editor. We use a shareware tool called UltraEdit because it can do syntax highlighting and this makes DecoderScript much easier to read, but any program that lets you create text files will work. UltraEdit is available from www.ultraedit.com, and we have included a wordfile.txt with this product that will provide syntax highlighting for DecoderScript. If you use a different text editor that supports syntax highlighting, the list of keywords and method names can be found in wordfile.txt in the section labeled */L6"Protocol Decodes"*.

Basic Structure

A decoder module usually has four sections:

Header
Reference
Table
Executable

The Header Section

```
/*  Internet Protocol version 4. Copyright 2000-2001 Frontline Test Equipment, Inc. All
rights reserved. */
"IPv4" 0x7f000401
```

A decoder module normally starts with a comment that:

Names the author (Frontline Test Equipment, Inc.)
References the protocol specification document used (Internet Protocol version 4)
And usually includes a copyright notice (Copyright 2000-2001 All right reserved)

These three comments are optional as indeed are all comments. However, we highly recommend using comments as they may later serve as a reference source for you or other engineers who may use your decoder.

The syntax of a **comment** matches that used by the C++ programming language. There are two types of comments. One opens with a slash and an asterisk and closes with an asterisk and a slash; in each case, the pair of characters must be written together with no intervening space. Such a comment may extend over several lines. The other type of comment starts with two slashes and ends at the end of the line. Comments may appear anywhere

The first required statement in the header identifies the protocol by name and id number. The protocol name, "IPv4" is the general name of the protocol typically expressed here in an abbreviated form. This name appears in the Protocol Stack Wizard and in the Decode Pane to indicate the protocol layer, so it is appropriate to choose a form that is suitable for these contexts.

The protocol name is followed by the decoder ID, 0x7f000401. Every protocol must have a unique id. Such IDs are expressed as hexadecimal numbers by convention, but can also be expressed in decimal. When expressed in hex, a "0x" is required before the number and must contain a minimum of 4 hex digits

Within a statement, items are always separated by spaces, never by commas or any other character. The protocol-identification statement has an optional third item which specifies the data order. Multi-byte fields within a protocol are interpreted in either network data order or host data order. The default data order for the decoder may be selected by including either "HOST_DATA_ORDER " or "NETWORK_DATA_ORDER ". Whichever is selected may be overridden as necessary on a per-field basis. If you omit the third item in the protocol-id statement then the default is network data order.

The Reference Section

The Reference section contains references to methods that perform operations of a global nature. They may be listed in any order. Here, as an example, is the reference section for the IP decoder:

```
NEXT_PROTOCOL (FromField protocol)
NEXT_PROTOCOL_SIZE (ThisLayerLength tot_length)
FRAME_AFTER_DECODING (IpReconstructFragments a)
```

Each statement in this section starts with one of the following keywords followed by a method invocation:

BYTE_STREAM_FRAMER
FRAME_AFTER_DECODING
FRAME_BEFORE_DECODING
NEXT_PROTOCOL
NEXT_PROTOCOL_OFFSET
NEXT_PROTOCOL_SIZE
PAYLOAD_IS_BYTE_STREAM (no method)
PREPROCESSING
POSTPROCESSING
RECOGNIZER
SUMMARY_COLUMNS (no method but special syntax to describe columns)

A decoder can only use one of each of these statements with the exception of PREPROCESSING and POSTPROCESSING. An unlimited number of PRE- and POSTPROCESSING directives may be included and in each case, they will be performed in the order in which they appear.

DECODE

The keyword, Decode, must be placed between your reference section and whatever follows. Generally this is the Table Section.

The Table Section

As you would expect, the table section of the decoder contains a list of tables. Table values are always included in the protocol specification. Here is an example of a table taken from the IP decoder:

```
TABLE prec
{ 7 "Network Control"}
{ 6 "Internetwork Control"}
{ 5 "CRITIC / ECP"}
{ 4 "Flash Override"}
{ 3 "Flash"}
{ 2 "Immediate"}
{ 1 "Priority"}
{ 0 "Routine"}
{ Default "Unknown"}
ENDTABLE
```

This table lists the order of precedence of a packet. The table is bracketed by TABLE and ENDTABLE keywords with the former bearing the name of the table. Each entry in the table is written as a list of items in braces. The first item in an entry is the field value and the second is a string for display in the Summary Pane. Tables are usually ordered by field value, though this is not a requirement. We recommend that you follow this practice.

The last entry in this table has the value Default. This serves as a catchall for values that have no entries. We urge you to include a default entry for every table except perhaps for small tables that have an entry for every possible field value. One last reminder about tables: comments may not be interspersed with table entries.

The Executable Section

The executable section is where the real decoding work is done. As its statements are processed, an implicit pointer is moved through the layer being decoded from beginning to end.

```
FIELD command_code (Fixed 1 Byte) (Table tableCommands)
        IN_SUMMARY "Command/Response" 150 "Command"

FIELD actuator_number (Fixed 1 Byte) (Decimal) "Actuator Number"
    VERIFY (FieldIsBetween 1 32)

FIELD actuator_value (Fixed 2 Bytes) (Decimal)
        "Value to set"

FIELD checksum (Fixed 1 Byte) (Hex)
        IN_SUMMARY "C'sum" 40 "Checksum" VERIFY (FPChecksum)

END_MAIN_PATH
```

Normally each FIELD statement decodes a given number of bits or bytes and then moves the pointer forward by that amount ready for the next FIELD. As we shall see, however, we can manipulate the pointer in various ways in order to process fields out of sequence or to process them twice when the need arises – as it often does. Strictly speaking this pointer is a bit pointer. If you are dealing with a protocol that is totally octet-oriented, you may safely imagine the pointer as a byte pointer.

Eight types of statements appear in the executable section. Five of them, FIELD, RESERVED, GROUP, GROUP FIELD and BRANCH, we refer to collectively as executable statements. The sixth, PARAMETER, is a pseudo-field, the seventh is the INCLUDE keyword and the eighth is END_MAIN_PATH. When the protocol analyzer sees END_MAIN_PATH, it stops decoding that layer and goes on to the next.
Every statement except BRANCH, RESERVED and INCLUDE defines some kind of protocol field or group of fields and every such entity must be given a unique name within a decoder. Thus, not only must all FIELDs have unique names, but you cannot have a FIELD with the same name as a GROUP or a PARAMETER. By convention we use names that are all lower case and with words separated by underscores.

We encourage you to follow along with the tutorial examples. FP.dec, the Fictitious Protocol decoder file, is in the My Decoders directory of your product installation. We also provide a capture file of FP data called FP.cfa. You can load FP.cfa into your protocol analyzer and see how the different types of data appear on the screen. FP.cfd is the capture file declaration used to create FP.cfa. This is a text file that can loaded into CFA Maker to create a capture file. FP.cfd shows the data we used to create FP.cfa and explains what type of data is in each frame. For information on how to obtain a copy of CFA Maker, please contact Technical Support.

Throughout this tutorial, hexadecimal numbers will be prefixed with "0x" to distinguish them from decimal values.

Fictitious Protocol is the next example we will use to further explain DecoderScript writing. First let’s look at the type of data we want to decode. Then we will build our decoder.

Fictitious Protocol

FP is an asynchronous point-to-point protocol that connects a Master (some kind of computer) with a Slave (some kind of equipment, let us say an air-conditioning or refrigeration unit). The Slave has 32 sensors (such as a thermometer) and 32 actuators (such as a thermostat). The protocol comprises commands that the Master sends to the Slave either to set a value in an actuator or to read a value from a sensor. The Slave responds with acknowledgements and sensor readings respectively. The Slave never sends anything except in response to a message from the Master.

The content of each frame is either a command or a response. A command looks like this:

Command Code	Data	Checksum

The Command Code is a single-byte field that identifies the command. It is followed by zero or more data bytes. The number of such data bytes is fixed for each command. The last field is a one-byte checksum that is the exclusive of all the other bytes.

A response looks a little different:

Response Code	Data Count	Data	Checksum

The Response Code field is a one-byte element that identifies the response. It is followed by a one-byte count of the data bytes that follow.
Here is a list of commands with the data associated with each:

Code	Operation	Data
0	Request Status	**(none)**
1	Set Actuator	**One-byte actuator number** **Two-byte value to set**
2	**Read Sensor**	**One-byte sensor number**

And here is a similar list of responses:

Code	Operation	Data
128	Status	**One-byte set of bit flags described below**
129	Actuator Set	**One-byte actuator number**
130	**Sensor Value**	**One-byte sensor number** **Two-byte sensor value**

The bit flags returned with the Status response include (in binary):

Bit	Meaning
00000100	**Alarm: one or more sensors are in the alarm range**
00000010	**Power: unit has noted a power disruption**
00000001	**Test: unit in diagnostic mode**

(other bits reserved)

Let us look at the simple exchange of one command and one response. The master sends a command to set actuator number 10 and the slave acknowledges this. Here is a snippet from the Frame Display:

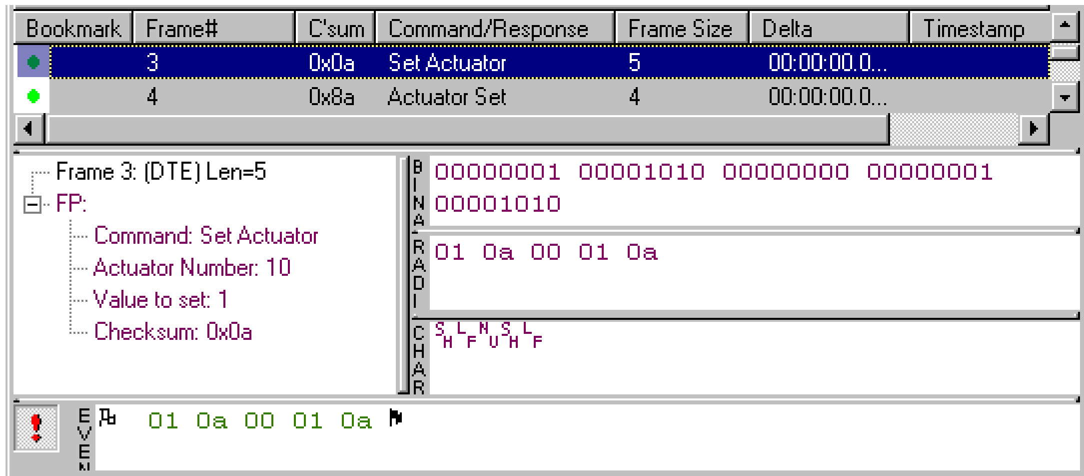


Figure 3 - Set Actuator Command and Response

We assume that you are familiar with the basic layout and operation of the Frame Display window. If you are not, we suggest you familiarize yourself with it using one of the FTS protocol analyzer demos.

In the Summary Pane (the Frame Display subwindow with column headings starting with “Bookmark”), we see a one-line entry for each frame. This pane gives you a view over the succession of frames in the capture with the most important details of each. The columns headed “Bookmark”, “Frame#”, “Frame Size”, “Delta”, and “Timestamp” are standard; you will see them in every Frame Display window no matter what protocol is being decoded. The columns labeled “C’sum” and “Command/Response” are inserted by our FP decoder.

You can make your decoder add any columns you choose to the Summary Pane and display whatever you deem useful in those columns. Our “Command/Response” column shows the name of the command or response in the frame. This information, the type of frame (whatever that means in the context of a particular protocol), is certainly the most crucial thing to put in the summary. The other custom column, “C’sum”, shows the checksum for the frame. There is enough column space left that we could add even more columns for FP; for example we could have an “Actuator/Sensor” column and display the relevant number for frames that contain such. Deciding what to show in the Summary Pane and how to organize that data into columns is an exercise for the decoder writer — there are no rules about it. By the way, the values in the “Delta” and “Timestamp” columns shown above are not meaningful because we artificially constructed the captures (since FP is not a real protocol) and no timestamps were included.

The frame that is selected (highlighted) in the Summary Pane is displayed in detail in the other panes. The key one is the Decode Pane (at left center). This is the one pane that a decoder has almost complete control over. The logical data panes on the right (Binary, Radix, and Character) show the data in various formats with the data representing the field selected in the Decode Pane highlighted. At the bottom, the Event Pane shows the raw data that comprises the frame including start-of-frame and end-of-frame events (those flag-like symbols) that we will explain later.

As we see in Figure 1, the Decode Pane starts with a one-line description of the frame ("Frame 3: (DTE) Len=5" in our case). Below this is a tree with one branch at the root level for each protocol represented within the frame. In our example, we have just the one protocol, FP. Everything on the FP branch is generated by the FP decoder. In our example every FP field is shown at the same level but a decoder can construct sub-trees of arbitrary complexity.
Now that we know what type of information we are dealing with, let's create out decoder script.

The Basic Structure of a Decoder File

Let's start building our decoder. We will start out with the bare bones and build from there. Some of the terms you have already seen in the preceding pages, while others are going to be new.
We will start our decoder by writing a decoder for FP's Set Actuator command. The structure for this command is

```
"Fictitious Protocol" 0x7f000000

DECODE

FIELD command_code (Fixed 1 Byte) (Hex) "Command"

FIELD actuator_number (Fixed 1 Byte) (Decimal) "Actuator Number"

FIELD actuator_value (Fixed 2 Bytes) (Decimal)
        "Value to set"

FIELD checksum (Fixed 1 Byte) (Hex) "Checksum"

END_MAIN_PATH
```

The first element in the file is the string "Ficititious Protocol". This is the name of the decoder as you wish it to appear in the protocol analyzer. As you remember, this is the Heading section. The name appears in the Protocol Stack Wizard and also in the Summary and Decode panes on the Frame Display.

The second element is the Decoder ID. Every decoder must have a unique Decoder ID. The Decoder ID can be expressed in either decimal or hexadecimal, but as they are rather large numbers, we use hexadecimal by convention. You recognize this as a part of the Reference section.

The next element is a statement reading simply "DECODE", which must be placed just prior to the first executable statement.

One thing to note here as we continue is writing a decoders is not a linear process. Even though a script is made up of sections that should appear in a certain order, you will commonly work in different parts of the script and different times.

Let's start with the Field Statement. Field Statements are found in the Executable section.

The FIELD Statement

The heart of a decoder is the set of FIELD, GROUP and associated statements that take values from given protocol fields and displays them in the decode window. Sometimes you want to directly display the value in a field in the form of a number; other times you want to display a string indicating what that value represents. Almost every protocol has fields that can be displayed most informatively by selecting a string from a table indexed by the field value.

Note:

Syntax rules and similar guidelines are offset from the main text of the tutorial in boxes like this one. Generally, such rules cover only simple cases of use.
Let's deconstruct these Field Statements

General Syntax Rules

A decoder is constructed of three basic elements: statements, comments and braces.

The syntax of a **comment** matches that used by the C++ programming language. There are two types of comments. One opens with a slash and an asterisk and closes with an asterisk and a slash; in each case, the pair of characters must be written together with no intervening space. Such a comment may extend over several lines. The other type of comment starts with two slashes and ends at the end of the line. Comments may appear anywhere.

A **statement** consists of a sequence of items separated by white space (spaces or line breaks). It would probably be more common usage to refer to a statement as a sequence of fields but we retain the term "field" to apply strictly to a protocol field and use "item" for the components of a statement. An item may be a keyword, a name/string, a numeric constant, or a method invocation.

<u>Keywords</u> that we have already met include FIELD and GROUP. We have a convention that keywords are always written in all capitals; this makes for good readability but is not required.

A <u>name/string</u> is a character string that represents, for example, the name of a field or the text to appear in a decode detail. A name/string must be enclosed in double quotes if it contains spaces or characters other than letters, numbers, and underscores. While names and strings are treated the same syntactically, conceptually they are rather different and by convention we write them in distinct ways. FIELDs and GROUPs are given names that do not include spaces and their names are not quoted. Furthermore, names are usually all lowercase and underscores are used as in this_is_a_field_name. We make a practice of giving methods names exemplified by DoThisToThat. Whenever names are matched, the comparison is done without sensitivity to case. Strings, that is text fragments to be displayed as part of the decode, are always double-quote delimited for readability.

A number is expressed as either a signed decimal (e.g. –10) or in hexadecimal notation prefaced by “0x” (for example, 0xFFFF).

A method invocation is written as a parenthesized expression consisting of a method name followed by a list of parameters as in: (MethodName param1 param2)

Braces are used to group statements together and to delimit table entries. Braces must appear in matched pairs.

General Syntax:

DecoderScript statements are written as a sequence of items separated by at least one space. A line break counts as a space so a statement may be spread over two or more lines (as shown in the actuator_value FIELD). Numbers may be expressed either in decimal or in hexadecimal prefaced by “0x”.

Following each name, we see an expression that defines the size of the field, (Fixed 1 Byte) for example. This size may serve as many as three purposes. First, as each protocol layer is decoded, a pointer (called the "bit pointer") passes through the data. This pointer starts out at the beginning of the layer and, as each field is processed, it is incremented by the size of the field. The second purpose is to store the field value in an appropriate way. And third, the size may affect how the field value is formatted.

Standard Methods

The field size is derived by calling a method. A method is akin to a function, routine or subprogram in a programming language; it is a piece of code that accepts certain inputs, does some calculation with them, and returns an output. There are several types of methods that serve different needs. The type used here is called a Size Method; others include Format Methods and Verify Methods. A standard method is one supplied with the protocol analyzer.

Method Syntax:

A method is always written within parentheses. The name of the method is given followed by a list of zero or more parameters. Methods can have optional parameters. Optional parameters have use a default value when a specific value is not given.

Size Method:

In our examples above, the size of each field is fixed and is obtained by invoking the Size Method called Fixed that returns a given constant. The Fixed method takes two parameters, a count and a unit name. The unit may be “bits”, “bytes”, “words”, etc. In other cases, the size of a field may be specified by the contents of another field, may be defined by some fixed value that marks its end (such as a null-terminated string), or may comprise the rest of the data in the frame. Various other Size Methods are provided to handle these situations.

The Format Method:

The next component of our FIELD statement is the Format method invocation. You will see (Hex) and (Decimal). All these are Format Methods and serve to format the field value for display. As you would guess, they format hexadecimal integers and unsigned decimal integers respectively. (Formatting signed decimals requires a Retrieve method to extract the sign bit; this is an advanced topic discussed later.)
The last item is the decode label (e.g., "Command") that appears at the end of the FIELD statement. The decode label is separated from the formatted value by a colon and space. Here is a screenshot of the Decode-Pane display of a Set Actuator command:

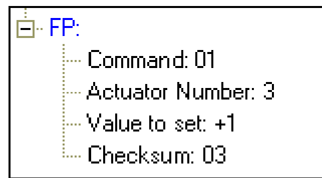


Figure 4 - Set Actuator Command

VERIFY Field Values

In the description of FP, we state that there are 32 actuators. The actuator numbers are 1 through 32, which means the actuator_number field should not contain any value less than 1 or greater than 32. We can check this by adding a VERIFY statement to the end of the actuator_number field and using a Verify Method. The resulting new statement is:

```
FIELD actuator_number (Fixed 1 Byte) (Decimal) "Actuator Number"
    VERIFY (FieldIsBetween 1 32)
```

FieldIsBetween takes two parameters, a low value and a high value. The method is inclusive, which means if the field value is 1 or 32 or any value in between it will pass the test. If the field value is less than 1 or greater than 32 the test will fail, and the field will be tagged in the Decode pane in red and the expected value displayed, as shown in this snippet from the Frame Display.

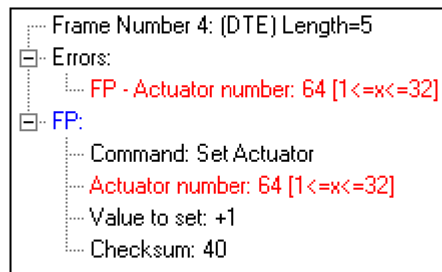


Figure 5 - How Errors Are Displayed

In addition, the frame number on the Summary pane will also be in red.

Verify Methods can be used to validate a message checksum as in:

```
FIELD checksum (Fixed 1 Byte) (Hex) "Checksum"
          VERIFY (FPChecksum)
```

Our decode of the Set Actuator command now looks like this (new items are bolded):

```
"Fictitious Protocol" 0x7f000000

DECODE

FIELD command_code (Fixed 1 Byte) (Hex) "Command"

FIELD actuator_number (Fixed 1 Byte) (Decimal) "Actuator Number"
    VERIFY (FieldIsBetween 1 32)

FIELD actuator_value (Fixed 2 Bytes) (Decimal)
          "Value to set"

FIELD checksum (Fixed 1 Byte) (Hex) "Checksum"
          VERIFY (FPChecksum)

END_MAIN_PATH
```

Custom Methods

Before we continue with the Verify Method, this is a good time to introduce Custom Methods. All the methods we have encountered previously have been what we call “standard methods”. A standard method is one supplied with the protocol analyzer. The method used here to verify the checksum, FPChecksum, as its name suggests, is a method specific to Fictitious Protocol. We call this a “custom method” since it is created by the decoder writer. The protocol analyzer supplies standard methods, including standard checksum methods, to do just about everything that is commonly needed in decoders and most decoders can be written with those alone. However, should the need arise for a method unique to your protocol’s particular requirements, then that is easily met by writing a custom method.

Now let’s get back to Verify Methods. Other very useful Verify Methods compares the field value with a constant as in these examples:

```
(FieldIs LessThan 10)
(FieldIs LessThanOrEqualTo 17)
(FieldIs EqualTo 0)
(FieldIs NotEqualTo –1)
(FieldIs GreaterThan 4)
(FieldIs GreaterThanOrEqualTo 0)
```

VERIFY Syntax:

When a VERIFY clause is used, it must be placed at the end of the FIELD statement. It consists of the keyword VERIFY followed by a call to a Verify Method such as FieldIs or FieldIsBetween. There are also several Verify Methods for validating checksums.

Before we look at adding a table to our decoder script, let's look at how to create Summary Pane columns using the IN_SUMMARY field statement

Creating Summary Pane Columns using IN_Summary

Normally every field is displayed in the Decode Pane. The special IN_SUMMARY clause prompts the protocol analyzer to display it in the Summary Pane as well. The keyword IN_SUMMARY is followed by the name of the column in which the value should be shown. Referencing a column in this way suffices to create it. The name is followed by a number that specifies the default width of the column in pixels.
We will enhance our FP decoder by putting the checksum and command_code fields in the Summary pane. The IN_SUMMARY keyword appears right before the label. The resulting field statements are:

```
FIELD command_code (Fixed 1 Byte) (Hex)
        IN_SUMMARY "Command/Response" 150 "Command"

FIELD checksum (Fixed 1 Byte) (Hex)
        IN_SUMMARY "C'sum" 40 "Checksum" VERIFY (FPChecksum)
```

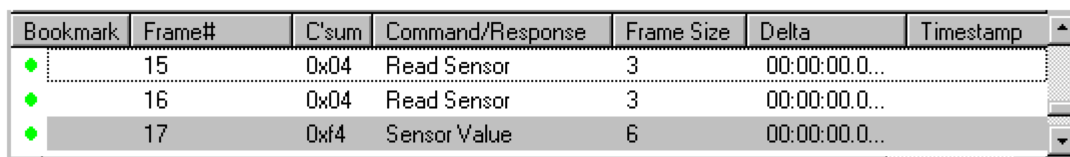
Our decode of the Set Actuator command now looks like this (changes are in bold):

```
"Fictitious Protocol" 0x7f000000

DECODE

FIELD command_code (Fixed 1 Byte) (Hex)
        IN_SUMMARY "Command/Response" 150 "Command"

FIELD actuator_number (Fixed 1 Byte) (Decimal) "Actuator Number"
   VERIFY (FieldIsBetween 1 32)

FIELD actuator_value (Fixed 2 Bytes) (Decimal)
        "Value to set"

FIELD checksum (Fixed 1 Byte) (Hex)
        IN_SUMMARY "C'sum" 40 "Checksum" VERIFY (FPChecksum)

END_MAIN_PATH
```

You can put the value of more than one field into the same Summary pane column. For example, the name of the command_code column is "Command/Response", which implies that somewhere there is a response_code field, which also puts its value in the Command/Response column. All you need to do is reference the same column name in the second field. For example:

```
FIELD response_code (Fixed 1 Byte) (Hex)
        IN_SUMMARY "Command/Response" 150 "Response"
```

The advantage of doing this is that you can easily see a command followed by its response in the Summary pane. In the case of our Fictitious Protocol, where a slave is not supposed to talk unless instructed to by the master, cases where a slave speaks out of turn or a master has to ask something twice jump out. Here's a fragment of the Summary pane illustrating this:



Bookmark	Frame#	C'sum	Command/Response	Frame Size	Delta	Timestamp
	15	0x04	Read Sensor	3	00:00:00.0...	
	16	0x04	Read Sensor	3	00:00:00.0...	
	17	0xf4	Sensor Value	6	00:00:00.0...	

Figure 6 - Illustration of Master Sending Twice

IN_SUMMARY Syntax:

IN_SUMMARY clauses appear right before the label. They consist of the keyword IN_SUMMARY followed by the column name the field value should appear in and the default column width in pixels.

Summary pane columns will appear in the order the fields are decoded. There is a keyword called SUMMARY_COLUMNS which will allow you to set the order of the columns.

Tables

The last enhancement needed to our decode of the Set Actuator command is to make the command_code field show the name of the command rather that just the hex value. That way the Summary and Decode panes will be easily readable as in the example above, where you can quickly scan down the Command/Response column and see what commands have been issued.

We use a Format Method called Table to accomplish this. At this point you may ponder and say, “I thought Table was a section within a decoder script. That is what it says on page 25.” This is true, table is indeed a section within the decoder script. However, Table is also a Format Method. We will explain.

Table uses a field value to index a table of strings. In effect, the Table Method acts as a pointer to a table within the Table Section. Tables are placed at the top of the decoder, right below the DECODE keyword and before any FIELD statements.

Here is an example that defines a table of FP commands:

```
TABLE tableCommands
{ 0 "Request Status" "Request Status"}
{ 1 "Set Actuator" "Set Actuator"}
{ 2 "Read Sensor" "Read Sensor"}
{ Default "???" "(unknown)"}
ENDTABLE
```

This is a DecoderScript representation of the list of FP commands given at the start of this chapter. Each entry starts with a number that corresponds to a command code except for the last in which the keyword “Default” tags a special entry that serves as a catchall for any values for which there are no explicit entries. The number can be in either decimal or hexadecimal. Two strings follow. The first is for display in the Summary Pane while the second is for the Decode Pane. In our table these two strings are the same for each entry other than the default. The value of having two distinct strings, one for each pane, is that you frequently want to display an abbreviated description in the Summary Pane (where space is tight) and a fuller description in the Decode Pane.

TABLE Syntax:

Tables begin with the keyword TABLE and end with the keyword ENDTABLE. The table name follows the TABLE keyword. Table names must be unique, which means that two tables cannot have the same name, and a table and a field cannot share the same name either. Table entries begin and end with braces. The elements of each entry are:

{ FieldValue "Summary Pane String" "Decode Pane String" }
If only one string is present, it will be used for both the Summary and Decode panes.

Once we have our table in place, we can alter our command_code field to use the table method to look up the appropriate string as follows:

```
FIELD command_code (Fixed 1 Byte) (Table tableCommmands)
        IN_SUMMARY "Command/Response" 150 "Command"
```

Below are two snapshots from the Decode pane. The picture on the left shows the command_code field formatted in hex while the picture on the right shows the field formatted from the table.

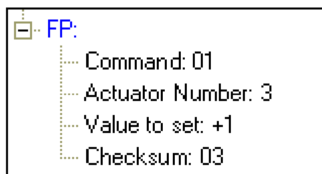


Figure 7 - Hex

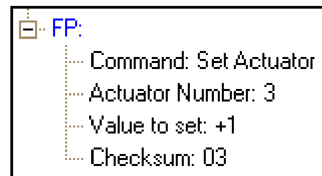


Figure 8 - From Table

Our decode of the Set Actuator command now looks like this:

```
"Fictitious Protocol" 0x7f000000

DECODE

TABLE tableCommands
{ 0 "Request Status" "Request Status"}
{ 1 "Set Actuator" "Set Actuator"}
{ 2 "Read Sensor" "Read Sensor"}
{ Default "???" "(unknown)"}
ENDTABLE

FIELD command_code (Fixed 1 Byte) (Table tableCommands)
        IN_SUMMARY "Command/Response" 150 "Command"

FIELD actuator_number (Fixed 1 Byte) (Decimal) "Actuator Number"
    VERIFY (FieldIsBetween 1 32)

FIELD actuator_value (Fixed 2 Bytes) (Decimal)
        "Value to set"

FIELD checksum (Fixed 1 Byte) (Hex)
        IN_SUMMARY "C'sum" 40 "Checksum" VERIFY (FPChecksum)

END_MAIN_PATH
```

Quite a change from our starting point!

The GROUP Statement

GROUP statements serve several useful purposes all related, as the name implies, to grouping together a sequence of statements such as the Set Actuator FIELDs above. The grouped statements are contained within braces and are normally indented for readability.

GROUP Syntax:

Statement starts with the keyword GROUP followed by a name. The name may optionally be followed by a string called a label. The GROUP is followed by a list of DecoderScript statements contained within braces.

Groups can be either labeled or unlabeled. For example:

```
GROUP labeled “labeled”

GROUP unlabeled
```

The difference lies in how they appear in the Decode Pane. For a labeled group, a sub-tree is displayed named with the given label. The user has to open the sub-tree to see the contained elements displayed. If the GROUP does not have a label then the GROUP's contents are displayed but no Node for the GROUP is displayed. There are other uses for GROUPs without labels that we will introduce later. Figure 7 shows how unlabeled vs. labeled groups appear in the Decode Pane. On the left is a straight decode of a Set Actuator Response, while on the right is a decode where the Response and Data Byte Count fields have been included in a GROUP labeled with the label "Response".

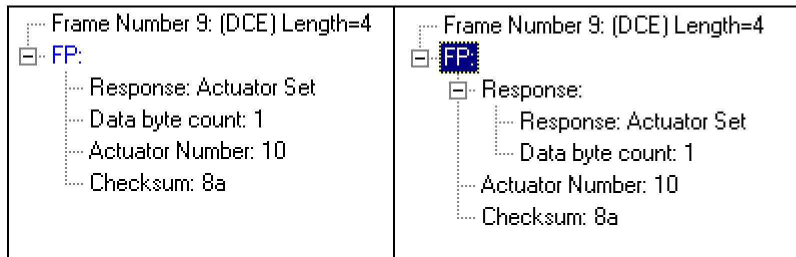


Figure 9 - Unlabeled vs. labeled GROUPs

If you recall from the description of FP, the Set Actuator command is just one of three possible commands. We've got a decode of the Set Actuator command, now we want to add a decode for the Read Sensor command. In order to separate the two, we'll put the Set Actuator FIELDS into a group.

```
GROUP cmdSetActuator "Set Actuator"
{
        FIELD command_code (Fixed 1 Byte) (Table tableCommands) IN_SUMMARY
        "Command/Response" 150 "Command"

        FIELD actuator_number (Fixed 1 Byte) (Decimal) "Actuator Number" VERIFY
        (FieldIsBetween 1 32)

        FIELD actuator_value (Fixed 2 Bytes) (Decimal) "Value to set"

        FIELD checksum (Fixed 1 Byte) (Hex) IN_SUMMARY "C'sum" 40 "Checksum"
        VERIFY (FPChecksum)
}
```

The format of the Read Sensor command is as follows:

Command Code	Sensor Number	Checksum

Our Read Sensor group will look like this:

```
GROUP cmdReadSensor "Read Sensor"
{
        FIELD command_code (Fixed 1 Byte) (Table tableCommands) IN_SUMMARY
        "Command/Response" 150 "Command"

        FIELD sensor_number (Fixed 1 Byte) (Decimal) "Sensor Number" VERIFY
        (FieldIsBetween 1 32)

        FIELD checksum (Fixed 1 Byte) (Hex) IN_SUMMARY "C'sum" 40 "Checksum"
        VERIFY (FPChecksum)
}
```

Notice that the command_code and checksum fields are identical in both groups. To avoid retyping the same information, we use the form:

```
FIELD field_name ;
```

Such a statement means: decode the field according to a FIELD definition given elsewhere named field_name. (There does not have to be a space between the field name and the semi-colon; we do that for readability.) This statement is particularly useful when you need to decode a certain field in multiple paths within your decoder. The full FIELD definition may be placed out of the main line of control and then referenced with the field name followed by a semi-colon. We can rewrite our Read Sensor group as follows:

```
GROUP cmdReadSensor "Read Sensor"
{
        FIELD command_code ;

        FIELD sensor_number (Fixed 1 Byte) (Decimal) "Sensor Number" VERIFY
        (FieldIsBetween 1 32)

        FIELD checksum ;
}
```

Incidentally, it is also possible to reference a GROUP multiple times using the same construction. This is useful if you have a series of statements that occur frequently. For example, a date sequence where the first byte is the day, the second the month and the third the year could be grouped into a GROUP date. Anywhere that same sequence is used, just enter GROUP date ; in the decoder and all three field statements in the group will be decoded.

We now run into a dilemma. The command_code field tells us whether the command is a Set Actuator or a Read Sensor command. How do we know which one it is? The answer is to use a conditional to test the value of command_code and then decide whether to decode sensor_number or actuator_number and actuator_value based on the value of command_code.

Conditionals

The first thing to do is take the Command Code field out of the two groups. We'll decode the Command Code first, and then use IF clauses on the Set Actuator and Read Sensor groups to decide between them. IF clauses may be used on individual fields or on groups. The syntax is the same in both cases.

IF Syntax:

An optional IF clause may be inserted in a FIELD statement following the Size Method. It consists of the keyword IF followed by a call to a Boolean Method. If the method returns FALSE then the FIELD is skipped. The effect of the test may be reversed by inserting the keyword NOT after the IF. Thus, the following are equivalent:

```
IF (FieldIs NotEqualTo 0 actuator_number)
IF NOT (FieldIs EqualTo 0 actuator_number)
```

IF clauses may also be put in GROUP statements following the name.

IF clauses depend on Boolean Methods that test for some condition and return True or False. The Boolean method FieldIs checks the value in a field we have already decoded.

FieldIs Method: FieldIs is a Boolean Method that compares the value of a previously decoded field with a constant. The first parameter is a relational operator (LessThan, LessThanOrEqualTo, EqualTo, NotEqualTo, GreaterThan, GreaterThanOrEqualTo). The second parameter is the number that the field value is to be compared to. The name of the field is supplied as the third parameter.

To distinguish between the Set Actuator and Read Sensor commands, we need to find out if the Command field is equal to 1 (Set Actuator) or 2 (Read Sensor). We add IF statements as follows (new items are in bold):

```
FIELD command_code (Fixed 1 Byte) (Table tableCommands) IN_SUMMARY
"Command/Response" 150 "Command"

GROUP cmdSetActuator IF (FieldIs EqualTo 1 command_code) "Set Actuator"
{
        FIELD actuator_number (Fixed 1 Byte) (Decimal) "Actuator Number" VERIFY
        (FieldIsBetween 1 32)

        FIELD actuator_value (Fixed 2 Bytes) (Decimal) "Value to set"
  }

GROUP cmdReadSensor IF (FieldIs EqualTo 2 command_code) "Read Sensor"
{
        FIELD sensor_number (Fixed 1 Byte) (Decimal) "Sensor Number" VERIFY
        (FieldIsBetween 1 32)
}
```

```
FIELD checksum (Fixed 1 Byte) (Hex) IN_SUMMARY "C'sum" 40 "Checksum" VERIFY
(FPChecksum)
```

The checksum field has also been removed from both groups because it is identical for both commands. Since the Read Sensor group only has one field in it, we could also write the decoder without the Read Sensor group and put the IF clause on the sensor_number field, as shown below, but this makes the decoder more difficult to read.

```
FIELD command_code (Fixed 1 Byte) (Table tableCommands) IN_SUMMARY
"Command/Response" 150 "Command"

GROUP cmdSetActuator IF (FieldIs EqualTo 1 command_code) "Set Actuator"
{
        FIELD actuator_number (Fixed 1 Byte) (Decimal) "Actuator Number" VERIFY
        (FieldIsBetween 1 32)

        FIELD actuator_value (Fixed 2 Bytes) (Decimal) "Value to set"
  }

FIELD sensor_number (Fixed 1 Byte) IF (FieldIs EqualTo 2 command_code) (Decimal)
"Sensor Number" VERIFY (FieldIsBetween 1 32)

FIELD checksum (Fixed 1 Byte) (Hex) IN_SUMMARY "C'sum" 40 "Checksum" VERIFY
(FPChecksum)
```

When you need to handle both sides of a test, the IF and the IF NOT, it is necessary to explicitly test for one condition and then the reverse; there is no ELSE facility to create this effect.

The BRANCH Statement

Now we run into another dilemma. We have one more command and three more responses to decode. While we could use groups and conditionals to decode all six commands and responses, this is not very efficient and quickly becomes unwieldy for protocols with large numbers of commands. The BRANCH keyword solves this dilemma.

We noted that the FIELD statements we quoted above serve to decode a Set Actuator and Read Sensor command in FP. If you look at the full FP decoder (on file FP.DEC), you will not find those statements appearing as such. What you will see is:

```
FIELD command_code (Fixed 1 Byte) (Table tableCommands)
  IN_SUMMARY "Command/Response" 150 "Command"

BRANCH (FromTable tableCommands command_code)

FIELD checksum ;
```

The BRANCH statement causes a jump to a statement specific to the command code. The target of the branch is provided by a Branch Method, in this case FromTable, which extracts the branch item from the table entry specified by the parameters.

BRANCH Syntax:

This statement starts with the keyword BRANCH followed by a call to a Branch Method that obtains a reference to the statement to be branched to. The only commonly used Branch Method is FromTable. The target of a BRANCH may be either a FIELD statement or a GROUP (discussed in the next section below).

The FromTable method finds the entry in a given table that matches a given value and returns the branch item from it. The first parameter (tableCommands) is the name of the table; the second (command_code) is a fieldname. The value of that field serves as the index. If the matching entry has no branch item, the method returns a value that suppresses a branch.
The fact that we can refer to the command_code field in our BRANCH reveals one crucial fact about how the protocol analyzer works: when a field is decoded, its value is not merely extracted, formatted and displayed but it is also saved. And that saved value may be used by subsequent DecoderScript statements. This is absolutely necessary in situations where the value in one field determines the length of another, and can be very handy in several other circumstances. Field values are automatically saved for the duration of the decoding of the layer.

You can also save data to be used by other layers in the same frame (intra-frame) or by other frames (inter-frame). These data persist for longer than the decoding of the layer. See the section on Intra-frame and Inter-frame Data for information on how to do this.

We will explain later what information you need to add to the Command table to allow the branch to occur. For now, assume that information is there. Because of the branch, we can remove our Set Actuator and Read Sensor groups from the main path of the decoder, and put them outside the main section. As the protocol analyzer moves through the decoder, the branch will cause it to jump to the appropriate group, decode that group, and then return to the main path at the same point as before. After altering the decoder, it looks like this:

```
FIELD command_code (Fixed 1 Byte) (Table tableCommands)
        IN_SUMMARY "Command/Response" 150 "Command"

BRANCH (FromTable tableCommands command_code)

FIELD checksum (Fixed 1 Byte) (Hex)
        IN_SUMMARY "C'sum" 40 "Checksum" VERIFY (FPChecksum)

END_MAIN_PATH

GROUP cmdSetActuator
{
            FIELD actuator_number (Fixed 1 Byte) (Decimal) "Actuator Number" VERIFY
            (FieldIsBetween 1 32)
```

```
        FIELD actuator_value (Fixed 2 Bytes) (Decimal) "Value to set"
    }
GROUP cmdReadSensor
{
        FIELD sensor_number (Fixed 1 Byte) (Decimal) "Sensor Number" VERIFY
        (FieldIsBetween 1 32)
}
```

The cmdSetActuator and cmdReadSensor groups have been moved after the END_MAIN_PATH keyword, and the BRANCH statement inserted after the Command field. The last piece of information is how to alter the Command Table.

Branching Information in Tables

Our original command table for FP looked like this:

```
TABLE tableCommands
{ 0 "Request Status" "Request Status"}
{ 1 "Set Actuator" "Set Actuator"}
{ 2 "Read Sensor" "Read Sensor"}
{ Default "???" "(unknown)"}
ENDTABLE
```

In order to use a BRANCH, we need to add some additional information to the table. Remember that the first item in each table entry is the field value to look up, and the second and third entries are the strings for display in the Summary and Decodes panes. Now we need to add the information for the BRANCH to extract. The resulting table looks like (changes are in bold):

TABLE tableCommands
{ 0 "Request Status" "Request Status"}
{ 1 "Set Actuator" "Set Actuator" **0 cmdSetActuator**}
{ 2 "Read Sensor" "Read Sensor" **0 cmdReadSensor**}
{ Default "???" "(unknown)" **0 Unknown**}
ENDTABLE

The fourth item in each table entry contains a number which is called the extra table data. The extra table data can be used to hold any number that would be useful in the decoding process. In the FP decoder we do not make use of the extra data field but must include a value for it, as a place-holder, when we have another item following. The last item identifies a DecoderScript statement, either a FIELD or a GROUP, that performs decoding specific to the command. With our new table, the BRANCH (FromTable tableCommands command_code) instructs the protocol analyzer to look up the value of command_code in tableCommands and jump to the GROUP or FIELD named at the end of the table entry for that value.

Our FP decoder now looks like this:

```
"Fictitious Protocol" 0x7f000000

DECODE

TABLE tableCommands
{ 0 "Request Status" "Request Status"}
{ 1 "Set Actuator" "Set Actuator" 0 cmdSetActuator}
{ 2 "Read Sensor" "Read Sensor" 0 cmdReadSensor}
{ Default "???" "(unknown)" 0 Unknown}
ENDTABLE

FIELD command_code (Fixed 1 Byte) (Table tableCommands)
        IN_SUMMARY "Command/Response" 150 "Command"

BRANCH (FromTable tableCommands command_code)

FIELD checksum (Fixed 1 Byte) (Hex)
        IN_SUMMARY "C'sum" 40 "Checksum" VERIFY (FPChecksum)

END_MAIN_PATH

GROUP cmdSetActuator
{
            FIELD actuator_number (Fixed 1 Byte) (Decimal) "Actuator Number" VERIFY
            (FieldIsBetween 1 32)

            FIELD actuator_value (Fixed 2 Bytes) (Decimal) "Value to set"
   }

GROUP cmdReadSensor
{
            FIELD sensor_number (Fixed 1 Byte) (Decimal) "Sensor Number" VERIFY
            (FieldIsBetween 1 32)
}
```

The statements within the cmdSetActuator and cmdReadSensor GROUPs are grouped simply for the purpose of making them a suitable object of a BRANCH. The point is that the object of a BRANCH is always a statement – it cannot be an unconnected sequence of statements because there is nothing akin to a “return” statement such as one finds in most programming languages. It can be a single statement, that is a FIELD, or a compound statement, namely a GROUP. Once that statement has been processed, control is automatically returned to the statement following the BRANCH.

You may have noticed that Request Status doesn't have any branching information. This command consists solely of the Command Code and the checksum, so there is nothing to branch to. In this situation the BRANCH statement is ignored and decoding continues on to the next statement.

SUPPRESS_DETAIL and NO_MOVE

Adding our last command to our decoder is a trivial task at this point. However, we also have three responses to decode. The FP decode says that the first byte of a frame is the command or response code; however, we don't know whether the frame contains a command or a response until we decode it. We could use the label "Command/Response" for the first byte, but that's not as easy for the user to understand as if we said either "Command" or "Response" depending on which it was.

Towards the beginning of the tutorial, we explained how Size methods move the bit pointer through the data. It would be useful if we could decode the first byte in the layer, look at it, decide if it's a command or a response, and only then display the field with the right detail tag. To do this, we would need to decode the field, prevent it from being displayed in the Decode pane, and leave the bit pointer at the beginning of the field when we're finished so it's in position to decode the field properly once we know if it's a command or a response.

The keyword NO_MOVE tells the protocol analyzer to leave the bit pointer where it is after decoding the field. Why do this? Because the command and response statements decode the same data again. (You could achieve the same effect by adding a StartBit method, described later, to the first field in the group but in this case, using NO_MOVE is a much cleaner choice.)

NO_MOVE Syntax:

A NO_MOVE keyword goes after the tag and before any VERIFY clause.

The keyword SUPPRESS_DETAIL is used to decode a field without displaying the contents in the Decode pane.

SUPPRESS_DETAIL Syntax:

A SUPPRESS_DETAIL keyword is used in place of the label.

By combining these two keywords on the same field, we can decode it without displaying the results and without moving the pointer. We can then use the value in the field to determine whether to label it a Command or a Response and display the proper string from the Commands table.
Note that when we decode an already decoded field in a new way, we are obliged to give it a new name, command or response in this case. Here's how it looks:

```
FIELD command_or_response (Fixed 1 Byte) (Hex) SUPPRESS_DETAIL NO_MOVE

FIELD command (Fixed 1 Byte) IF (FieldIsBetween 0x00 0x02 command_or_response)
(TABLE tableCommands) IN_SUMMARY "Command/Response" 0 "Command"
```

```
GROUP response_group IF (FieldIsBetween 0x80 0x82 command_or_response)
{
        FIELD response (Fixed 1) (TABLE tableCommands) IN_SUMMARY
                "Command/Response" 0 "Response"
    FIELD response_count (Fixed 1) (Decimal) "Data byte count"
                VERIFY (FieldIsBetween 1 3)
}

BRANCH (FromTable tableCommands command_or_response)

FIELD checksum (Fixed 1 Byte) (Hex)
    IN_SUMMARY "C'sum" 40 "Checksum" VERIFY (FPChecksum)

END_MAIN_PATH
```

Responses have the additional response_count field giving the number of data bytes in the response. We created a response_group to join together the response and response_count fields and put the conditional on the group.

The GROUP FIELD and RESERVED Statements

Now let us look at the statements for decoding an FP status response since that illustrates the decoding of bit fields plus two new statement types. It is also another example of the NO_MOVE keyword.

```
GROUP FIELD rspStatus (Fixed 3 Bits) (Hex) "Status" NO_MOVE
{
   RESERVED (Fixed 5 Bits)
      FIELD alarm_bit (Fixed 1 Bit) (Binary) "Alarm"
   FIELD power_bit (Fixed 1 Bit) (Binary) "Power"
   FIELD test_bit (Fixed 1 Bit) (Binary) "Test mode"
}
```

We have met FIELDs and GROUPs and here we encounter a GROUP FIELD. This statement combines some of the best features of FIELD and GROUP. In particular it allows us to create a statement group (with braces) and at the same time display some data representative of the group – which is something a plain GROUP cannot do.

GROUP FIELD Syntax:

Statement consists of a FIELD statement prefixed with the keyword GROUP.

The FP status response consists of a set of bit flags. The GROUP FIELD displays these flags as a batch; that is, it formats the whole byte as a hex number. (It could, by the way, use the Binary method instead of Hex and display them in binary notation.) The GROUP FIELD also creates a sub-tree in the Decode Pane. If the user expands the sub-tree then the items within the group are shown individually.

Here are two snapshots of the Decoder Pane, one showing the sub-tree closed and the other expanded:

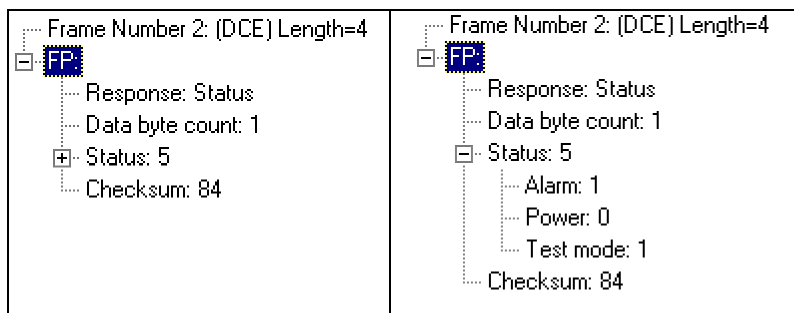


Figure 10 - GROUP FIELD Collapsed and Expanded

The second new statement seen here is RESERVED. This tells the protocol analyzer to skip over a certain number of bits that are not used. Nothing is displayed as a result of it.

RESERVED Syntax: Statement starts with the keyword RESERVED followed by a call to a Size Method that specifies the size of the reserved field.

Finally, for this statement sequence, note the NO_MOVE keyword on the GROUP FIELD statement. This tells the protocol analyzer not to move the pointer after decoding the field.

START_BIT

There's a bug in our rspStatus group. Here's the code again:

```
GROUP FIELD rspStatus (Fixed 3 Bits) (Hex) "Status" NO_MOVE
{
    RESERVED (Fixed 5 Bits)
          FIELD alarm_bit (Fixed 1 Bit) (Binary) "Alarm"
    FIELD power_bit (Fixed 1 Bit) (Binary) "Power"
    FIELD test_bit (Fixed 1 Bit) (Binary) "Test mode"
}
```

Notice that the first 5 bits are skipped in the RESERVED statement, and the last three are decoded. The GROUP FIELD, however, decodes the first three bits in the byte. In order for the GROUP FIELD to decode the correct 3 bits, we need to move the bit pointer 5 bits forward first.

START_BIT Methods are used to move the bit pointer forwards and backwards in the layer. The Move method simply moves the bit pointer backwards or forwards the specified number of bits. Other methods move the pointer to certain fields, or move the pointer to the beginning or end of the layer.

START_BIT Syntax:

A START_BIT clause is placed after the name. The keyword is followed by a call to a StartBit method that serves to reposition the pointer before decoding of the field starts.

Move Method: Move is a StartBit method that moves the pointer a specified distance from its current position. The first parameter is a count and the second a unit. If the second parameter is omitted then “bytes” is assumed. If the count is negative then the pointer is moved backwards; if positive then the move is forwards.

To fix our bug, we'll add a START_BIT (Move 5 Bits) clause to our rspStatus group.

```
GROUP FIELD rspStatus START_BIT (Move 5 Bits) (Fixed 3 Bits) (Hex) "Status" NO_MOVE
{
    RESERVED (Fixed 5 Bits)
        FIELD alarm_bit (Fixed 1 Bit) (Binary) "Alarm"
    FIELD power_bit (Fixed 1 Bit) (Binary) "Power"
    FIELD test_bit (Fixed 1 Bit) (Binary) "Test mode"
}
```

Now our GROUP FIELD will decode the correct 3 bits.

Comments

Here is some DecoderScript used in decoding sensor commands and responses:

```
/*
Sensor number. This is a one-byte field. We display it in decimal and verify that it is valid
(i.e. in the range 1 to 32).
*/
FIELD sensor_number (Fixed 1) (Decimal) "Sensor Number"
    VERIFY (FieldIsBetween 1 32)

/*
Sensor value. This is a two-byte field. */
FIELD sensor_value (Fixed 2) (Decimal) "Sensor Value"
```

The new element is the comment. Comments are written as in the C++ programming language, either starting with “/*” and ending with “*/” or starting with “//” and ending at the end of the line. Comments serve solely to aid people reading the decoder; the protocol analyzer ignores them.

Comment Syntax:

A block comment starts with a slash and an asterisk written together and ends with an asterisk and a slash also written together. A single block comment may extend over several lines.

An end-of-line comment starts with two slashes and ends at the end of the line.

The FP Decoder

Here's the final version of the FP decoder in its full glory:

```
/* Decoder for FP (Fictitious Protocol) by Manuel Ryder */
"FP" 0x7f000000

DECODE

/*
** Table of FP command and response codes.
*/
TABLE tableCommands
{ 0x00 "Request Status" "Request Status"}
{ 0x01 "Set Actuator" "Set Actuator" 0 cmdSetActuator}
{ 0x02 "Read Sensor" "Read Sensor" 0 cmdReadSensor}
{ 0x80 "Status" "Status" 0 rspStatus}
{ 0x81 "Actuator Set" "Actuator Set" 0 rspActuatorSet}
{ 0x82 "Sensor Value" "Sensor Value" 0 rspSensorValue}
{ Default "???" "(unknown)" 0 Unknown}
ENDTABLE

/*
** Find the value of the first field, but don't display it and
** don't move the pointer.
*/
FIELD command_or_response (Fixed 1) (Hex) SUPPRESS_DETAIL NO_MOVE

/*
** If the value of the command_or_response field is between 0x00
** and 0x022, we know it's a command. If it's between 0x80 and
** 0x82, we know it's a response. The response has an extra
** field that commands don't have, so we create a group and put
** the IF test on the group as a whole. If the test fails, the
** entire group will be skipped.
**
** Unknown commands will fail both tests and branch to the
** Unknown group, which decodes the first byte as an Unknown
** Command or Response.
**
** The IN_SUMMARY puts the result of either field in the same
** Summary pane column.
*/

FIELD command (Fixed 1) IF (FieldIsBetween 0x00 0x02 command_or_response) (TABLE
tableCommands) IN_SUMMARY "Command/Response" 120 "Command"
```

```
GROUP response_group IF (FieldIsBetween 0x80 0x82 command_or_response)
{
    FIELD response (Fixed 1) (TABLE tableCommands) IN_SUMMARY "Command/Response"
120 "Response"
    FIELD response_count (Fixed 1) (Decimal) "Data byte count" VERIFY (FieldIsBetween 1 3)
}

BRANCH (FromTable tableCommands command_or_response)

/*
** Display and verify what should be the checksum.
*/
FIELD checksum (Fixed 1) (Hex) IN_SUMMARY "C'sum" 40 "Checksum" VERIFY
(FPChecksum)


END_MAIN_PATH

/*
**
** Common fields. In this section we have FIELD statements for
** fields that are used in multiple places in the decoder.
**
** Actuator number. This is a one-byte field. We display it in
** decimal and verify that it is valid (i.e. in the range 0 to
** 32, where 0 means "all").
*/
FIELD actuator_number (Fixed 1) (Decimal) "Actuator Number"
    VERIFY (FieldIsBetween 0 32)

/*
** Actuator value. This is a two-byte field.
*/
FIELD actuator_value (Fixed 2) (Decimal) "Value to set"

/*
** Sensor number. This is a one-byte field. We display it in
** decimal and verify that it is in the range 1 to 32).
*/
FIELD sensor_number (Fixed 1) (Decimal) "Sensor Number"
    VERIFY (FieldIsBetween 1 32)

/*
** Sensor value. This is a two-byte field.
*/
FIELD sensor_value (Fixed 2) (Decimal) "Sensor Value"
```

```
/*
**
** Decode commands.
**
    */

GROUP cmdReadSensor
{
    FIELD sensor_number ;
}

GROUP cmdSetActuator
{
    /*
              ** We want to display something different if the actuator
              ** number is 0 (meaning "All Actuators"), so we need to
              ** process the field first, and then use that field as a
              ** test for the IF statement.
    */
    FIELD actuator_id (Fixed 1) (Decimal) SUPPRESS_DETAIL NO_MOVE

    FIELD all_actuators (Fixed 1) IF (FieldIs EqualTo 0 actuator_id) (Constant "All Actuators")
"Actuator Number"

    FIELD one_actuator (Fixed 1) IF (FieldIs NotEqualTo 0 actuator_id) (Decimal) "Actuator
Number" VERIFY (FieldIsBetween 1 32)

    FIELD actuator_value ;
}

/*
**
** Decode responses.
**
*/

GROUP FIELD rspStatus START_BIT (Move 5 Bits) (Fixed 3 Bits) (Hex) "Status" NO_MOVE
{
          RESERVED (Fixed 5 Bits)
    FIELD alarm_bit (Fixed 1 Bit) (Binary) "Alarm"
    FIELD power_bit (Fixed 1 Bit) (Binary) "Power"
    FIELD test_bit (Fixed 1 Bit) (Binary) "Test mode"
}

GROUP rspActuatorSet
{
```

```
    FIELD actuator_number ;
}

GROUP rspSensorValue
{
    FIELD sensor_number ;
    FIELD sensor_value ;
}

GROUP Unknown
{
    FIELD unknown_cmd_or_resp (Fixed 1) (Hex) IN_SUMMARY "Command/Response" 120
"Unknown Command/Response Code"
    FIELD unknown_data (ToEndOfLayer 1) (StringOfHex) "Data"
}
```

Chapter 4: Putting It All Together

Decoder IDs

All decoders require a unique ID. This ID is used to determine the next protocol in the stack and other internal housekeeping chores. Decoders can be distributed as plain text, or they can be encrypted. Custom decoders of either type can only be loaded if the protocol analyzer has been installed with the option that allows creation of decoders.

Encrypted Decoders

If you plan to distribute your decoder to the general public, or to a large group of people within your own company, contact your protocol analyzer supplier for a Publisher ID. Publisher IDs are the first two hex digits of the decoder ID. Be sure to use a different number for each decoder file. Since no one else will have your Publisher ID, your decoder IDs will be unique.

For example, if your Publisher ID is 0x3045, your first decoder might be 0x30450000, your second 0x30450001, etc.

If you need to encrypt your decoders, you MUST have a Publisher ID. Along with the ID, we will send you a utility for encrypting your decoders. (See the readme for the utility for more information on how to use it.) In order for a user to load your encrypted decoder, the decoder file must have their product serial number embedded in the file. You will need to get the user's serial number, enter it in the utility, and the utility will encrypt the decoder with the user's serial number. The utility will give you a magic number to give to the user, which the user will use when installing the decoder. The combination of the magic number and the product serial number become a key to unlock the decoder.

Plain Text Decoders

If you don't need to encrypt your decoder and are not planning to distribute to many other people, you don't need a Publisher ID. Just use Publisher ID 0x0000 followed by a number, and distribute your decoders in plain text. Using this Publisher ID, decoder numbers 0x00000000 through 0x0000ffff are reserved for plain text decoders.

[ProtocolName].personality files

[ProtocolName].personality contains information for decoders included with the protocol analyzer. For decoders that you write, appropriate information should be added to your own [MyProtocolName].personality file in the My Decoders directory. The part in brackets doesn't have to be the protocol name, but it's a useful way of keeping track of which personality file goes with which protocol. The personality files for the core product are located in directories named for each protocol under <installation path>\App Data\Decoders. Each protocol directory contains the decoders needed for that protocol, as well as a file called [protocol name].personality. We suggest you take a look at a personality file installed on your system while working through this chapter. Personality files can be read in any text editor. A sample

personality file for the Fictitious Protocol, fp.personality, is located in the My Decoders directory of the installed product.

Personality files contain several sections. The Rules section defines rules for transitioning from one protocol layer to the next. The Predefined Stacks section lists predefined stacks that the user can select from the Predefined Stack selection box in the Protocol Stack Wizard. Port Assignments define which protocol to autotraverse to given a port number. The Filters section defines filters to appear in the Predefined Filters box in the Filters window.

Other sections you may see in the personality files included with the protocol analyzer are used for starting the software. If you are writing decoders for serial or Ethernet analyzers, you do not need to pay any attention to the extra sections. If you are writing decoders for industrial automation analyzers (these analyzers usually have a launcher where you can select which analyzer to use based on the protocol you want to decode), then see Appendix 2 on How to Insert Fictitious Protocol into a Fictitous Product for instructions on adding your protocol decoder name to the launcher.

Rules

The Rules section defines rules for what is called “autotraversal”. Autotraversal is the process of switching from one decoder to the appropriate decoder for the protocol at the next layer. In some cases one protocol always follows another but, in most cases, data in the current-layer protocol must be used to determine which of several possible protocols comes next. A few examples should make this concept clear.

Here is a fragment from ip.personality relating to IP. IP is an obvious case study since a large number of protocols can be used at the next layer.

```
[Rules]
0x7f000401,0x7f000407,0x01        IPv4 - ICMP
0x7f000401,0x7f000404,0x06        IPv4 - TCP
0x7f000401,0x7f000405,0x11        IPv4 - UDP
0x7f000401,0x7f000406,0x02        IPv4 – IGMP
```

Rule Definition   Port Assignment Display

The rule is defined by the three items in each entry in the list. List items are separated by commas, and there should be no spaces between list items. The rule is processed until a space, tab, or paragraph character appears. Anything after a space, tab or paragraph is treated as a comment and ignored for autotraversal purposes.

However, the “Port Assignment Display” section is used to display protocol names on the port assignments tabs in the “Set Initial Decoder Parameters” dialog. This entry follows a space or tab character after the rule definition, and must be in the form of:

<this-protocol name><space><dash><space><next-protocol name>.

In the rule definition, the first two items are decoder IDs. In particular, 0x7f000401, as you might guess, is the ID for IP. The other IDs are for protocols that may follow IP. The third number is a key that links the two. Specifically, this key is a value that may be returned by the NextProtocol method in IP. In this case, it is actually the value extracted from the PROT field of the IP header, a code that directly identifies the next protocol. A value of 1 means the next protocol is ICMP and, if you look in the ICMP decoder, you will see that it has an ID of 0x7f000407. Hence, the first rule.

Once a layer in a stack has been decoded, the NextProtocol method for that layer is invoked. The result, together with the ID of the current protocol, is used to search the rules table for an entry that will determine which decoder to use next. As this implies, a key need be unique only within the set of entries for a particular protocol. You might look at a rule in the following way:

<this-ID>,<next-ID>,<key>

means:

if the ID of the current protocol is <this-ID> and the value returned by its NextProtocol method is <key> then the next protocol is the one with ID <next-ID>

What happens if there is no match in the rules table for a given ID and key combination? This does not necessarily imply that anything is wrong; it just means that there is no decoder for the next protocol. It was noted that over a hundred protocols could follow IP. However, examination of a current ip.personality will reveal far less than one hundred entries – many decoders have not yet been written!

Another example entry from async.personality is:

0x7f000303,0x7f000302,ALWAYS

As you would guess, the keyword ALWAYS means that the protocol with ID 0x7f000302 (which happens to be PPP) always follows Async PPP (ID 0x7f000303).

In many other examples, you will see that the second decoder ID and the key are the same. All the entries for TCP are like this, as in:

0x7f000404,0x7f00040a,0x7f00040a

where the second ID happens to be for NBSS (NetBIOS Session Services). The point is that the key can be anything that happens to be convenient. With TCP, the actual decoder ID serves well since the protocol is determined by looking up a port number in the Port Assignments section (see below).

Predefined Stacks

The Predefined Stacks section contains the protocol stacks that are listed in the Predefined Stack list box in the Protocol Stack Wizard. The format of an entry is

```
<stack-name>,Auto|No,<protocol-ID>,[<protocol-ID>,…]
```

where:

<stack-name> is the name that appears in the list box.

“Auto” means that autotraversal is to be used and the protocol analyzer should attempt to determine what protocol comes next in the frame.

“No” means that autotraversal should not be used. In this situation, every frame will be decoded using the protocol(s) given in the list, in the order they are given. Any frame that does not use the defined stack will be decoded incorrectly.

<protocol-ID> is the ID of the protocol (if standalone) or the ID of the lowest-layer protocol (if in a stack).

Additional decoder IDs may be appended to an entry to further define a stack. Thus we see:

```
Frame Relay/HDLC,No,0x7f000101,0x7f000010,
Frame Relay/SNAP,Auto,0x7f000101,0x7f000204,
```

to differentiate HDLC over Frame Relay and SNAP over Frame Relay.

Predefined Stacks are sorted into alphabetical order in the program, so it does not matter what order they appear in the .personality file.

Port Assignments

The job of the Port Assignments section is to return the key used by the Rules section in the personality files by looking up the port number and returning the value next to it.

There are several Port Assignments sections based on which protocol port numbers to use. The section name appears after the Port Assignments name in the section header. The section name is referenced in the NEXT_PROTOCOL FromPortAssignment method as the first parameter. For example, the TCP decoder header has:

```
NEXT_PROTOCOL (FromPortAssignment TCP src_port dst_port)
```

The first parameter indicates the port numbers in the TCP section should be used, and the other sections ignored. The second and third parameters are the field names for the source and destination ports.

Within a section, the format of each entry is

[Port Assignments:section name]
<port number> ,<key>

For example, the section for TCP given in ip.personality is:

[Port Assignments:TCP]
25,0x7f000409
53,0x7f00040d
80,0x7f000408
138,0x7f00040b
139,0x7f00040a
5000,0x7f009006
67-68,0x7f000412

The first value is the port number. Notice that a range of port numbers may be specified and the range is inclusive. The port number may be either the source or destination port. If a match is found for the port number, the second value, the key, is returned to the NEXT_PROTOCOL method. The key may be any unique number.

This key is then used during a scan of the Rules section. For TCP, the protocol analyzer uses the next decoder ID as the key, but we could have just as easily used the port number. Any value can be used as the key. (For various technical reasons we could not use the port number as the key in the personality files and skip using the Port Assignments section entirely.)

If you use a port number already listed in the Port Assignments section for a different protocol than the standard one, you only need to change the entry in the appropriate personality file. Simply change the second decoder ID (which is the one to traverse to) to the decoder ID for your decoder.

Filters

The Filters section defines filters that will appear in the predefined filter section. This is mostly a convenience for your users because they can very quickly select a predefined filter and they don't have to think about how to define it.

The format is

[filter name],[location in tree],[compound filter],Include|Exclude,[bpf string]

where:

[filter name] is the name that will appear in the filters window.

[location in tree] is where you want it to appear in the tree that is used to organize filters in the predefined window.

[compound filter] is the name of the compound filter that this filter is a part of (note that compound filter is rarely used in predefined filters).

Include|Exclude defines whether the filter should include data that matches the filter or exclude it.

[bpf string] is the filter string in BPF format.

For information on creating filter strings using BPF, see the online Help for the protocol analyzer product.

What to Send to End-Users

A complete decoder package requires the following components:

- The decoder file with a unique decoder ID. The file must have a DEC extension for plain text decoders and DEX for encrypted decoders. This file must be placed in the My Decoders directory, but can be placed in a subdirectory under My Decoders. The protocol analyzer reads the My Decoders directory and the user-defined subdirectories for custom decoders.
- A protocol analyzer product that has the option for creation of decoders if decoders are distributed as DEC files OR a “fileset” that contains the decoder if it is to be distributed in encrypted form. Please see the information about filesets at the end of this section.

In certain cases, some or all of the following are required as well:

- A Frame Recognizer method.
- Any other custom methods used by the decoder.
- A magic number for installation of the decoder (encrypted decoders only).
- A [MyProtocolName].personality file.

Methods must be compiled into a DLL and placed in the My Methods directory in order for the decoder to find them. [MyProtocolName].personality must be placed in the My Decoders directory, but can be placed in a subdirectory under My Decoders. The protocol analyzer reads the My Decoders directory and the user-defined subdirectories for custom decoders.

You may also optionally wish to send all or one of the following:

- Pre-defined protocol stack in [MyProtocolName].personality to appear in the Protocol Stack Wizard.
- I/O Settings Configuration file for serial analyzers.
- SelectorChoices.txt for industrial automation analyzers (see Appendix 2).

A pre-defined protocol stack may be useful if there are several protocols in your stack and you need them to be set up in a certain way. If you include a pre-defined stack, your users just need to select that one item in the Protocol Stack Wizard and you're guaranteed to have the stack set up correctly.

To supply an I/O Settings Configuration, start the protocol analyzer, open the I/O Settings window, and setup the parameters exactly as you want them to be. Save the configuration. The file will be saved to the My Configurations\filename.cfg directory by default. Include the CFG file with your decoders and have the user place it in their My Configurations directory. Once the user starts the protocol analyzer, they will need to open the I/O Settings window and load the configuration, but once they do, you will be sure what the parameters are set for.

A convenient way to supply all of these components to an end-user, and ensure they are placed in the correct directories, is to distribute them in a fileset. For more information on filesets and the tools used to create them, please contact your software vendor's Technical Support department.

Chapter 5: DecoderScript Reference

If you have the relevant background, it may be helpful to think of a decoder as the description of a large table in which each entry defines a field, a group, or a branch. The protocol analyzer traverses this table starting at the top, jumps around as instructed, and terminates when it meets an END_MAIN_PATH. (If you lack experience with such languages, don't worry; it is absolutely not required!)

Yet another way of looking at DecoderScript is as a statement language that controls the interface between the protocol analyzer and a set of methods that perform the nitty gritty of decoding.

We now describe each of these reference statements in turn.

BYTE_STREAM_FRAMER

BYTE_STREAM_FRAMER identifies a method that is sort of like a frame recognizer, but it only acts on the data found at the current layer. Because of this, it has some different needs and is implemented as a different type. This method is only invoked if the layer that was just decoded (i.e. the previous layer in the stack) has the PAYLOAD_IS_BYTE_STREAM keyword in it.

BYTE_STREAM_FRAMER is used when the payload of a particular layer is simply a byte stream that the upper layers know nothing about. For instance, imagine a protocol where the payload is always exactly 50 bytes long, and the actual messages are split across as many frames as is needed. If all the payloads are clipped out of the frames and put together, it would resemble the byte stream that comes from a serial port. We would need some type of recognizer to split that byte stream into intelligible frames, but standard frame recognizers work only on data that has not previously been framed by any other method.

FRAME_AFTER_DECODING

FRAME_AFTER_DECODING references a Frame Transformer method that processes a frame after it has been decoded. Such methods are seldom used but serve a purpose when the decoder for one layer in a protocol stack needs to mess with the frame to prepare it for the decoder at the next layer.

A subtlety of these methods is that if FRAME_AFTER_DECODING replaces a frame, then any NEXT_PROTOCOL_SIZE and NEXT_PROTOCOL_OFFSET methods are ignored and the entire replaced frame is passed to the next layer. The big subtlety here is that the transformation methods don't always transform; for instance, a stripper routine that doesn't find a 0x7d in the frame won't do anything to that frame, and then the NEXT_PROTOCOL_SIZE and _OFFSET will be used for those frames.
In the IP example above, there may be padding at the end of the frame, so we strip off any bytes after the stated size. Alternately, the payload may be split over multiple frames, and if so, and this frame has the last discovered fragment, then we assemble all the fragments and send the assembled frame to the next layer instead of the physical frame.

FRAME_BEFORE_DECODING

FRAME_BEFORE_DECODING references a Frame Transformer method that prepares the frame for decoding. This may entail such matters as decryption, decompression, undoing character escaping and byte-stuffing. IP does not need one of these.

NEXT_PROTOCOL

NEXT_PROTOCOL is a method that can be used to determine the protocol at the next layer in a stack. In the case of IP for example, this would determine if the next layer is TCP, UDP, or one of many other less common possibilities. The method returns a value that is looked up in an personality file to retrieve the standard FTE id for the next protocol. (Now you know one reason why every protocol is required to have a unique id.)

IP has a field, "protocol", that has a unique value and determines the next protocol to use, so we call a method that picks that field out.

NEXT_PROTOCOL_SIZE

NEXT_PROTOCOL_SIZE is a method used to determine the size of the next protocol layer. IP uses a method, ThisLayerLength, that is passed the name of the field that contains the length (tot_length) and a fixed increment that is added (zero in this case).

NEXT_PROTOCOL_OFFSET

NEXT_PROTOCOL_OFFSET is a method used to determine the offset of the next protocol layer. By default the protocol analyzer assumes that the next layer starts at the next byte in the frame; this is typically the case so a NextProtocolOffset method is rarely needed.

PAYLOAD_IS_BYTE_STREAM

PAYLOAD_IS_BYTE_STREAM is used in conjunction with BYTE_STREAM_FRAMER for cases where the payload for that layer is just one part of a byte stream that includes the payloads from frames before and/or after it. It means that the payload of all the coming layers should be considered a continuous stream and the divisions of the physical frames should be ignored.

For example, assume you have a custom protocol called MyCustomProtocol. This protocol traverses to higher layers in the stack. The payload for MyCustomProtocol can be split up among several physical frames. In order to correctly decode MyCustomProtocol, you need to take the MyCustomProtocol payloads from several frames and string them all together. PAYLOAD_IS_BYTE_STREAM tells the decoder to do this.

Then you need a way to tell when one MyCustomProtocol layer ends and the next begins, very much like a frame recognizer. For efficiency and technical reasons, we can't use the real frame recognizer, so we created a way to make a frame recognizer for cases like this. BYTE_STREAM_FRAMER invokes a method that acts as a frame recognizer for these embedded byte streams.

To use this feature, put the PAYLOAD_IS_BYTE_STREAM keyword in the header of your BYTE_STREAM_FRAMER keyword in the header and reference a method that can correctly determine when the MyCustomProtocol layer is finished.

When MyCustomProtocol traverses to IP, the BYTE_STREAM_FRAMER method is invoked and the payloads of the next frames are joined to the payload of the first frame until the end of the MyCustomProtocol frame is reached.

If MyCustomProtocol should traverse to a protocol that doesn't have the BYTE_STREAM_FRAMER keyword in it, then the physical packets are the frame boundaries.

PREPROCESSING

PREPROCESSING references a method that does things before a layer is decoded. Such a preprocessor has access to the layer data (as transformed by the FRAME_BEFORE_DECODING method if there is one), uninitialized field values (ai64Field), and inter-frame data. Note that a PreProcessing method cannot make any changes to the frame; that is the province of the FRAME_BEFORE_DECODING method. You can have multiple PreProcessing invocations in a decoder. For example:

PREPROCESSING (DoThis)
PREPROCESSING (DoThat)

POSTPROCESSING

POSTPROCESSING references a method that does things after a layer has been decoded. Such methods serve a purpose when decoded data needs to be saved for use in decoding subsequent layers or frames. We call such data “intra-frame data” if it is used across multiple layers within a frame, and “inter-frame data” if it is used across multiple frames. We might, for example, save a command code so that we can properly interpret a command-specific reply. As with PreProcessing methods, you can have multiple PostProcessing methods in a decoder. Note that some simple forms of data are available that don't require PostProcessing methods.

RECOGNIZER

RECOGNIZER identifies a Frame Recognizer for the protocol. This Frame Recognizer will be called only if the protocol is the lowest in the current stack. This makes it safe to define a recognizer for a protocol that could appear at different layers in different stacks. In short, you can define a Frame Recognizer for a protocol and be sure that the protocol analyzer will use it only if and when appropriate.

SUMMARY_COLUMNS

SUMMARY_COLUMNS defines the name, order and default column width for the summary columns. This keyword does not reference a method, but instead references the names of the columns.

The syntax for the SUMMARY_COLUMNS field is:

```
SUMMARY_COLUMNS
{
        "summary_label"  width
        "summary_label"  width
        etc.
}
```

Columns will appear in the order they are listed. The Frame Number, Timestamp, Delta, Length and Bookmark summary columns can be optionally included in the list using the following keywords: **$FRAME_NUMBER**, **$TIMESTAMP**, **$DELTA**, **$LENGTH**, and **$BOOKMARK**. Do not specify a width when using the special keywords. If the special keywords don't appear, then the columns are placed in their normal locations (bookmark and frame number at the beginning, the others at the end.)

For example:

```
SUMMARY_COLUMNS
{
        “Address” 80
        “Command” 145
        $DELTA
        “Parameter” 90
}
```

Once you've defined the columns using the SUMMARY_COLUMNS keyword, you still need to include the IN_SUMMARY keyword on the fields the column data comes from. Use the same column label as given in the SUMMARY_COLUMNS list. Leave out the column_width field on the IN_SUMMARY.

For example, using the list above as our starting point:

```
SUMMARY_COLUMNS
    {
        “Address” 80
        “Command” 145
        $DELTA
        “Parameter” 90
    }

DECODE

FIELD cmd (Fixed 4) (Table cmds) IN_SUMMARY "Command" "Server Command"
FIELD address (Fixed 1) (Decimal) IN_SUMMARY "Address" "Address"
FIELD param (Fixed 1) (Decimal) IN_SUMMARY "Parameter" "Command Parameter"
```

This will create the summary pane output of:

Bookmark Frame Number Address Command Delta Parameter Length Timestamp
Any summary columns defined in the SUMMARY_COLUMNS list must be defined on the appropriate fields using the IN_SUMMARY keyword. The reverse is also true. Any fields that use the IN_SUMMARY clause must be listed in the SUMMARY_COLUMNS list.

There are two disadvantages to using just the IN_SUMMARY clause. First, the columns are defined in the order that the fields are encountered in the decoder. The SUMMARY_COLUMNS keyword gives you more control over the column order. Second, if you define the same summary column name in two fields, the field is reused, but you still have to supply a default field width. The first field width encountered is the one that will be used, but this makes it difficult to know at a glance what the default width is.

The last entry in this table has the value Default. This serves as a catchall for values that have no entries. We urge you to include a default entry for every table except perhaps for small tables that have an entry for every possible field value.

Whenever a lookup value isn't found, the protocol analyzer will display both the Default entry (if there is one) and the decimal value it tried to look up.

Our first example showed a particularly simple table. Here is a fragment of another table that makes use of almost all items. This is from the SMB (Server Message Block) decoder and lists a few of the commands carried by the protocol:

```
TABLE commands
{ 0x00 "CREATE_DIRECTORY" "Create Directory" 0 create_new_dir}
{ 0x01 "DELETE_DIRECTORY" "Delete Directory" 0 delete_dir}
{ 0x02 "OPEN" "Open Specified File" 0 open_file}
{ 0x03 "CREATE" "Create New File" 0 create_or_open_file}
{ 0x04 "CLOSE" "Close File And Flush Buffers" 0 close_file}
{ 0x05 "FLUSH" "Flush All File Buffers To Disk" 0 flush}
{ 0x06 "DELETE" "Delete The Specified File" 0 delete_file}
{ 0x07 "RENAME" "Rename File To A New Name" 0 rename_file}
```

Here each entry consists of five items. We see that the entries are listed in numerical order by field value. The second item, as before, is a string for display in the Summary Pane; this contains the formal name of the command. The third item is a more descriptive indication of the command for display in the Decode Pane. If only one string is given as in the X.25 table above, it is used in both the Summary and Decode Panes. The fourth item is a number, which in this case is always zero; SMB does not use this item and zeroes are inserted just as placeholders. This item is called extra table data and is stored as a 64-bit signed integer (which means you can use negative as well as positive numbers for extra table data). The fifth item, called the branch data, is the name of a FIELD or GROUP. The only standard use of this field is as the target of a BRANCH that obtains it using the FromTable method. In principle, however, it could be used in other ways by means of custom methods.

We see that tables can combine lists of strings with lists of processors and even numeric data for use in decoding fields. It happens that there are few protocol decoders that make use of both the extra data and the branch data. The HTTP decoder (HTTP.dec) offers a good example of how extra data can be used. It uses the extra data to distinguish three classes of messages: those that contain URLs, responses and data. Here is the table:

```
TABLE req_resp
{ 0x4f5054494f4e53   "OPTIONS"   "OPTIONS"   1}
{ 0x474554         "GET"       "GET"       1}
{ 0x48454144        "HEAD"      "HEAD"      1}
{ 0x504f5354        "POST"      "POST"      1}
{ 0x505554         "PUT"       "PUT"       1}
{ 0x44454c455445     "DELETE"    "DELETE"    1}
{ 0x434f4e4e454354   "CONNECT"   "CONNECT"   1}
{ 0x485454502f312e30 "HTTP/1.0"  "HTTP/1.0"  2}
{ 0x485454502f312e31 "HTTP/1.1"  "HTTP/1.1"  2}
{ Default          "HTTP data" "HTTP data" 3}
ENDTABLE
```

Note that the value entries are simply hexadecimal encodings of the ASCII representations of the commands that follow. The first few executable statements are:

```
FIELD type (OneToken 8) (Table req_resp) IN_SUMMARY Token 50 Token

GROUP req_msg IF (MatchTableData req_resp type 1)
{
    FIELD req_data (ToTerminatingValue Exclusive 0x0d)
                   (StringOfASCII) IN_SUMMARY URL 120 URL
    FIELD cr_lf (Fixed 2) (Hex) SUPPRESS_DETAIL
    GROUP headers ;
}
```

The initial FIELD extracts the first word of text from the data and looks it up in the table to identify the message type. The GROUP then decodes all those messages that are given a type of 1 in the table. Similar GROUPs then appear for types 2 and 3.

You may find the need to construct a table that includes sequences of identical entries, such as:

```
{ 0x8000 "NATIONAL ADVANCED SYSTEMS"}
{ 0x8001 "NATIONAL ADVANCED SYSTEMS"}
{ 0x8002 "NATIONAL ADVANCED SYSTEMS"}
```

You can group such a range of entries into one, writing it as:

```
{ 0x8000 0x8002 "NATIONAL ADVANCED SYSTEMS"}
```

This is an example from the Internet Sockets table used by the protocol analyzer. By including a second numeric item at the beginning, an inclusive range can be specified. The lower number must be given first.

Some tables are so large that you might prefer to keep them in files by themselves. And if you want to use a particular table in two or more different decoders then, almost certainly, you would prefer to keep only a single copy of it. For such cases, an alternate form of the TABLE statement exists. You will find this example in the TCP and UDP decoders:

```
TABLE ports @"ports.tbl"
```

The @ symbol specifies that the ports table should be read from the named file, which must be stored in the same directory as the DEC file. In such a case the ENDTABLE statement must appear as the last statement in the table file.

Note that if you use the branch data item, the field names you use must be put into any decoder that includes that table. If the branch data is irrelevant to the decoder that uses it, the fields may be defined as dummies like this:

```
FIELD create_new_dir (Fixed 0) (Decimal) SUPPRESS_DETAIL
```

That will define the field so the decoder will load, but the field has no effect.

Eight types of statements appear in the executable section. Five of them, FIELD, RESERVED, GROUP, GROUP FIELD and BRANCH, we refer to collectively as executable statements. The sixth, PARAMETER, is a pseudo-field, the seventh is the INCLUDE keyword and the eighth is END_MAIN_PATH.

Every statement except BRANCH, RESERVED and INCLUDE defines some kind of protocol field or group of fields and every such entity must be given a unique name within a decoder. Thus, not only must all FIELDs have unique names, but you cannot have a FIELD with the same name as a GROUP or a PARAMETER. By convention we use names that are all lower case and with words separated by underscores.

We will examine each statement type in turn.

FIELD

A FIELD statement always does the following:

Names a field
Defines the size of that field
Generates material to display the content of the field in the Decode Pane

Normally it also:

Moves the pointer forward to the next field

Optionally it may:

Reposition the pointer prior to decoding
Inhibit updating of the pointer (leaving it pointing to the same field)
Make decoding of the field conditional
Massage the field value before storing it
Display something in the Summary Pane
Check the content of the field
Tag the field for use in filtering and other operations
Invoke a method that verifies a frame check sequence (CRC or checksum)
Store a value in a variable

Formally, the composition of a FIELD statement is as follows:

```
FIELD field_name
[START_BIT StartBitMethod]
SizeMethod
[<Conditional-Clause>: IF [NOT] BooleanMethod [ELSE_SKIP
        SizeMethod] | PRINT_IF [NOT] VerifyMethod]
[<Retrieve-Clause>: RETRIEVE RetrievalMethod [ALSO
        RetrievalMethod]*]
[<Processing-Clause>: PROCESSING RetrievalMethod [ALSO
        RetrievalMethod]*]
FormatMethod [ALSO FormatMethod]
[IN_SUMMARY|IN_SUMMARY_ONLY|IN_COLLAPSED “summary_tag”
        column_width]
"detail_tag" | SUPPRESS_DETAIL
[<Tag-Clause>: TAG “tag”]
[MARGIN_TEXT FormatMethod]
[INFO]
[NO_MOVE]
[HOST_DATA_ORDER | NETWORK_DATA_ORDER]
[<Verify-Clause>: VERIFY|SUPPRESS_IF_VERIFIED [NOT]
        VerifyMethod [ALSO [NOT] VerifyMethod]]
[<Store-Clause>: STORE variable_name [<Retrieve-Clause>]]
```

Keywords are shown in all capitals. Optional components are shown in square brackets. Elements in angle brackets are compound components that are described below.

The FIELD statement, as defined here, may seem terribly complicated but, as we have seen in the tutorial, most FIELD statements are really rather simple and straightforward because most or all optional components are unneeded.

Here is the composition of a FIELD with all optional parts omitted:

FIELD field_name SizeMethod FormatMethod “detail_tag”

We will now examine each element in turn. We will start with the required components and then list the optional ones in the order in which they appear.

field_name

A unique name is required for every FIELD even if that name is never referenced.

SizeMethod

The second required item is the invocation of a method that determines the size of the field. For a field of fixed size we will have something like:

```
FIELD command_code (Fixed 1 Byte) (Hex) "Command code"
```

The Fixed method takes a first parameter that is a numeric count and a second parameter that expresses the unit. Acceptable units are bits, (4-bit) nibbles, (8-bit) bytes, (16-bit) words, (32-bit) dwords, and (64-bit) qwords. The unit specifier may be written in singular or plural form (e.g. “bit” or “bits”) as appropriate.

The other commonly used method for sizing a field serves when the size is given by the content of another field, as in:

```
FIELD data_length (Fixed 1 Byte)
FIELD data (FromField Bytes data_length –2)
```

Here we suppose that a one-byte field indicates the size of the variable-length field that follows. The FromField method takes three parameters. The first is the unit (as above). The second parameter is the name of the FIELD that contains the length. The third is a number that is added to the result since, in some cases, the length field includes or excludes other things (for example, itself). We call this number an adjustment.

Note that the field containing the length must be processed before the FromField method is called.

A complete catalogue of methods is provided in the Method Reference.

It may help your understanding of the mechanics of the decoder to know that what a Size Method returns is simply a number of bits. That number is used in extracting the field value from the raw frame data, formatting the field value, and incrementing the pointer. This does not mean, however, that you can put a literal number in this item; a method invocation is required.

FormatMethod

The next required item is the FormatMethod. This is the invocation of a method that formats the field value for display in the Frame Display window. The value may be displayed in the Decode Pane, the Summary Pane, the collapsed decode line, or all of these. Common formatting methods include Decimal and Hex which encode integers in the appropriate base, and Table

which looks up the value in a table to find a corresponding string. There are also methods for formatting character data and various other things.

The size of the field (as defined by the SizeMethod) is available to formatting methods so you can depend on appropriate significance in the encoding. Thus with a field to be formatted as a character string, the formatting method knows how many characters it should process; it does not look for a terminator or anything like that.

Note that a FormatMethod must be included even if, by virtue of a subsequent SUPPRESS_DETAIL keyword, nothing is actually displayed.

“detail_tag"

The last required item is either the label to be used for displaying the field value in the Decode Pane or the keyword SUPPRESS_DETAIL, which prevents the field from being displayed.

We now list optional components of the FIELD statement:

START_BIT StartBitMethod

If we need to move the pointer to get it pointing to the field to be decoded then we include a START_BIT clause. We put the START_BIT keyword after the field name plus a call to a suitable method. Here is an example:

```
FIELD start_again START_BIT (FromStart 0 Bytes)
```

This is used to reposition the pointer to the start of the layer in the midst of a decode when we have a need to reprocess certain fields. Note that, as the name here implies, we must give a new name to a field that we reprocess in this way. Several methods are provided for repositioning the pointer. The one used here selects a position relative to the start of the layer expressed, as usual, as a count plus a unit specifier. Similarly the method FromEnd allows us to reposition relative to the end of frame. The Move method moves a fixed number of bits or bytes forward or backward from the current position. And the MoveToField method repositions to a named field.

Conditional-Clause

A conditional clause takes one of two rather different forms:

```
IF [NOT] BooleanMethod [ELSE_SKIP SizeMethod]

PRINT_IF [NOT] VerifyMethod
```

The first style is used to make the entire processing of the field conditional: if the condition is met then the field is decoded and otherwise it is skipped. With the second style the field is always processed and the condition governs only whether it is displayed or not. This distinction leads to an important semantic difference between the two: An IF clause is evaluated before the field value is obtained while a PRINT_IF is evaluated after.

Conditional clauses are optional and, on any FIELD statement, only one type (IF or PRINT_IF) may be used. We will now examine each in deeper detail.

IF [NOT] BooleanMethod [ELSE_SKIP SizeMethod]

The keyword IF introduces the first style of conditional clause. The IF is followed by a call to a Boolean Method with an optional intervening NOT. A Boolean Method is one that makes some test and returns True or False.

Here is a good example:

```
FIELD fmt_length (Fixed 1 Byte) (Decimal) "Format Length"

FIELD fmt (FromField Bytes fmt_length 0) IF NOT (Fieldis EqualTo 0 fmt_length)
(StringOfHex 32) "Format Data"
```

This example shows a way of dealing with a variable-length field where the length is given by another field. If the length is zero then the second field does not actually exist. The conditional clause is:

```
IF NOT (FieldIs EqualTo 0 fmt_length)
```

The FieldIs method makes a numeric comparison of the value in a named field (fmt_length in this case) with a given constant (0 here). Note that the result of the test is complemented by virtue of the NOT; we want to process the field only if the length is *not* zero. Actually, the clause could equally well be written:

```
IF (FieldIs NotEqualTo 0 fmt_length)
```

You may find times that you want to use an IF and skip some amount of data if the test fails. An IF clause can be extended to allow this as in this example taken from the Van Jacobson Compressed headers decoder (VJC.DEC):

```
FIELD Seq (Fixed 1 Bit) IF (TestTableData Special Special) ELSE_SKIP (Fixed 1 Bit) (Decimal)
IN_SUMMARY Seq 40 Seq
```

Here a particular bit is meaningful only when another field has been set in a certain way (as tested by the TestTableData call). To use this option on an IF, place the keyword ELSE_SKIP after the call to the Boolean Method and follow that by a call to a Size Method that determines how much data is to be skipped.

PRINT_IF [NOT] VerifyMethod

This form of conditional clause is triggered by the PRINT_IF keyword. The condition is tested by a Verify Method and the test may be reversed by the presence of a NOT.

A Verify Method is like a Boolean Method in that it returns True or False. A different set of testing operations is provided though, mostly functions to test the value obtained for the field.

Here is an example from the SMB decoder. SMB makes extensive use of bit flags and, to make them stand out, the decoder generally displays such flags only when they are set. (Flags that are not set are assumed to be of no interest.)

```
FIELD case (Fixed 1 Bit) PRINT_IF (FieldIs EqualTo 1) (Constant "Pathnames Are Caseless")
"Bit 3"
```

<Retrieve-Clause>

The optional Retrieve Clause allows you to modify the field value as it is retrieved from the raw data. The data might, for example, need to be masked in some way; the bits might need to be rearranged; or you might want to scale the value for easier handling. In any event, a Retrieve Clause may be used only for fields that are 64 bits in size or smaller.

The format of a Retrieve Clause is:

```
RETRIEVE RetrievalMethod [ALSO RetrievalMethod …]
```

It is introduced by the RETRIEVE keyword. The clause lists at least one Retrieval Method and may include any number separated by the ALSO keyword. Each Retrieval Method performs some transformation on the field value.

Consider this FIELD statement:

```
FIELD year_mfr (Fixed 1 Byte) (RETRIEVE AddInteger 1980) (Decimal) “Year of manufacture”
```

This converts a year expressed as an offset from 1980 into a full 4-digit number. In some cases, such as this one, you may need to consider carefully whether it would be easier to convert the value when retrieving it or in the process of formatting it. If you choose to convert the data when formatting it then you may have to write a custom Format Method for the purpose. Bear in mind that, when you convert data, the altered value will appear only in the Decode and Summary Panes. The original data (the “real” data if you will) will still show up in the Binary, Radix, Character and Event Panes.

There could be times when you want to perform multiple operations on a field value. And, as we said, you can do just that by stringing multiple Retrieval Methods together using the ALSO keyword. The functions are applied in the order (left to right) that you specify. You might, for example, want to construct a field composed of the eighth bit of three consecutive bytes while throwing the rest of the bits away. SplitField is a method that allows you to specify the number of bits on the right to keep, and the number of bits to remove from the middle. The method then shifts the remaining bits to the right. For example:

```
FIELD three_bits (Fixed 3 Bytes) RETRIEVE (SplitField 1 7) ALSO (SplitField 2 7) ALSO
(SplitField 3 7) (Hex) “Three bits”
```

Assume that the data in the three bytes is 000000010000000100000001.
After the first SplitField, the data looks like: 00000001000000011.
After the second SplitField, the data looks like: 0000000111.
After the third SplitField, the data looks like: 111, which is the 8th bit in each byte put together.

Standard Retrieval Methods also include functions for doing arithmetic with values in other fields.

Another use for a Retrieve method is to gain a hook into each field as it goes by. Since the method is called every time we see a field, in both the compiling and the retrieval stages, you can call a custom retrieve method that uses inter-frame data to record some information about what is being decoded. You must be careful, though, because your method is called more than once for the same field. If you want to get a hook into each field only during compilation, use the PROCESSING construct described below.

<Processing-Clause>

The optional Processing Clause allows you to gain control when the frame is first compiled. It is similar to the Retrieve Clause, with the following differences:

1. Processing clauses can be used on fields larger than 64 bits, where Retrieve clauses can only be used on fields 64 bits in size or smaller.

2. The Processing clause is executed only once and is always executed when the frame is first compiled. The Retrieve clause is executed only when the frame is displayed in the protocol analyzer and so is unpredictable both in the number of times it is executed and when it happens.

Processing clauses are most useful when you need to grab a value and save it as inter-frame data. You can string several Processing clauses together using the ALSO keyword, just as you can with Retrieve clauses.

The format of a Processing Clause is:

```
PROCESSING ProcessingMethod [ALSO ProcessingMethod …]
```

There are currently no standard Processing methods.

ALSO FormatMethod

This optional clause allows you to display a field value in two different formats or together with a label of some kind. It must immediately follow the call to the primary Format Method. The result of the second Format Method is displayed after the first one with a space separating them.

The ALSO clause works well when you want to display a field value and add a unit specifier to it, as in:

```
FIELD port_speed (Fixed 2 Bytes) (Decimal) ALSO (Constant “bps”)
```

ALSO VerifyMethod

Same as ALSO FormatMethod except that it allows you to call two Verify Methods for one field.

IN_SUMMARY "summary_tag" column_width

The next optional FIELD component is the IN_SUMMARY clause. This is used when we want the field displayed in the Summary Pane on the Frame Display and in the summary line when a protocol is collapsed in the main window of the Protocol Navigator. As already discussed, we use the summary line to show the most critical data in a layer, data that summarizes what the layer is doing. The keyword IN_SUMMARY is followed by the name of the Summary Pane column that we want the data to show up in and the default width in pixels for this column. In the Protocol Navigator, the column name is used as the label. Here is an example reprised from the FP decoder.

```
FIELD command_code (Fixed 1 Byte) (Table tableCommands)
    IN_SUMMARY "Command/Response" 150 "Command"
```

The user is free to change the width of any column by manually adjusting it on screen. Note that you can put the contents of more than field in the same Summary column. For example, we could put the Command and Response in the same Summary column.

```
FIELD command_code (Fixed 1 Byte) IF (FieldIs EqualTo 1) (Table tableCommands)
IN_SUMMARY "Command/Response" 150 "Command"

FIELD response_code (Fixed 1 Byte) IF (FieldIs EqualTo 0) (Table tableCommands)
IN_SUMMARY "Command/Response" 150 "Response"
```

There are two additional keywords that can be used in place of the IN_SUMMARY keyword. These restrict the summary line display to either just the Summary pane or just the Protocol Navigator.

IN_SUMMARY_ONLY - the field will appear only in the Summary pane on the Frame Display
IN_COLLAPSED - the field will appear only in the Protocol Navigator

NOTE: If you used the SUMMARY_COLUMNS option at the top of your decoder, do not include the column_width field above. Including the column_width in this case will cause a syntax error when the decoder is loaded.

SUPPRESS_DETAIL

This optional keyword appears in place of the detail tag. Used when you don't want to display the contents of the field in the Decode pane.

<Tag-Clause>

The optional Tag Clause is introduced by the keyword TAG followed by a Tag method. Tagging a field is necessary for some new features that are not yet available. For example, it is anticipated that tracking information will be gathered for tagged fields and users will be able to filter on tagged fields or use them as triggers. At the present time, however, tagging a field doesn't do anything useful but does no harm either.

MARGIN_TEXT FormatMethod

When this keyword is present, the output of the field is put into the margin of the Protocol Navigator window, next to the frame number. Be careful using this keyword, as the margin text includes the contents of any field encountered with this keyword in any protocol contained in the frame. The margin is a fixed size and if too many fields try to put information in the margin, some of it may not be displayed.

INFO

The presence of this optional keyword causes the field output to be duplicated in a node called "Information" at the top of the Decode pane on the Frame Display. The Information line is displayed directly under the Errors line (if one is present). Users can filter on the presence of the Information node, search for frames containing the Information node and hide it.

NO_MOVE

The optional NO_MOVE element tells the protocol analyzer not to update the pointer after processing the field. This is useful when you want to process a field twice and when you want to process a field out of sequence. If you have a FIELD with both a START_BIT and a NO_MOVE then the pointer is left where it was before the FIELD was processed, *not* at the start of the field. If you think about it, this is what is needed for a situation where you want to peek at a field ahead of the pointer.

HOST_DATA_ORDER | NETWORK_DATA_ORDER

Include this item if the field should be decoded using a data order other than the established default. You may of course include it anyway to document how the field is encoded.

<Verify-Clause>

The optional Verify Clause allows you to invoke a method to check the value in the field. You can compare it with a constant or with the content of another field, validate it as a CRC or checksum or, with a custom method, check it in any other way you might need. In all cases (with standard methods, at least) the test can be reversed by including a NOT (as in an IF clause). The format of the clause is:

```
VERIFY|SUPPRESS_IF_VERIFIED [NOT] VerifyMethod [ALSO [NOT] VerifyMethod]
```

The clause is usually introduced by the keyword VERIFY followed by a call to a Verify Method with an optional NOT in between. A Verify Method not only performs a check but, if the check fails, also generates some text indicating what the result of the check should be; this message is then displayed in the Decode Pane. In addition, when the check fails, the entire field is displayed in the Errors section of the Decode Pane and in red instead of the usual black.

You may add a second check to a Verify Clause by following the basic part with the keyword ALSO and a second call to a Verify Method with an optional NOT.

An alternative form of the Verify Clause is introduced by SUPPRESS_IF_VERIFIED. When this keyword is used, everything is done as described above except that the field is displayed *only* if the check fails (and then of course in red). This is useful for dealing with fields that “should” always contain a certain fixed value.

An example from the IP decoder shows a check that the protocol version number is at least 4:

```
FIELD version (Fixed 4 Bits) (TABLE ver_nums) "Version" VERIFY (FieldIs
GreaterThanOrEqualTo 4)
```

Another extract from the IP decoder gives an example of using two verification checks:

```
FIELD ihl (Fixed 4 Bits) (Decimal) IN_SUMMARY IHL 30 "Header Length" VERIFY (FieldIs
GreaterThanOrEqualTo 5 ) ALSO (IsNoMoreThanLayerInDwords)
```

Here a length field is checked both for a minimum value (5) and for consistency with the layer length.

And an example taken from the Ethernet decoder illustrates the use of a Verify Method to check a CRC. ("FCS" stands for Frame Check Sequence.)

```
FIELD fcs START_BIT (FromEnd 4) (Fixed 4) (Hex) FCS NO_MOVE VERIFY (Check_CrcCRC32 0
0xFFFFFFFF Swap)
```

<Store-Clause>

The optional Store Clause allows you to copy the current field value into what is essentially a variable. The format is:

```
STORE variable_name [<Retrieve-Clause>]
```

Syntactically a variable name is the same as a field name. In fact, a variable is implemented as a special kind of field, a field that just has a value and a size. (Note that a proper field also has an offset associated with it, along with formatting options.) Like most data-holding entities, a variable holds a 64-bit value which, of course, means that you can only STORE the value for a field that is 64 bits or less.

A variable is defined by its use. In other words, you create a variable by writing its name in a Store Clause – and that is the only way to create one. You use a variable just as you would use a field – by passing its name as a parameter to a suitable method. Here is an example:

```
FIELD string (FromField Bytes my_variable) (StringOfAscii) “String”
```

Here we have a field that contains a character string of variable length. The Size Method FromField indicates that the length of the field, in bytes, is the value of the field called my_variable, which is really a variable. Now you may well ask, why is a variable needed here? Surely you can pick up the length of the string directly from the field that defines it? The answer is best given by an example. Suppose that the string may be very, very long. And that its length may be defined by a one-byte or a two-byte field depending on what it takes, as in:

```
FIELD s1_or_2 (Fixed 1 Bit) (Binary) SUPPRESS_DETAIL

FIELD len_1 (Fixed 1 Byte) IF (FieldIs EqualTo s1_or_2 0) (Decimal) “Length of string” STORE
string_length
```

```
FIELD len_2 (Fixed 1 Word) IF (FieldIs EqualTo s1_or_2 1) (Decimal) “Length of string”
STORE string_length

FIELD string (FromField Bytes string_length) (StringOfAscii) “String”
```

This DecoderScript fragment decodes three successive fields. The first field is a single bit that indicates whether the next field, which contains the length of the character string, is a byte (0) or a word (1). The IF clauses on the next two statements ensure that the next field is decoded appropriately and that the correct length of the string is stored into the variable string_length. The final statement then decodes the string itself.

Now you might possibly wonder why you could not code this sequence as:

```
FIELD s1_or_2 (Fixed 1 Bit) (Binary) SUPPRESS_DETAIL

FIELD string_length (Fixed 1 Byte) IF (FieldIs EqualTo s1_or_2 0) (Decimal) “Length of string”

FIELD string_length (Fixed 2 Bytes) IF (FieldIs EqualTo s1_or_2 1) (Decimal) “Length of
string”

FIELD string (FromField Bytes string_length) (StringOfAscii) “String”
```

This bypasses the variable and would simply decode a field called string_length as a one- or two-byte value as appropriate. If it worked that is. In practice, it fails because the string_length field has two definitions and that is forbidden. The only solution is to use a variable and this is exactly the type of thing that variables are intended for.

The optional Retrieve Clause within the Store Clause allows you to perform some transformations on the field value prior to storing it in the variable. For example, assume you have a 2 bit field. The value of the field indicates the length of the data according to the following table:

Value	Length
0	1 byte
1	2 bytes
2	4 bytes
3	8 bytes

You need to decode the field and somehow get the right value in the something called "length" so you can use it later. You could do a bunch of IF tests and then store each field into a variable called "length" or you could use a nifty method called ReplaceWithExtraData, which replaces the value in the field with the extra data from a table. Here's the table and decoder you could use:

```
TABLE length_table
{ 0 "" "1 byte" 1}
{ 1 "" "2 bytes" 2}
```

```
{ 2 "" "4 bytes" 4}
{ 3 "" "8 bytes" 8}
ENDTABLE

FIELD length_index (Fixed 2 Bits) (Table length_table) "Length" STORE length RETRIEVE
(ReplaceWithExtraData length_table)
```

The lifetime of a variable is from the first encounter with it through the end of the decoding of a layer. This is what programmers call its scope. This means that, if you are careless, you can use a variable before setting it. Should you do so, no error will occur, but the value obtained will be garbage. Note that you cannot use variables to pass data to the decoder for another layer nor to a future instance of your own decoder (that is effectively to the decoding of a subsequent frame). Instead you must use what we call a Decoder Data Object.

FIELD Order of Statement Processing

There may be situations where you need to consider in what order the various components of a FIELD statement are processed. Mostly they are done in the order in which they appear, but there are exceptions. So, here is the order of execution:

1. Any IF condition is tested. If this tests false and an ELSE_SKIP exists, then the ELSE_SKIP is processed. Otherwise the rest of the steps are skipped.
2. Any StartBit method is called.
3. The Size Method is invoked to determine the size of the field.
4. If the field size is 64 bits or less, then the value is extracted.
5. If the field size is 64 bits or less, any Retrieve Clause is processed to massage the value.
6. If the field size is 64 bits or less, any Store Clause is processed.
7. Any PRINT_IF condition is tested and if it proves false then the next 2 steps are skipped.
8. If an IN_SUMMARY is included then the value is formatted for the Summary Pane.
9. The value is formatted for the Decode Pane, unless the keyword SUPPRESS_DETAIL is present.
10. Any Verify Method is called. If the VERIFY is part of a PRINT_IF, the PRINT_IF is then processed.
11. Any NO_MOVE is processed.

One item of interest that becomes clear from this list is that an IF clause cannot test the value or size of the current field.

FIELD field_name;

Finally on the topic of FIELDs, there is one alternative form of the statement that we must discuss. We observed that every field must have a unique name. But suppose you have a field that appears repeatedly in different threads of your decoder.

If the meaning of the field is identical and you want its value displayed uniformly in all cases then you can define a FIELD for it once and then reference that repeatedly using the form:

```
FIELD field_name ;
```

The semi-colon tells the analyzer to look up a FIELD defined elsewhere with the given name and use that. The full definition of the field may be placed in any suitable position in the decoder. The full definition need not even be inline. (That is it can be placed somewhere where control would never reach – that is, after the END_MAIN_PATH statement and not within the target of a BRANCH.)

RESERVED

The RESERVED statement allows you to define a field that is skipped in the decoding process. Nothing is displayed for a reserved field in either the Decode or Summary Pane. The syntax is:

```
RESERVED SizeMethod
```

The SizeMethod is exactly as for a FIELD. Conditionals, VERIFYs and other such things are not allowed on RESERVED statements. If you have a field that you want to value check but not display by default then you could use:

```
FIELD reserved (Fixed 1 Byte) (Hex) "Reserved" SUPPRESS_IF_VERIFIED (MustBe 0)
```

This will process a one-byte reserved field and check that its value is zero. If the value matches zero then, by virtue of the IF clause, the field will be skipped and nothing displayed. If the value is not zero then the field will appear in the Decode Pane in red.

GROUP

The GROUP statement allows you to batch a sequence of fields together into a group for hierarchical display in the Decode Pane. A typical case where this might be used is in displaying a set of bit flags, as in this example from the SMB decoder:

```
GROUP share_access "Share Access"
{
    FIELD reserved_5_bits ;
    FIELD share_delete (Fixed 1 Bit) PRINT_IF (FieldIs EqualTo
        1) (Constant "Delete Access") "Share Access"
    FIELD share_write (Fixed 1 Bit) PRINT_IF (FieldIs EqualTo
        1) (Constant "Write Access") "Share Access"
    FIELD share_read (Fixed 1 Bit) PRINT_IF (FieldIs EqualTo 1)
      (Constant "Read Access") "Share Access"
    FIELD reserved_3_bytes ;
}
```

The composition of a GROUP statement is as follows:

```
GROUP group_name [START_BIT StartMethod] [IF [NOT] BooleanMethod SizeMethod]
[REPEAT COUNT|SIZE|UNTIL Method] [“detail_tag”]
{
    element
    element
    …
}
```

The optional START_BIT and IF clauses work exactly as for a FIELD. The new fields are:

REPEAT COUNT|SIZE [**TRUNCATED_FIELD_OK**]**|UNTIL Method**

The major new element we see on GROUP is the optional REPEAT clause. REPEAT is used with a qualifying keyword, COUNT, SIZE or UNTIL plus an appropriate method.

REPEAT COUNT is used when the group is to be repeated a known number of times. A Size Method is used to indicate the number as in:

```
GROUP seven_things REPEAT COUNT (Fixed 7 Times) “Seven things”
```

Be careful here because the value returned by a sizing method is a number of bits. If you wrote instead:

```
GROUP seven_things REPEAT COUNT (Fixed 7) “Seven things”
```

then the GROUP would be repeated 56 times! (That’s because the default unit for the Fixed method is bytes.) The best thing to do is always use the "Times" for the units in Repeat Count clauses.

If the repeat count is the value of a field then use the FromField method as in:

```
GROUP n_things REPEAT COUNT (FromField Times thing_count 0) “Things”
```

REPEAT SIZE is used when a group is repeated an unknown number of times to process a known amount of data. A Size Method is used to define the span of the repeated group. Suppose, for example, we have a protocol in which a layer can contain an arbitrary number of variable-length commands terminated by a one-byte checksum. Then you would use something like:

```
GROUP master_command REPEAT SIZE (ToEndOfLayer 1)
```

ToEndOfLayer returns the number of bits left up to the end of the layer minus the number of bytes given as the parameter. So in this case it will return the number of bits left excluding a one-byte checksum. The GROUP is then repeated until this number of bits has been processed.

When processing REPEAT SIZE, you can think of the number of bytes inside the repeat loop as a self-contained sub-layer. If you try to access bits past the end of that layer, then an error will be reported in the decode. This is usually a sign that either your decoder is faulty or that the frame that was passed in was damaged. In either case, the error is a useful indication. It is possible to imagine a case, however, where it is not desirable to display an error if the group is truncated. The optional keyword, TRUNCATED_FIELD_OK, suppresses that error.

REPEAT UNTIL is used in combination with a Boolean method. Here is an example from the HTTP decoder:

```
GROUP headers REPEAT UNTIL (CurrentDataMatches 16 0x0d0a)
```

The CurrentDataMatches method checks for a given value at the pointer. This particular group repeats until a 16-bit quantity matches hexadecimal 0D0A (that is a CR+LF in ASCII).

Detail Tag

The detail tag on a GROUP is optional (unlike a FIELD). The choice of including one or not has one important effect as illustrated in the following examples:

```
GROUP str "Structure"
{
    FIELD str_code (Fixed 1 Byte) (Hex) "Structure code"
    FIELD str_size (Fixed 1 Byte) (Decimal) "Structure size"
    FIELD str_data (FromField Bytes structure_size 0)
(StringOfDecimal 10) "Structure data"
}
```

Here the GROUP has a detail tag. This gives rise to an expandable branch in the Decode Pane as shown in this screen-shot fragment:

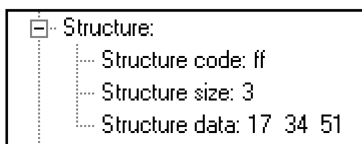


Figure 11 - GROUP with Detail Tag

Remove the “Structure” tag and the expandable branch goes away. The items within the group are displayed as leaves of the current branch as shown here:

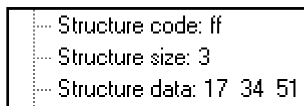


Figure 12 - Group without Detail Tag

The GROUP statement serves one vital purpose that has nothing to do with actually grouping fields. This stems from the terribly useful fact that you can have a GROUP without a detail tag. Then nothing appears for it in the Decode Pane and it serves purely as an IF statement, as in:

```
GROUP early_dialect IF (FieldIs EqualTo 2 word_count)
{
    GROUP andx ;
    FIELD byte_count ;
    FIELD tree_andx_service ;
}
GROUP late_dialect IF (FieldIs EqualTo 3 word_count)
{
    GROUP andx ;
    FIELD optional_support (Fixed 2) (Hex) "Optional Support"
    FIELD byte_count ;
    FIELD tree_andx_service ;
}
```

The GROUP statements check the value of the FIELD word_count. If the word_count is 3, the FIELD optional_support is present. If the word_count is 2, FIELD optional_support is not present. Since the GROUPs have no detail tags, nothing is displayed for them.

There is no ELSE mechanism so you must create the effect by using contrasting GROUPs. To extend the above example, you might add:

```
GROUP other_dialect IF NOT (FieldIsInRange word_count 2 3)
{
    GROUP andx ;
    FIELD byte_count ;
}
```

GROUP FIELD

The GROUP FIELD statement allows you to combine some of the best features of a GROUP with those of a FIELD. This allows you to display an expandable group in the Decode Pane with a value on the entry.

A GROUP FIELD statement is composed of a FIELD statement with the keyword GROUP on the front. Any element that is valid for a FIELD statement is valid for a GROUP FIELD.

```
GROUP FIELD field_name [remaining elements follow those for a FIELD]
{
    element
    element
    …
}
```

Note that groups do not have sizes. The SizeMethod item defines the size of the field, not the group as a whole.

A typical use is to display a set of bit flags, as in this example:

```
GROUP FIELD status_byte (Fixed 3 Bits) (Binary) "Status" NO_MOVE
{
    FIELD active_flag (Fixed 1 bit) (Binary) "Active flag"
    FIELD overflow_flag (Fixed 1 bit) (Binary) "Overflow flag"
    FIELD error_flag (Fixed 1 bit) (Binary) "Error flag"
    }
```

Note the NO_MOVE flag on the GROUP FIELD. The resulting display in the Decode Pane resembles this partial screen-shot:

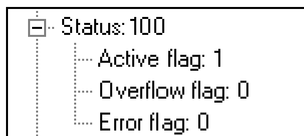


Figure 13 - Bit Flags with GROUP FIELD

The combined value of the flags is always revealed but they are not shown individually (which takes a good bit of vertical screen space) unless the branch is expanded. This is especially a good means of handling such a field when all the flags are zero except in a few rare cases.

Here is a contrasting example:

```
GROUP FIELD response (Fixed 1 Byte) (Table Responses) "Response"
{
    FIELD data_count (Fixed 1 Byte) (Decimal) "Count of data bytes"
        VERIFY (MustBe 1)
    FIELD data (Fixed 1 Byte) (Decimal) "Data"
}
```

This is used to display a three-byte sequence consisting of a response code, a byte count (which must have the value one), and a response-specific data byte. The resultant display is similar to:

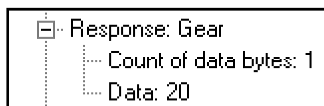


Figure 14 - GROUP FIELD from Table

Imagine a protocol message that carries a sequence of such responses. As it first opens, the Decode Pane shows a list of responses (e.g. “Gear” in our case) and any one of them can be expanded to see the details.

BRANCH

The BRANCH statement allows transfer of control to a given FIELD, GROUP or GROUP FIELD. The format is either:

```
BRANCH (BranchMethod)
```

where BranchMethod is a method that resolves to the name of a field or group,
or

```
BRANCH IF (BooleanMethod) yes_field no_field
```

where control goes to the yes_field if the condition is true and the no_field if the condition is not true. The yes_field and no_field items may be field names or group names.

The name "branch" may be slightly misleading. The effect is more like that of a GOSUB in BASIC or an "execute" command in certain languages. The statement that is the target of the branch is performed and then control returns to the statement following the BRANCH. If the target is a FIELD then that single statement is performed. If the target is a GROUP or GROUP FIELD then that statement along with any enclosed within its brace block are performed.

BRANCH is used in combination with a Branch Method. As an example, consider a protocol that carries a large set of commands, each of which needs its own set of decoder statements. Then we can put the names of the GROUPs that process the commands in a table and call them as in this example:

```
BRANCH (FromTable command_table command_code)
```

In this case, the value in the field named command_code is looked up from the table named command_table and uses the branch item in the matching table entry as the target for the BRANCH. If an entry without a branch item is referenced in this way then the BRANCH becomes a no-operation – no branching is done and processing continues with the next statement in line.

BRANCH IF is used in conjunction with a Boolean Method to determine which of two field/group statements to decode. For example, assume that the contents of a field defined whether the bytes that follow are a command or a data stream. If the field contains a zero, a data stream follows, and if the field contains a number greater than zero, the number is the number of the command. Our branch statement would look like this:

```
DECODE

FIELD data_or_command (Fixed 1) (Decimal) SUPPRESS_DETAIL

BRANCH IF (FieldIs EqualTo 0 data_or_command) data command

END_MAIN_PATH
```

```
FIELD data (ToEndOfLayer 0) (StringOfHex 24) "Data"

GROUP command
{
	[command fields]
}
```

In this situation, if the data_or_command field contains a zero, the data field will be decoded, otherwise the command group will be decoded.

PARAMETER

This defines a runtime parameter for the decoder. You can think of it as a field that the user can directly change on the fly. This is used when the stream of data (or the capture file we are decoding) does not have enough information to allow the decoder to figure out everything. An example is a protocol that can have 8 bit fields on one installation and 16 bit fields on another. We could write two identical decoders except for the size of the bit fields, but that is unwieldy and hard to maintain. Instead, we can let the user tell us which installation he/she has.

Here is the PARAMETER statement:

```
PARAMETER field_name user_prompt [ verify_clause | list_clause ] default_value [ALWAYS_ASK
| NEVER_ASK | ASK_IF_NOT_DEFINED]
```

field_name
This is exactly like the field name in the FIELD statement. In fact, all the PARAMETER statement does is define a field, but it is a field that is not dependant on the data in the frame.

user_prompt
Since the value of the parameter is set by the user, the user must be prompted for the value. This string is the text of that prompt.

There are two types of parameters: lists and numbers. The following two optional clauses define which type of parameter this is. If neither is present, then the parameter is a number. When the user is prompted for a list-type parameter, a combo box is presented to the user, so that only the values requested can be entered. When the user is prompted for a number-type parameter, an edit box is presented and the user must type in a number.

verify_clause is:

```
	VERIFY VerifyMethod
```

This makes sure that the 64-bit integer that the user enters meets the desired criteria.

list_clause is:

```
	LIST { item … }
```

This defines a list of legal values that the parameter can take. The user is prompted with this list.

default_value
This is the value that the parameter should take if the decoder is not told differently by the user. It can also be thought of as the “starting value”. If a list was defined, this item is one of the members of that list, otherwise it is a number.

ALWAYS_ASK
If this is present, then every time the decoder is loaded, the user is asked to enter the parameter. This is useful if the parameter is always changing and you need to make sure the user is always aware of it.

NEVER_ASK
If this is present, then the user is never asked for the parameter value. The user can always go to the parameter menu and change it, but doesn’t have to. This is used when the parameter has a good default value that only needs to be changed rarely. Most users will never notice it.

ASK_IF_NOT_DEFINED
If this is present, then the user is asked for the parameter the first time the decoder is loaded, then never asked again. The user can always find the parameter in the menu and change it. This is used when the parameter is important to a particular environment, and the testing unit isn’t going to leave that environment.
Note that there are two ways to do the following common case:

What if you needed a parameter that was the size in bytes of a particular field and the only legal values are 1,2, or 3? You could use either of the following statements:

PARAMETER field_size “Size of field (in bytes)” VERIFY (FieldIsBetween 1 3) 1
PARAMETER field_size “Size of field (in bytes)” LIST { 1 2 3 } 1

Which is better? It is up to you. The first will present the user with an edit box and the user can type in any value. If the value is not between 1 and 3, then an error message pops up and the user can choose another value. The second will present the user with a combo box with only the items 1, 2, and 3 in it, so the user can’t possibly pick an illegal value. In this case, we think that the second way is better, but what if the user could pick a number between 1 and 30? The combo box would seem clunky to the user.

Parameter values are kept in a file in the same directory as the .DEC file, with the same name and the extension .INI. If this file is deleted, all the parameters revert to their defaults. Capture files store the parameters used to capture the data in the file, and will always load those parameters rather than the ones in the INI file.

INCLUDE

The keyword INCLUDE allows you to put some of the code in a different file. After the INCLUDE keyword should be a filename to be inserted at that point in the decoder. The file can contain tables and/or fields. The file must be in the same directory as the DEC file. By convention we use the extension .DH, but any extension except for DEC or DEX is ok.

For example:

INCLUDE commandtable.dh

Files should not be included in more than one place in the decoder, as this will result in duplicate table or field definitions in most cases.

END_MAIN_PATH

The mandatory END_MAIN_PATH statement marks the end of the main sequence of executable statements (roughly the equivalent of the "main program" in many programming languages). For each layer, executable statements will be processed starting at the first that appears and stopping when END_MAIN_PATH is reached. Any statements that appear after the END_MAIN_PATH will be processed only if referenced by a BRANCH or by a FIELD statement of the form "FIELD <name>;". By the same token, any statements that play the role of subprograms should be placed after the END_MAIN_PATH.

Intra-frame and Inter-frame Data

Intra-frame data is used when you need to use or display a value from a lower layer in a higher layer decoder. Inter-frame data is used when you need to use a value from a prior frame in the current frame. A major difference is that the intra-frame data is saved for the duration of the frame; inter-frame data is saved until it's changed or we run out of frames. By contrast, the STORE keyword discussed earlier stores data only for the duration of the layer. We'll start by talking about intra-frame data.

Intra-frame Data

Intra-frame data are actually special RETRIEVE methods that take one parameter: the name the data should be referenced by, which we'll call the "intra-frame data name". This isn't a parameter in the traditional sense of an input to the method, it's more like a variable name. Each field you save must have a unique name. This name is used to tag the data to be saved, and is referenced by both the method that saves the data and the method that retrieves it.

A good example of intra-frame data use is in the IP and HTTP decoders. In the HTTP decoder, we wanted to display the source and destination IP addresses in the Summary pane because we felt it would be helpful to see that information along with the HTTP information. To do this, we need to save the values of the source and destination IP addresses when they are decoded in the IP layer, and then retrieve them for display in the HTTP layer. Intra-frame data is the right choice for this type of situation because we have no need of the IP addresses once the frame has been fully decoded.

Storing Intra-frame Data

The first step is to save the source and destination address in the IP decoder (IP.dec). The method used to save intraframe data is StoreIntraframeField, and it is used in a RETRIEVE clause on the field to be saved. The method name is followed by the intra-frame data name.

Here is a code fragment from the IP decoder, showing how the method is used to save the field values.

```
FIELD src_addr (Fixed 4 Byte)
    RETRIEVE (StoreIntraframeField source_ip_address) (IPAddress) IN_SUMMARY Source
    90 "Source Address"

FIELD dest_addr (Fixed 4 Byte)
    RETRIEVE (StoreIntraframeField destination_ip_address) (IPAddress) IN_SUMMARY
    Destination 90 "Destination Address"
```

Retrieving Intra-frame Data

Now that the addresses have been saved in the IP layer, we need to get it back so we can display it in the HTTP layer. The method used to retrieve intra-frame data is simply called IntraframeField, and it too references the name of the data to be retrieved.

Here is the code from the HTTP decoder (HTTP.dec) showing how the source and destination IP addresses are retrieved and formatted for display in the Summary pane.

```
/* Display Source and Destination IP Addresses, but only in Summary pane. */

FIELD source_ip (Fixed 0)
RETRIEVE (IntraframeField source_ip_address) (IpAddress) IN_SUMMARY "Source IP" 100
SUPPRESS_DETAIL

FIELD dest_ip (Fixed 0)
RETRIEVE (IntraframeField destination_ip_address) (IpAddress) IN_SUMMARY "Dest IP" 100
SUPPRESS_DETAIL
```

Notice that the field size is (Fixed 0). When retrieving data, we shouldn’t specify a field size because the retrieved data isn't present anywhere in the current layer. Retrieved data can be used for other purposes besides display in the Summary pane. Once retrieved, you can use the field value as you would any other field value in the decoder.

Inter-frame Data

Inter-frame data is used when you need to save a value between frames. The value is saved in the FRM file. This file has the same name as the capture file with a FRM extension, and it is located in the same directory as the capture file. The FRM contains all frame related information, including any context sensitive information and inter-frame data.

An example of a protocol type that would require inter-frame data is one where a master sends out a request and gets a response from a slave, but the response doesn't reference which request it's responding to. In order to decode the response correctly, you need to know the request code from the previous frame. (This example doesn't reference a real decoder, but the basic problem is not uncommon.) To make our example complete, assume that the master has an address of 0x00 and slaves have addresses ranging from 0x01 to 0xFF. Our decoder will check the address byte and decode it as a request if it comes from the master and as a response if it comes from a slave.

Storing Inter-frame Data

The first step is tagging the field we want to save. Just as with intra-frame data, inter-frame data uses a special RETRIEVE method to identify the field to be saved, along with an inter-frame data name. The StorePersistentField method stores the value in the field to the FRM file. Here's our sample request field:

```
FIELD request (Fixed 1) RETRIEVE (StorePersistentField request) (Table request_codes)
"Request"
```

Retrieving Inter-frame Data

Now on the next frame, we need to retrieve the value of the request field so we can use it in decoding the response. Use the PersistentField method to retrieve inter-frame data.

```
FIELD request_code (Fixed 0) RETRIEVE (PersistentField request) SUPPRESS_DETAIL
```

Here's the full decoder, using our request and request_code fields. Assume the first byte in the frame is the address. If the data is a request, the second byte is the request code, followed by any additional data that may be required. If the data is a response, the bytes after the first contain the data requested. There's a 2 byte CRC on the end of each frame.

```
DECODE

TABLE request_codes
{ 0 "Send" "Send File" 0 send_file}
{ 1 "Rcv" "Receive File" 0 receive_file}
{ 2 "Ack" "Acknowledge Receipt of File" 0 ack}
{ DEFAULT "Unknown"}
ENDTABLE

FIELD address (Fixed 1) (Hex) "Address"

GROUP frame_is_request IF (FieldIs EqualTo 0x00 address)
{
        FIELD request (Fixed 1) RETRIEVE (StorePersistentField request) (Table
        request_codes) "Request"

    BRANCH (FromTable request_codes request)
}
```

```
GROUP frame_is_response IF (FieldIs NotEqualTo 0x00 address)
{
        FIELD retrieved_request_code (Fixed 0) RETRIEVE (PersistentField request)
        (Table request_codes) "Request Responding To"

    BRANCH (FromTable request_codes retrieved_request_code)
}

FIELD crc16 (Fixed 2) (Hex) "CRC" VERIFY (Check_CrcCRC16 0 0xffff)

END_MAIN_PATH

GROUP send_file
{
        GROUP send_request IF (FieldIs EqualTo 0x00 address) "Send File Request"
    {
            [FIELDs here]
    }
        GROUP send_response IF (FieldIs NotEqualTo 0x00 address) "Send File
        Response"
    {
            [FIELDs here]
    }
}
```

If you need to do something more complex with your inter-frame data, such as matching values from more than one field before deciding what to return, you will need to write a custom data object. These objects and how to write them are described in Chapter 10: Decoder Data Objects.

Chapter 6: Methods

In this chapter we present an introduction to different types of methods that are available. A detailed reference guide to the methods is provided in the Appendices; please refer to that when you write a decoder. The Method Reference Appendix gives a listing of every method, its parameters and a description, often with examples, of how to use the method and what it does. All the methods referenced in this section are located in the DecoderScript Sample Methods.mth file in the My Methods\Source directory of the installed product. Many of these methods are actual methods used by the protocol analyzer, so we have altered the method names by inserting underscores before and after the names to ensure that if you compile them, they will not collide with existing methods.

In what follows we discuss each method type and list the contexts in which methods of the type may be used.

Boolean Methods

Boolean methods return a value of True or False. They are supplied for use in IF (and PRINT_IF) clauses in FIELD and GROUP statements, and in REPEAT UNTIL clauses in GROUPs.

Branch Methods

Branch Methods are used in BRANCH statements. The return value of a Branch Method is (effectively) the name of a FIELD, GROUP, or GROUP FIELD to which a branch is to be made. In all cases the resultant statement is executed and then control is returned to the statement following the BRANCH.

There are two main types of Branch methods. The first branches from a table, and the second branches based on whether the data is from the DTE or DCE side. This latter method is used in serial connections, mostly for master-slave type circuits where there is nothing in the data to distinguish between a command sent by the master and a response sent by the slave. In this case, the monitoring protocol analyzer must be placed such that the data from the master and slave appear on the expected sides of the circuit.

Format Methods

Format Methods are used to encode field values for display in the Frame Display window. The field value is implicitly passed to each method, as is the field size. The value that the method returns is a string.

Most of the methods that format single numeric values work with quantities of up to 64 bits. This applies to Binary, Decimal, Hex, IP Address, Octal and Time. If you try to use, say, Decimal to display the value of a field that is larger than 64 bits an error will be generated. Should you find a need to display a value that is larger than 64 bits as, say, a decimal then you would need to write a custom method for the purpose.

The StringOfXYZ methods can be used to display data from fields that are potentially larger than 64 bits. The StringOf methods may have unexpected results when used in conjunction with a RETRIEVE method, since they look directly at the data in the layer and not at the altered contents of the field. This is especially true when used with intra-frame methods, since the StringOf method will not look at the retrieved data value but will look at whatever happens to be below the bit pointer in the layer.

You can string two Format methods together using the ALSO keyword. For example, if you had a Field that contained the temperature and you wanted to make sure it was clear that the temperature was in Fahrenheit, you could write:

```
FIELD temperature (Fixed 2 Bytes) (Decimal) ALSO (Constant "Fahrenheit") "Temperature"
```

You can string together two Format methods but not more than that.

NextProtocol Methods

These methods are used when protocols are layered so that, when decoding one layer, the protocol at the next higher layer can be identified and its data isolated. There are actually three categories of NextProtocol methods:

NextProtocol	methods that identify the protocol at the next layer
NextProtocolOffset	methods that determine the offset of the next layer
NextProtocolSize	methods that determine the size of the next layer

We will now survey each category in turn.

NextProtocol

A NextProtocol Method returns a value that determines the protocol at the next layer. This return value is not the ID of the protocol in the decoder, but a number that is looked up in the appropriate personality file in order to obtain the decoder ID.

NextProtocolOffset

A NextProtocolOffset Method determines the number of bytes between the end of the current layer and the start of the next. In all but a very few cases the next layer immediately proceeds the current one. In other words, the offset is almost always zero – that is the default and no NextProtocolOffset Method need be used.

There are no standard NextProtocolOffset Methods.

NextProtocolSize

A NextProtocolSize Method returns the size of the next layer in bytes. Such a method is required only in circumstances when the next layer occupies less than the rest of the total frame. One common case is when the frame is terminated by a CRC.

Retrieval Methods

Retrieval Methods are used in RETRIEVE clauses in FIELD statements when the field value and/or size is to be modified. They can also be used as subordinates in STORE clauses. Retrieval Methods are applicable only for fields of size 64 bits or less.

You can string two or more RETRIEVE clauses together using the ALSO keyword. You should not repeat the keyword RETRIEVE for subsequent method invocations. For example, assume you have a field that contains a signed decimal, which is a temperature in Fahrenheit. You are converting a decoder for use in your international offices and want to display the temperature in Celsius. First you need to make the value into a signed decimal and then perform the calculations. The result would look like:

> FIELD temperature (Fixed 2 Bytes) RETRIEVE (SignExtension 16 Bits) ALSO (SubtractInteger 32) ALSO (MultiplyInteger 5) ALSO (DivideInteger 9) (Decimal) "Temperature"

The number of RETRIEVE clauses you can string together with ALSOs is unlimited.

Size Methods

Size Methods are used to define the size of fields and in REPEAT COUNT and REPEAT SIZE clauses on GROUPs. Every FIELD and GROUP FIELD definition must include an invocation of a Size Method. The most commonly used are Fixed and FromField; in many decoders you will not need any other. Note that any size method returns a count of bits.

StartBit Methods

START_BIT clauses invoke methods that reposition the pointer prior to field processing.

Tag Methods

Tag Methods are used in TAG clauses, and typically return whatever is passed in to them.

Verify Methods

These methods are used in VERIFY clauses. In practice, Verify Methods are used for two purposes: verifying checksums and CRCs, and validating the values in fields. We will examine Verify Methods broken down into these two groups. In principle, Verify Methods could also be applied to other kinds of internal consistency checks, for example verifying that two length fields are consistent.

A verify function returns True or False; when False is returned then the field is displayed in red in the decode and text explaining the error is appended to the method's output string.

As with Retrieve methods, you can string together multiple Verify methods using the keyword ALSO.

Checksum/CRC Methods

These methods calculate a certain function over all the bytes in the layer and check the result. They do not take any parameters and make no explicit use of the current field value (with one exception, Checksum1sComplement16bit). Because of this it would be possible to apply one of these methods on any FIELD at all; nevertheless we put them only on CRC and checksum FIELDs because that is where we want a checksum/CRC failure to be signaled.

It is worth repeating that all the standard methods (with the exception of Checksum1sComplement16bit) perform their check over the entirety of the layer assuming that the last byte/word/dword (as appropriate) contains the sender's check. Should you need to verify a CRC over some sub-portion of a layer then you will need to supply a custom method for the purpose.

It is never valid to include a NOT in a VERIFY clause when using a checksum/CRC method. (Including a NOT will not generate an error but the NOT will be ignored.)

Field-Validation Methods

These methods are used to validate the value in a field. All these methods may be preceded in the VERIFY clause by a NOT.

A Sample Method

Methods are written as C++ functions within a framing structure of proprietary statements. Here is an example of a Format Method that formats a certain kind of date:

```
FORMAT
METHOD __MyFormatDate__
    /*
    ** MyFormatDate is called for a three-byte field
          ** comprising, byte by byte:
    **        year (0 - 255, relative to 1900)
    **        month (1 - 12)
    **        day (1 - 31)
    */
CODE
{
    static const char *keyMonth[13] =
          {"???", "Jan", "Feb", "Mar", "Apr", "May", "Jun",
           "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

    int iYear, iMonth, iDay;
```

```
    iYear = ((int)i64CurrentField & 0xFF) + 1900;

    iMonth = ((int)i64CurrentField >> 8) & 0xFF;
    if (iMonth > 12) iMonth = 0;

    iDay = (int)i64CurrentField >> 16;

    if (eParam1 == eDash)
        csOutput.Format("%d-%s-%d", iDay, keyMonth[iMonth],
                  iYear);
    else
        csOutput.Format("%d/%d/%d", iMonth, iDay, iYear);
}
ENDCODE
PARAM "Format Preference" list {Dash Slash} = Dash
```

The variable i64CurrentField is a 64-bit integer that holds the value of the current field, that is the field decoded by the FIELD statement from which this method is invoked. i64CurrentField is one of several values that can be used by any Format Method. Others include the size of the field and a boolean indicating whether the result of the formatting is destined for the Summary Pane or the Decode Pane. The result of the format goes into an output variable called csOutput, an object of the type MFC CString.

Our MyFormatDate method has one parameter that determines whether the date is formatted as, for example, “30-May-2001” or “5/30/2001”. The parameter is declared in the very last line, not at the start as you might expect:

```
PARAM “Format Preference” list {Dash Slash} = Dash
```

This says there is a parameter of type *list* with a default value of “Dash”. Note that the parameter is not given a formal parameter name; rather, it is known by the fixed name eParam1.

The Method Context

The context of a method is defined by:

- Its parameters
- The input values available to it
- The output variables that it sets
- Intra-frame and/or inter-frame data
- Other resources available to it (for example, functions for accessing Tables)

The input values (like i64CurrentField), output variables (like csOutput), and other resources are defined according to the method type. You are free to choose parameters for your methods as you please.

Notation for Constant/Variable Names

We use a form of Hungarian notation for names of input constants and output variables. In the case of i64CurrentField, for example, the “i64” preface tells you that the value is a 64-bit signed integer. Every scalar name starts with a one-, two-, or three-character tag that defines its type (in the C++ sense). These tags are:

b	Boolean
byt	Unsigned Byte (8-bit)
cs	String object
i	Signed Integer (32-bit)
i64	Signed Integer (64-bit)

An array name has an initial ‘a’ for array followed by one of the tags above. Thus abytLayer is an array of bytes. All arrays used are one-dimensional.

We add the following additional tags as appropriate:

bit	Bits
byt	Bytes
frm	Frame
n	Number of
ofs	Offset (which may be in either bits or bytes)
siz	Size (always in bits)

Thus aisizbitField is an array of integers that hold sizes (of fields) in bits, inbytTruncated is an integer that holds a count of bytes (that have been truncated), and ifrmofsbitLayerStart is an integer that holds a bit offset (of the start of a layer) within a frame.

CString and CArray Objects

Methods make use of objects from two Microsoft Foundation Classes (MFC), CString and CArray. For information about such objects please refer to Microsoft’s documentation. In addition, some types based on CArray are defined. These are:

```
typedef CArray<CString,CString&> AString;
typedef CArray<int,int> Aint;
typedef CArray<UINT,UINT> Auint;
typedef CArray<__int64,__int64&> Aint64;
typedef CArray<UCHAR,UCHAR> Abyte;
typedef CArray<unsigned short,unsigned short> Aushort;
typedef CArray<void*,void*> Apvoid;
typedef CArray<bool,bool> Abool;
```

Method Inputs

Here we present a complete catalogue of the input data available to methods. We start with a list of values, given in alphabetical order, and then have a table indicating which of the values each type of method can use.

Most of the descriptions of these values make sense only within the domains of methods that need them. Many, for example, refer to the "current field" which is relevant only to methods called from executable DecoderScript statements.

Some of these values are bit or byte offsets of various things (such as the current field). Except where otherwise stated, such offsets are always offsets within the current protocol layer.

Note that most of these items are declared "const". Methods are prohibited from having side effects, including writing to files; in particular they are not permitted to change anything except designated output variables. As a result, in the context of a method, most of these values are readable but not writable. The only exceptions are i64FieldValue and iFieldSize, which serve as both inputs and outputs to Retrieval Methods.

#define **abFieldHasData**(field) is a macro that indicates whether or not the data for the given field is present in the ai64Field array. It is implemented as a macro for convenience but may be treated as if it were defined as a Boolean array (as implied by its Hungarian prefix).

const UCHAR * const **abytLayer** is a byte array that holds the undecoded data for the current layer. Thus abytLayer[0] is the first byte of the layer and so on.

const Aint64& **ai64Field** is an array of field values for the current layer. This allows you to read the value for a given field if you know that field's index and provided, of course, that that field has already been decoded. For a field with index i, if abFieldHasData[i] holds true then ai64Field[i] holds the value for the field; otherwise the value is wider than 64 bits and must be extracted directly from abytLayer. Note that the only reliable way to get a field's index is to pass the field in as a parameter.

const Aint& **aisizbitField** is an array that holds the size of each decoded field in the current layer. As always this is the size in bits. It is indexed in the same way as ai64Field.

const bool **bCurrentFieldHasData** holds true if the value of the current field is contained in the i64CurrentField and false otherwise. As noted elsewhere, the value of a field is extracted into i64CurrentField only when its size is 64 bits or less; if it is wider than 64 bits then any method that needs the value must retrieve it from abytLayer. bCurrentFieldHasData is the same as (isizbitCurrentField <=64), but it's better to use bCurrentFieldHasData because the definition of whether the current field has data may change in future versions.

const bool **bHostDataOrder** contains true if the bytes of the current field are designated as being in host data-order and false if they are in network data-order. Note that this value is relevant only to multi-byte fields in the abytLayer array; any value in i64CurrentField and ai64Field has already been normalized.

const bool **bIsSummary** is available only to Format Methods. It holds true if the method is being called to produce output for the Summary Pane and false for the Decode Pane.

const bool **bNot** is available only to Verify Methods. It holds true if and only if there is a “NOT” in the verify clause. (You may wonder why bNot does not apply to Boolean Methods as well since any call to a Boolean Method can be prefixed by a NOT. The answer is that a NOT in the Boolean context is handled behind the scenes; in other words, the protocol analyzer internally complements the output of a Boolean Method when that is appropriate. A Verify Method, by contrast, needs to know if a NOT is present because, when the test fails, it is supposed to generate a string for display in the Decode Pane noting what the result should have been.)

const __int64 **i64CurrentField** holds the value retrieved for the current field if that value is 64 bits or less in size. The boolean bCurrentFieldHasData can be used to check whether i64CurrentField contains the field value or not. If it does not then any method that needs the value must extract it from the abytLayer array.

__int64 **i64FieldValue** holds the value that is propagated through a series of Retrieval Methods. It is both an input and an output. The iFieldSize value indicates its size.

const int **iDrf** contains error indications.

int **iFieldSize** holds the size of i64FieldValue as it is propagated through a series of Retrieval Methods. It is both an input and an output.

const int **ifrmofsbitLayerEnd** contains the offset *within the frame* of the last bit of the current layer. This offset does not take account of any truncation (see section on Truncated Frames below).

const int **ifrmofsbitLayerStart** holds the bit offset of the current layer *within the frame*. Since every layer is constrained to starting on a byte boundary, the value in ifrmofsbitLayerStart always equals (ifrmofsbytLayerStart * 8).

const int **ifrmofsbytLayerStart** contains the byte offset of the current layer within the frame. In other words, it counts the number of bytes from the start of the frame to the start of the current layer.

const int **iofsbitCurrentPosition** is the bit offset of the working position in the layer. This offset corresponds to “the pointer” that we described in the chapter on DecoderScript. (We do not call it a pointer here because it is not a pointer in the sense of C but it does effectively point to the current bit.) This value differs from iofsbitCurrentField in certain subtle but important ways. At times their values are the same but at the end of the processing of a FIELD statement, for example, iofsbitCurrentPosition is updated according to the size of the field and the presence or absence of a NO_MOVE.

This value is used mostly in StartBit Methods, the output of which serve directly to update iofsbitCurrentPosition.

const int **iofsbitCurrentField** holds the bit offset of the start of the current field.

#define **iofsbitFieldStart()**provides the bit offset of each decoded field. You might expect this to be an integer array but it is actually implemented as a macro for internal convenience. You can use iofsbitFieldStart much like an array with a field index; just remember to use round instead of square brackets.

const int **iofsbytCurrentPosition** holds the offset of the byte corresponding to iofsbitCurrentPosition.

const int **inbytAvailableLayer** contains the number of data bytes that have been stored for the current layer. For a more thorough explanation, see the section below on Truncated Frames.

const int **inbytFullLayer** holds the original full length, in bytes, of the current protocol layer. This may differ from the available length as explained below in the section on Truncated Frames.

const int **inbytLeftInAvailableLayer** holds the number of bytes stored for the current layer from the current position. If the current field starts on a byte boundary then the count includes that byte; otherwise it starts at the next byte. inbytLeftInAvailableLayer may differ from inbytLeftInFullLayer for reasons explained in Truncated Frames below.

const int **inbytLeftInFullLayer** holds the number of bytes in the current layer from the current position. If the current field starts on a byte boundary then the count includes that byte; otherwise it starts at the next byte. For reasons explained in Truncated Frames below inbytLeftInFullLayer may differ from inbytLeftInAvailableLayer.

const int **inbytTruncated** holds the number of bytes truncated in the current layer. Most often this is zero but see the discussion under Truncated Frames below.

const int **isizbitCurrentField** contains the size of the current field, that is the value returned by the Size Method for the field. As ever, this is the size in bits.

const int **iStream** holds the index of the current data stream. The first stream is numbered 0, the second 1, and so on. As of this writing, Ethernet which has a single stream (0), and serial communication links that have two streams (0 and 1 representing DTE/DCE, send/receive, etc.) are supported. The protocol analyzer has the potential to support multiple streams.

const __int64 **evnFrameNumber** holds the frame number of the frame currently being decoded.

Name	Boolean	Branch	Bytestream framer	Format	Frame Transformer	Next Protocol	Next Prot Offset	Next Prot Size	Post Processing	PreProcessing	Retrieval	Processing	Size	StartBit	Tag	Verify	Comment
abFieldHasData	•	•	•	•	•	•	•	•	•		•	•	•	•		•	Does the ai64Field value have data?
abytLayer	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	The entire layer.
aisizbitField	•	•	•	•	•	•	•	•	•		•	•	•	•		•	Array with the size of each field.
ai64Field	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	Array of all the field values.
bCurrentFieldHasData				•							•	•				•	Is i64Current Field valid?
bHostDataOrder	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	Is this field in host data order?
bIsSummary				•													Was this called for a summary format?
bNot																•	Was NOT present in the decoder when defining this method?
i64CurrentField				•							•	•				•	The value of the current field, if it is < 64 bits wide.

Name	Boolean	Branch	Bytestream framer	Format	Frame Transformer	Next Protocol	Next Prot Offset	Next Prot Size	Post Processing	PreProcessing	Retrieval	Processing	Size	StartBit	Tag	Verify	Comment
i64FieldValue											•	•					The new value for a field in a RETRIEVE method.
iDrf	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	Error indications.
iFieldSize											•	•					The new size of a field in a RETRIEVE method.
ifrmofsbitLayerEnd	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	The ending bit of the layer.
ifrmofsbitLayerStart	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	The starting bit of the layer.
ifrmofsbytLayerStart	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	The starting byte of the layer.
inbytAvailableLayer	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	The number of bytes in the layer that are actually present in the frame.

Name	Boolean	Branch	Bytestream framer	Format	Frame Transformer	Next Protocol	Next Prot Offset	Next Prot Size	Post Processing	PreProcessing	Retrieval	Processing	Size	StartBit	Tag	Verify	Comment
inbytFullLayer	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	The number of bytes in the entire layer, including those bytes that were truncated.
inbytLeftInAvailableLayer	•	•	•		•	•	•	•	•				•	•			The number of bytes left in the array from the current position.
inbytLeftInFullLayer	•	•	•		•	•	•	•	•				•	•			The total number of bytes left in the layer, including truncated bytes.
inbytTruncated	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	The number of bytes truncated.
iofsbitCurrentField	•			•							•	•				•	The bit offset of the current field after the SIZE method is called.

Name	Boolean	Branch	Bytestream framer	Format	Frame Transformer	Next Protocol	Next Prot Offset	Next Prot Size	Post Processing	PreProcessing	Retrieval	Processing	Size	StartBit	Tag	Verify	Comment
iofsbitCurrentPosition	•	•	•		•	•	•	•	•				•	•			The bit offset of the position pointer.
iofsbitFieldStart	•	•	•	•	•	•	•	•	•		•	•	•	•		•	Array or the bit offsets of all the fields.
iofsbytCurrentPosition	•	•	•		•	•	•	•	•				•	•			The byte offset of the position pointer.
isizbitCurrentField				•							•	•				•	The size of the current field.
iStream	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	The stream that the frame was received on.
evnFrameNumber	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	The number of the current frame.

Redundancies in Input Values

You may notice that certain values appear to be redundant. For example, ifrmofsbitLayerStart is redundant because it is always equal to (ifrmofsbytLayerStart * 8). In all such cases this redundancy is provided very deliberately. This serves efficiency so that, for example, you can use ifrmofsbitLayerStart in preference to either setting your own variable to (ifrmofsbytLayerStart * 8) or repeatedly using the expression (ifrmofsbytLayerStart * 8).

Here is one other example of redundancy:

ifrmofsbitLayerEnd = ifrmofsbitLayerStart + (inbytLeftInFullLayer*8) – 1

Truncated Frames

In most cases, complete frames are available for decoding. However, the protocol analyzer will work with frames that have been truncated. This feature can be very handy when frames tend to be long and only the protocol information at the start of each is of interest but not the "data" that follow. Such truncation often occurs when frames are imported from some other source.

As a reminder, the values that pertain to truncated frames are:

inbytLeftInAvailableLayer
inbytLeftInFullLayer
inbytTruncated

Whenever you write a method that directly accesses the abytLayer array, we urge you to consistently use the relevant values (especially inbytLeftInAvailableLayer) to verify that what you want to access is really there before grabbing it!

The Current Field

Methods of many types have access to what we call "the current field". By this we mean the field being decoded. This has obvious meaning only to those methods invoked directly from FIELD and GROUP FIELD statements. (These methods are Format, Size, StartBit, RetrievingData, and Verify Methods.)

Other Resources

This section describes various functions that are built in to the protocol analyzer and available to method writers. These functions are available to methods of all types.

When calling one of these functions in a method, you must preface it by "MTH::" as in this example:

```
int i = MTH::TableGetDetail(tblParam1, i64CurrentField, bIsDefault);
```

Miscellaneous Functions

Here are a couple of general utility functions. Methods may of course also call C runtime-library functions and Windows API functions.

Function Name	Purpose
GetNextByteBoundary	Adjusts an offset to the next byte boundary
GetLongField	Extracts data with optional byte reordering
GetRawFrame	Extracts the raw data for an entire frame and returns true if successful

int **GetNextByteBoundary**(int ibitOffset) takes a number representing a bit offset and returns a byte offset. If the bit offset points to a byte boundary then the byte offset of that byte is returned. Otherwise the byte offset of the next byte is returned. Use of this function is exemplified by the StartBit Method called NextByteBoundary.

```
START_BIT
METHOD __NextByteBoundary__
    /* Moves the pointer to the beginning of the next byte.
            If the pointer is already at a byte boundary, it doesn't move. */
CODE
  iOutput = MTH::GetNextByteBoundary(iofsbitCurrentPosition) * 8;
ENDCODE
```

void **GetLongField**(Abyte& abResults, const BYTE* pabFrame, int iCurrentBit, int iSize, bool bHostDataOrder)extracts iSize bits starting at bit offset iCurrentBit from the byte array pabFrame and stores them into abResults. If bHostDataOrder is true then the byte ordering is reversed. This function is most commonly used to extract data from abytLayer.

bool **GetRawFrame**(pCaptBuff, evn, abData) extracts the raw data from an entire frame and returns true if successful.

```
__int64 evn;        // This is the frame number. It must come from the
                    // evnFrameNumber parameter. If you want
                    // to retrieve a different frame, you must save that
                    // frame's evnFrameNumber in inter-frame data to
                    // retrieve it now.

Abyte abData;       // The array to receive the data.

if (MTH::fxnGetRawFrame(pCaptBuff, evn, abData))// get the frame
{
    for (int i = 0; i < abData.GetSize(); i++)
    {
            // abData[i] is one byte of the returned frame.
    }
}
else
{
    // error case - we didn't get the frame.
}
```

Defining a Method

Finally we can now get to the heart of the matter and tell you how to create a method. Let us start with another example:

```
BOOLEAN
METHOD __FieldIsBetween__
/* Checks a field value to see if it falls between two other values. */
CODE
    bOk = ((ai64Field[fldParam3] >= i64Param1)
              && (ai64Field[fldParam3] <= i64Param2));
ENDCODE
PARAM "Low value" int64
PARAM "High value" int64
PARAM "Which field" field
```

The method starts with a keyword that states its type, BOOLEAN in this case. Then comes the keyword METHOD followed by the method's name and a comment describing what the method does. The actual C++ code of the method is delimited by the keywords CODE and ENDCODE. Finally comes the list of parameters. Each parameter is defined by a statement of the form:

```
PARAM "<description>" <type> [= <default value>]
```

<description> text that describes the parameter.
<type> is one of the parameter types listed in the table in the following section. [=<default value>] allows you to make the parameter optional by specifying the default value to use if the decoder writer doesn't include one. A good example of this is the Size method Fixed, which allows the Units parameter to be optional by specifying a default of "Bytes".

A method may have up to 15 parameters. Only input parameters are allowed.

Parameter Types

Methods can be created with parameters of your choice. The types of parameter available are:

Declared as	Description	Parameter Name
field	Name of a field	fldParam#
float	IEEE-standard double-precision (64-bit) floating-point number (double in C++)	dParam#
int	32-bit signed integer	iParam#
int64	64-bit signed integer	i64Param#
list	One of a given enumeration	eParam#
str	ASCII string	strParam#
table	Name of a DecoderScript table	tblParam#

The ‘#’ in the parameter name is replaced by its ordinal (1, 2, 3,… 15) as you can see in the FieldIsBetween example above.

When a method has a field parameter, the caller inserts the name of a field as the actual parameter. The value that arrives in the fldParam# parameter is however the index of that field in the ai64Field array. “–1” is also allowed as an actual field parameter; it means “no field” and the method may treat that as it will. The idea is that it allows a field parameter to be optional.

For an example of how a list parameter is defined and used, see the example in the section on Boolean Methods above.

Output Values

The purpose of most methods is to produce some output, for example a string for display in the case of a Format Method. In every such case, the output is stored into designated variables. Methods do not return values and may not transmit results through parameters.

Output variables are described in detail in the following sections covering each method type. Here is a summary:

Method Type	Output Variable	Significance
Boolean	bool bOk	true or false
Branch	int iOutput	reference to branch field in a table entry
ByteStreamFramer	bool bFrameTransformed; CArray<WORD,WORD> ausLowIndex; CArray<WORD,WORD> ausHighIndex; CArray<BYTE,BYTE> abVirtualFrame ebsfReturn eAction	info so that the layer can be parsed.
Format	CString csOutput	formatted string for display
Frame Transformer	bool bFrameTransformed; CArray<WORD,WORD> ausLowIndex; CArray<WORD,WORD> ausHighIndex; CArray<BYTE,BYTE> abVirtualFrame	(see Chapter 9)
NextProtocol	__int64 i64Output	numeric code for protocol
NextProtocolOffset	int iOutput	offset in bytes
NextProtocolSize	int iOutput	size in bytes
PreProcessing		(none)
Processing		(none)
PostProcessing		(none)

Method Type	Output Variable	Significance
Retrieval	__int64 i64FieldValue	adjusted value
	int iFieldSize	size
Size	int iOutput	bit count
StartBit	int iOutput	bit offset
Tag	CString csOutput	tag name
Verify	bool bOk	true or false
	CString csOutput	expected result if verification fails

Method Contexts and Examples

We now describe the input parameters, output variables and other context resources for each method type in turn. In many cases we quote examples that correspond to standard methods but the code we publish here is not necessarily identical to the code the real method is built from.

Boolean Methods

Boolean Methods set the variable bOk to either true or false (as normally defined in C++). We have already given one example, FieldIsBetween above. Here is another that also serves as an example of how list parameters are used:

```
BOOLEAN
METHOD __FieldIs__
/* Makes a specified comparison on a given field value */
CODE
    switch (eParam1)
    {
    case eGreaterThan:
            bOk = ai64Field[fldParam3] > i64Param2;
            break;
    case eGreaterThanOrEqualTo:
            bOk = ai64Field[fldParam3] >= i64Param2;
            break;
    case eLessThan:
            bOk = ai64Field[fldParam3] < i64Param2;
            break;
    case eLessThanOrEqualTo:
            bOk = ai64Field[fldParam3] <= i64Param2;
            break;
    case eEqualTo:
            bOk = ai64Field[fldParam3] == i64Param2;
            break;
```

```
        case eNotEqualTo:
                bOk = ai64Field[fldParam3] != i64Param2;
                break;
        default:
                bOk = FALSE;
        }
    ENDCODE
    PARAM "Operator" list {GreaterThan GreaterThanOrEqualTo LessThan LessThanOrEqualTo
    EqualTo NotEqualTo}
    PARAM "Match value" int64
    PARAM "Field to test" field
```

Note that when one of the enumerated values for a list parameter is used within the method code, it is prefixed by an 'e'.

Branch Methods

A Branch Method stores a reference to a FIELD or GROUP into the iOutput variable. This can be seen in the FromTable method:

```
    BRANCH
    METHOD __FromTable__
     /* gets the field from the table */
    CODE
        bool bIsDefault;
        iOutput = MTH::TableGetBranchField(tblParam1,
     ai64Field[fldParam2], bIsDefault);
    ENDCODE
    PARAM "Which table" table PARAM "Which field" field
```

Byte Stream Framer Methods

This allows you to process the layer as more than one frame. See the discussion for PAYLOAD_IS_BYTE_STREAM for more information. This method is only called if the layer directly above it in the stack has the PAYLOAD_IS_BYTE_STREAM keyword.

This method has some things in common with a frame recognizer in that its job is to split a byte stream into sections. However, it has some important differences, too. First, it is called on just the payload portion of the frame, and the payloads can be split up across many frames, so the data it works on is not contiguous. Second, this method has access to all the data in the payload at once, unlike the recognizer, which only has access to one byte at a time.

Payloads could be split in a few different ways. Your particular payload may not exhibit all these possibilities. The easiest possibility is that the payload is exactly one layer. (That is also the normal case, where we aren't doing byte stream framing). Another possibility is that the payload could contain two or more layers, but the last byte of the payload is always the end of a layer.

The third case is that the payload is completely independent of physical layers that it resides; that is, a payload can contain 1½ frames, or ½ of a frame, or a layer can extend over many frames. In this third case, you will need to keep the information from previous frames as inter-frame data.

This method returns a value in the variable eAction. The legal values are:
ebsfTotallyContained: The layer is completely contained in this payload.
ebsfReconstructed: This method has reconstructed the layer in a manner similar to Frame Transformation methods.
ebsfPartial: This payload is just part of a layer. Wait until the next payload comes in to decode it.

If the return value is ebsfReconstructed, then the three variables ausLowIndex, ausHighIndex, and abVirtualFrame must be set.

If the return value is ebsfTotallyContained, then the variable iBytesUsed is set to the number of bytes consumed by this layer.

Format Methods

Format Methods put whatever output they produce into the csOutput object. The simplest example is:

```
FORMAT
METHOD __Constant__
    /* just outputs the string passed into it */
CODE
    csOutput = csParam1;
ENDCODE
PARAM "String to print" Str
```

Because custom Format Methods are the most popular, we will give another example. This example formats a string of bytes as unsigned decimal integers. It is equivalent in function to the standard method called StringOfDecimal, but is coded differently.

```
FORMAT
METHOD __MyStringOfDecimal__
    /* formats a string of decimal bytes */
CODE
    int iSize = isizbitCurrentField / 8;
    int iStart = iofsbitCurrentField / 8;
    int iPrintSize = min(iParam1, iSize);
    char* pOutput = csOutput.GetBuffer(iPrintSize*4+5);

    pOutput[0] = '\0';

    for (int i = 0; i < iPrintSize; i++)
            sprintf(&pOutput[i*4], "%-3u ", abytLayer[iStart+i]);
```

```
    csOutput.ReleaseBuffer();
    if (iPrintSize != iSize)
            csOutput += "...";
ENDCODE
PARAM "Length of display string" int
```

Since this method must be capable of handling a field that is longer than 8 bytes (64 bits) we cannot use i64CurrentField. Instead we access the field data through the byte array called abytLayer which holds the undecoded data for the protocol layer. The value in iofsbytCurrentPosition gives us the index into this array of the first byte of the current field. We simply loop through the bytes formatting each one with spaces in between. A lot of the code simply deals with making sure the string ends up being of appropriate length and adding an ellipsis if there is any truncation. The one parameter is the maximum length of the formatted string in characters.

Note that if a method that works directly with abytLayer is called for a field that happens to be 64 bits or less in size then the effect of any Retrieval Methods applied to the field value will not be reflected in the value that is formatted.

NextProtocol Methods

A NextProtocol Method produces a number and stores it into the variable i64Output. Here is the simplest example where the code is extracted from a named field:

```
NEXT_PROTOCOL
METHOD __FromField__
    /* obtains the value in a named field */
CODE
    i64Output = ai64Field[fldParam1];
ENDCODE
PARAM "Field that contains the next protocol" field
```

NextProtocolOffset and NextProtocolSize Methods

A method of either of these types obtains or calculates the offset/size of the next protocol layer and stores it into the variable called iOutput as in this example:

```
NEXT_PROTOCOL_SIZE
METHOD __FromField__
    /* obtains the size from a given field */
CODE
    iOutput = (int)ai64Field[fldParam1] + iParam2;
ENDCODE
PARAM "Field that contains the next protocol size" field
PARAM "Adjustment" int
```

PreProcessing, Processing and PostProcessing Methods

These follow the usual pattern opening with the keywords PREPROCESSING, PROCESSING and POSTPROCESSING. They create no output and the only external effects they can manifest are via Decoder Data Objects and by throwing exceptions. PreProcessing methods can also initialize values in ai64Field. You can have multiple PRE-, POST- and PROCESSING directives in a Decoder. Pre- and PostProcessing methods are used in the header, while Processing methods are used in individual FIELDs.

Retrieval Methods

A Retrieval Method normally modifies the value supplied to it in i64FieldValue and may also adjust the size of this value as supplied in iFieldSize. When a sequence of Retrieval Methods is invoked, these values are initialized to the value and size of the current field. The end result of such a sequence is either to change the value and/or size of the current field or to set the value and size of a variable. The actual setting of the value and size of the field (i64CurrentField and isizbitCurrentField) or variable is done behind the scenes.

It is quite acceptable and sometimes useful for a Retrieval Method to operate without changing either i64FieldValue or iFieldSize. The typical case would be where the method stores the field value and/or size into a Decoder Data Object. This use of a Retrieval Method is discussed later on.

Here is a simple example:

```
RETRIEVING_DATA
METHOD __AndMask__
    /* AND the current field with the parameter */
CODE
    i64FieldValue = i64FieldValue & i64Param1;
ENDCODE
PARAM "Mask to apply to data" int64
```

Size Methods

Size Methods set the iOutput variable to the number of bits determined to be the size of the field. Here is an example in a method that sizes a null-terminated string of 8-bit characters:

```
SIZE
METHOD __String__
    /* size is from current byte to the first null */
CODE
{
    int iEndPtr = iofsbytCurrentPosition;
    while ((iEndPtr < inbytAvailableLayer-1) &&
              (abytLayer[iEndPtr] != 0))
              iEndPtr++;
```

```
    iOutput = (iEndPtr-iofsbytCurrentPosition+1)*8;
}
ENDCODE
```

Many Size Methods take a parameter that identifies a unit of measure (bit, byte, etc.). The Fixed method serves as a good example of this:

```
SIZE
METHOD __Fixed__
    /* returns the number of bits specified */
CODE
    iOutput = iParam1 * __SizeOfUnitsInBits__(eParam2);
ENDCODE
PARAM "Number (in units)" int
PARAM "Units" list {Bit Bits Nibble Nibbles Byte Bytes Octet Octets Word Words Dword
Dwords Qword Qwords} = Bytes
```

Note the defaulting of the second parameter to “Bytes”.

StartBit Methods

A StartBit method sets the iOutput variable to the bit distance from the start of the layer. And Move is our example here:

```
START_BIT
METHOD __Move__
    /* moves the pointer the requested number of bits, negative
     * numbers move back */
CODE
    iOutput = iofsbitCurrentPosition + iParam1 * __SizeOfUnitsInBits__(eParam2);
ENDCODE
PARAM "Amount to move" int
PARAM "Units" list {Bit Bits Nibble Nibbles Byte Bytes Octet Octets Word Words Dword
Dwords Qword Qwords} = Bytes
```

Tag Methods

These take no input parameters, and output a string. This string is the name of the tag that will be used for this field. Commonly, it is sufficient to use the standard method, Tag, which just returns whatever string is passed to it. You may enforce tag naming standards by writing a tag method that formats a string in a certain way. This method type is not called for each frame, but on demand, so it must return the same value no matter when it is called. Therefore, it doesn’t make sense to use inter-frame data, and it has no access to the current frame. The output string must come completely from the parameters passed to it.

Here is the code for the standard method:

```
TAG
METHOD Tag /* This returns the constant string passed to it */
CODE
    csOutput = csParam1;
ENDCODE
PARAM "Constant" Str
```

Verify Methods

Verify Methods set the Boolean variable bOk to true or false and, in the case of false, the object csOutput to a string indicating what the value was expected to be. Here is code for the IsInTable method which checks that there is an explicit entry in a given table for the current field value.

```
VERIFY
METHOD __IsInTable__
/* returns OK if the current field value matches an entry other than the default */
CODE
    /* First try to get anything out of the table. It doesn't matter
    which function we ask for because we only care about the
    bIsDefault value, which is the same for all four of the calls.*/
    bool bIsDefault;
    MTH::TableGetDetail(tblParam1, i64CurrentField, bIsDefault);
    /* If bIsDefault is false that means it is IN the table, so
    reverse the sense.*/
    bOk = !bIsDefault;
    /* The user can reverse the test by adding "NOT" in the
    decoder.*/
    if (bNot)
    bOk = !bOk;
    /*If we failed the test, we will add some text explaining what
    the error is.*/
    if (!bOk)
      csOutput = "Entry Is Not In Table";
ENDCODE
PARAM "Table" table
```

Note the reference to bNot. Any Verify Method that checks a field (that is, one other than a checksum/CRC checker) must apply bNot as appropriate. In the case of IsInTable, when a failure occurs, the Method returns “Entry Is Not In Table”.

Programming Exceptions

If you need to throw an exception (in the C++ sense) in your method, you may use either of two facilities intended for the purpose:

```
throw CfdFrameException("<error message>");

throw CfdFieldException("<error message>");
```

As the names imply, the first form aborts decoding of the entire current frame while the second merely aborts decoding of the current field. The parameter to the exception is a string explaining the cause of the exception that will be displayed in the Decode Pane.

Here is an example:

```
RETRIEVING_DATA
METHOD __DivideInteger__
/* Divides the current field value by an integer constant */
CODE
    if (i64Param1 == 0)
    {
            throw CFdFieldException("Divide by zero error. Integer parameter is zero.");
            i64FieldValue = -1;
    }
    else i64FieldValue = i64CurrentField / i64Param1;
ENDCODE
PARAM "Constant value to divide by" int64
```

Common

In your .MTH file you may include a common section. This usually goes at the top and contains anything you need to have there that is other than methods. The common section is the place to put general #includes – that is for .h, .cpp or any other appropriate type of file other than an .MH which should be included outside the common section. A common section could also contain class, type and structure definitions, #defines, and so on. It is delimited by the keywords COMMON and ENDCOMMON as in this example taken from FTESTD.MTH.

```
COMMON
    static int __SizeOfUnitsInBits__(int iUnits)
    {
                static int aiSize[] = { 1, 1, 1, 4, 4, 8, 8, 8, 8, 16, 16, 32, 32, 64, 64 };
                if ((iUnits < 0) || (iUnits >= sizeof(aiSize)/sizeof(aiSize[0])))
                return 0;
            return aiSize[iUnits];
    }
ENDCOMMON
```

Be careful with the common section. It is easy to misuse it and put code and variables in it that could better be put somewhere else. Think of it as equivalent to global data, and avoid using it more than necessary. In particular, although it is legal to do so, never declare variables in this section. Because methods get executed in unexpected ways, you can never be sure what the state of a variable you declare here will be. (See Chapter 10, Understanding How the Decoding Process Affects Data Objects if you want to know why). If you are tempted to declare a variable here, you probably need to use the inter-frame data construct explained in Chapter 10. You can declare constants in this section without harm.

Where do Custom Methods reside?

The standard methods provided with the protocol analyzer reside in DLLs in the directory "<installation path>\App Data\Methods". Sources are generally not included. Methods written by customers should be placed in a different directory.
When the protocol analyzer is installed it creates a number of subdirectories that begin with "My". One of these is "My Methods", and that is where custom methods belong. Your source module should be placed in a subdirectory of "My Methods" called "Source" with an extension of "MTH". Thus, if you create a set of methods for a protocol called Proto, then you might save it as file "<installation path>\My Methods\Source\Proto.MTH". You can put as many methods into one file as you wish. If you prefer, you may also put methods that are to be compiled together into separate files and then include these in your .MTH file. Our convention is to name such with an extension of .MH, but you are free to use any name. Such inclusions are specified as follows:

```
include "MyMh.mh"
```

When your methods are compiled and linked, they get built into a DLL that is deposited in the "<installation path>\My Methods" subdirectory.

How does a Custom Method get compiled?

You must have a suitable version of Microsoft Visual C++.NET 2003 (7.1 compiler) installed on your system but you do not invoke the build yourself. Instead, run the protocol analyzer in Capture File Viewer and select "Recompile Methods" from the Methods menu (or click the icon that resembles a pile of bricks). In fact just starting the program may be enough; it checks to see if you have any methods that need to be recompiled and, if it finds any, asks you if it should proceed with that.

Chapter 7: Frame Recognizers

Before protocol data can be decoded, it must be split up into frames and then layers. A decoder, remember, is designed to process one frame at a time. If we look at one side of a communication session, we see a succession of frames like this:

Frame	Frame	Frame	Frame	Frame

The question we examine in this chapter is: how are the boundaries between frames determined?

In some cases, such as with Ethernet, the hardware sends complete frames. In other cases, such as with bit-synchronous protocols (HDLC, SDLC, etc.) the hardware detects the beginnings and end of frames and hands notifications of these events to the protocol analyzer interspersed with byte-at-a-time data. Then there are cases where the data are received as an unstructured byte stream; this is typical of asynchronous protocols such as SLIP and PPP. In this last situation, the protocol analyzer must depend on software to figure out where frames begin and end. Such a piece of software is called a Frame Recognizer. If your protocol needs a Frame Recognizer you might be lucky enough to be able to use one of the standard recognizers supplied with the protocol analyzer, but most likely you will need to develop a custom one.

No matter how frames are recognized, start-of-frame and end-of-frame events will be inserted into the captured data to mark the boundaries. When a Frame Recognizer is used, data is normally fed through it right after being captured so that it can be decoded and displayed in real time. It is also possible to reframe such captured data at a later stage by selecting "Reframe…" from the File menu of the Frame Display. This process removes existing start- and end-of-frame events and feeds the data through the Frame Recognizer again. This is very handy when you are debugging a Frame Recognizer and also serves when you capture data without knowing or specifying what protocol is in use.

In a nutshell, a Frame Recognizer is called for each event in sequence. Its job is to spot where frames begin and end, and for each byte, to return an indicator of frame state.

Recognizer Syntax

Frame Recognizers are similar to methods. They may even be referred to as compound methods. A Frame Recognizer takes the following form:

```
RECOGNIZER name [PER_STREAM]
    <stuff that appears in the class definition such as private variables>
    @RESET
    <code - Called when there is a reset, when first starting, and when the lowest layer in the
    protocol stack changes>
    @CTOR
    <code - Constructor>
    //
```

```
// Note that the @RESET code will be called right after the
// constructor. Therefore the constructor need not and
// should not do any work that is better done is @RESET.
//
@DTOR
<code - Destructor>
@EVENT_HANDLER
<code - Called for the NOTIFY type events (defined below)>
@SIGNAL_CHANGE
<code - Called when a signal-change event happens>
@TIMER
<code - Called when a timer event occurs>
//
// Periodic calls are made to this code with the current time
// as a parameter. This gives the Frame Recognizer a chance
// to time-out.
//
@IMPLEMENTATION
<code - Anything else that needs to be defined, such as private functions.>
//
// Use "<recognizer_name>::<member_name>" when referring to
// members within this block.
//
@EACH_BYTE
<code - Called for each byte received>
@END_RECOGNIZER
[PARAM "comment" type]...
```

All of the code blocks are optional. If a block is omitted then its tag (e.g. @TIMER) should be omitted as well.

If PER_STREAM is present, then an instance of the recognizer is created for each stream. The instance for one stream cannot "see" another stream's instance.

Parameters are declared as for regular methods and are visible to all code blocks.

Input Values

The input values available to recognizers:

const int **iByte** contains a character read from a communication stream.

const int **iDrf** contains error indications.

const int **iNdrf** contains signal-change indications.

const ePdaNotifyEventReasons **nerReason** holds the code for a notification. It is defined as an enum of the following symbols:

nerCbStarted	=> Capture Buffer Started
nerCbReset	=> Capture Buffer Reset
nerIoConfigChanged	=> I/O Configuration Changed
nerInitialize	=> Initialization

const int **iStream** holds the index of the current data stream just as for methods.

const __int64 **i64Timestamp** holds a time value. The format of this value exactly matches the FILETIME type defined by Windows. Timestamps are intended for the convenience of certain Frame Recognizers that must use a time-out to determine when a frame has ended.

Here is a list of the input values available to each block:

RESET	iStream
CTOR	iStream
DTOR	
EVENT_HANDLER	iStream, nerReason
SIGNAL_CHANGE	iStream, iNdrf, i64Timestamp
TIMER	iStream, i64Timestamp
IMPLEMENTATION	
EACH_BYTE	iStream, iByte, iDrf, iNdrf, i64Timestamp

Return Values

The SIGNAL_CHANGE, TIMER and EACH_BYTE code blocks must return one of the following values. Other blocks do not return anything. In the following descriptions, SOF means Start-Of-Frame and EOF means End-Of-Frame.

eBrokenFrame	Frame is broken (usually means the end of the frame didn't appear when we expected it to). A broken frame marker is inserted into the capture following the current event.
eContinue	No change. (No SOF or EOF is inserted into the capture for the current event.)
eInsertEofAfter	Insert EOF after this event.
eInsertOtherSideEofBefore	Insert EOF into other stream before this event.
eInsertSofBefore	Insert SOF before this event.
eInsertSofAfter	Insert SOF after this event.
eInsertThisSideEofBefore	Insert EOF before this event.
eNewFrameEndOtherSide	Insert EOF for the other side and SOF for the current side.
eNewFrameEndThisSide	Insert EOF and SOF for this side before this event.

eSingleByteFrame	Insert SOF for this side before this event and EOF for this side after this event. Used for frames that consist of a single byte.

These values are defined as an enumeration called eRecognizerResult. Thus, should you want to declare a local variable to hold the result your code is going to return, you would do so with a statement such as:

```
eRecognizerResult eResult = eContinue;
```

Some of the most common return types have been enumerated for you. If you need to return something not in the list above, see Appendix 4 for information on how to create a custom return value.

You might wonder about the significance of the "before" in eInsertOtherSideEofBefore. If the EOF is being inserted for the other stream why would it matter if it comes before or after the current event? The answer is that, in situations where this code is used, frames alternate (which is why an event on one side can imply an EOF on the other) and putting the EOF before the current event serves to group it with the frame data from that other side and make for a more appropriate ordering as seen in the Event Pane.

Parameters

You can define parameters for a Frame Recognizer just as for a method with the restriction that such parameters may only be of type float, int, list and string. You cannot have a field or table type parameter since no such entities exist in the recognizer context. This effectively means that any actual parameter to a Frame Recognizer has to be a constant. You might, for example, have a recognizer that handles two or more versions of a protocol and pass the actual version for any particular case as a parameter. Parameters may be referenced in any section of a recognizer.

Error Processing

Frame Recognizers are called in the data collection thread. This means that there is no way to abort if a Frame Recognizer runs into a serious error, because then data collection would stop. If a Frame Recognizer runs into a problem so serious that it cannot continue its work, it must figure out something to do. In such a situation, it is acceptable for it to put itself into a state where it always returns eContinue.

A Frame Recognizer may throw a CfdFrameException exception in its EVENT_HANDLER code. This is the only case allowed; a Frame Recognizer should not generate an exception in any other circumstance.

Efficiency

The EACH_BYTE, TIMER and SIGNAL_CHANGE code of a Frame Recognizer should be written with great attention to efficiency. Of all code that a user might write for use in decoders or methods, this is the most critical. Bear in mind that it is called in real time as each data byte is captured from the communication line.

The Simplest Recognizer: HalfDuplex

Here is a recognizer that is about as simple as can be. It serves for a half-duplex protocol on which frames alternate from one end and then the other. In other words, as the embedded comment indicates, the recognizer assumes that a frame ends and another begins when the direction of traffic changes. (This code is in the file DecoderScript Method Samples.mth file located in My Methods\Source. There are additional examples of simple frame recognizers in this file that demonstrate recognizers based on characters, control signals, simple timing and parity errors.)

```
RECOGNIZER __HalfDuplex__
// The HalfDuplex frame recognizer. This method assumes frame is // everything in a
stream until a byte comes in from
// another stream.
    int m_iCurrentStream;
@RESET
    m_iCurrentStream = -1;
@EACH_BYTE
    // If the last byte we looked at was in the same stream
    // as this one.
    if (m_iCurrentStream == iStream)
    {
            // Remain in the same stream
            return eContinue;
    }
    else
    {
            //Switching streams
            m_iCurrentStream = iStream;

            // Start a new frame and end the previous
            return eNewFrameEndOtherSide;
    }
@END_RECOGNIZER
```

This recognizer has no parameters, and pays no attention to signal changes or even byte values; everything is keyed off iStream. Note that iStream will contain either 0 or 1 so it works to initialize m_iCurrentStream to –1.

A Frame Recognizer for FP

When we introduced our Fictitious Protocol in Chapter 2 we carefully said nothing about how protocol units were framed. Indeed, we did not even suggest what type of communication link FP would be used on. Let us now suppose that FP is used over an RS-232 type of connection and framed like this:

Start Flag	Frame	Stop Flag

using the following values:

Start Flag	F1 hex (or 11110001 binary)
Stop Flag	F2 hex (or 11110010 binary)

Let us look at the Frame Recognizer for FP:

```
RECOGNIZER __FP__ PER_STREAM
    //
    // Fictitious Protocol data
    //
    enum {START_FLAG = 0xF1,  /* frames start with F1 hex */
               STOP_FLAG = 0xF2};  /* frames end with F2 hex */
    //
    // Processing states
    //
    #define STATE_NOT_IN_FRAME    1
    #define STATE_IN_FRAME        2

private:
    int iState;
@RESET
    iState = STATE_NOT_IN_FRAME;
@EACH_BYTE
    //
    // Process the data byte based on the state
    //
    switch (iState)
    {
            //
            // If we are not within a frame then we look
            // only for a frame-start flag.
            //
    case STATE_NOT_IN_FRAME:
            if (iByte == START_FLAG)
            {
                    iState = STATE_IN_FRAME;
                    return eInsertSofAfter;
```

```
            }
            break;

            //
            // If we are within a frame then need to look
            // only for a frame-end flag.
            //
    case STATE_IN_FRAME:
            if (iByte == STOP_FLAG)
            {
                    iState = STATE_NOT_IN_FRAME;
                    return eInsertThisSideEofBefore;
            }
            else if (iByte == START_FLAG)
            {
                    iState = STATE_NOT_IN_FRAME;
                    return eBrokenFrame;
            }
    }

    return eContinue;
@END_RECOGNIZER
```

Almost any Frame Recognizer will need to maintain some member variables, at least to record its state between invocations and perhaps to save other information as well. FP's recognizer uses a private variable iState to hold its state. This state is simply "in frame" or "out of frame".

Let us look at the method piece by piece starting with its framing:

```
RECOGNIZER FP PER_STREAM
…
@END_RECOGNIZER
```

As we observed above, a recognizer is not a method and it does not start and end like a method. You may however include a recognizer in a source file along with methods.

Now let us get into the nitty gritty of recognizing FP frames. First, note that the RESET code initializes the value of our state variable:

```
@RESET
        iState = STATE_NOT_IN_FRAME;
```

Then we find the main work of the recognizer done in the EACH_BYTE block:

```
case STATE_NOT_IN_FRAME:
    if (iByte == START_FLAG)
    {
        iState = STATE_IN_FRAME;
        return eInsertSofAfter;
    }
    break;
```

On finding a start flag, we switch state to "in frame" and return eInsertSofAfter. This indicates that there is a start-of-frame after the current byte. Note that we do not include the start flag within the frame since we do not want the flag itself to appear in our decoder. In some other protocols, this may not be the appropriate procedure. If the flag has meaning beyond that of simply marking the start of a frame then it should be included. Suppose for example that an opening flag can be either F1 or F6 with one indicating a command and the other a response. Then you would want to put the SOF in front of the flag.

The second case is when we are "in frame" and looking for a stop flag:

```
case STATE_IN_FRAME:
    if (iByte == STOP_FLAG)
    {
        iState = STATE_NOT_IN_FRAME;
        return eInsertThisSideEofBefore;
    }
    else if (iByte == START_FLAG)
    {
        iState = STATE_NOT_IN_FRAME;
        return eBrokenFrame;
    }
    break;
```

When we find our stop flag we switch back to "out of frame" state and set the output to eInsertThisSideEofBefore so that an end-of-frame marker will be inserted immediately before the current byte. Why is this value not called eInsertEofBefore? Look back at the list of return codes and you will see there is also one called eInsertOtherSideEofBefore. Inserting an EOF before the current event is the one case where we need to be able to do that for either stream.

To be careful we also check for another start flag since a start flag should never appear inside a frame. If one does, it could be because a frame was incorrectly terminated, because a station died while transmitting a frame, or because a byte was corrupted in transmission. In other words, there is no reliable conclusion we can draw about what is up and we output eBrokenFrame.

Handling Signal-Change Events

The interpretation of signal changes depends totally on the type of communication line that the capture came from. In the case of an RS-232 line, a signal change is inserted to mark each transition of a modem-control signal (DTR, DSR, RTS, CTS, CD, etc.). For Ethernet there are no signal changes. Because they are not universal, no symbols for identifying RS-232 signals are provided.

Parameter 1 is for side A and is an 8 digit hex number. The first 4 digits determine the Start of Frame signal and value. The second 4 digits determine the End of Frame value. The top most bit of each 4 digit number is used to specify the on or off state to watch for. The values for the control signals are as follows:

RTS 0x01
CTS 0x02
DSR 0x04
DTR 0x08
CD 0x10
RI 0x20

Therefore if you want RTS to control side A as on=SOF and off=EOF, parameter
1 should be set to 0x80010001. Following this logic, side B controlled by DSR would require parameter 2 to be set to 0x80080008 yielding the following statement in your decoder: RECOGNIZER (SignalChange 0x80010001 0x80040004) Note that one could combine the above signal values. For example to watch for either RTS or CTS use the value of 3 as follows: (SignalChange 0x80030003,...) Notice that the binary 0x01 and 0x02 have been OR'ed. (0001 OR 0010 = 0011 = 0x03.)

Because the encodings can change, the only way to be really sure what the encoding currently is for a signal change event is to start the protocol analyzer, open the Set I/O Configuration window and click the Names button. The signal changes will be listed in this window in order, where the top item is the low bit. For example, the signal changes in the Names window are listed in the same order as the list above.

Here is an example:

```
@SIGNAL_CHANGE
    if (iNdrf & 0x01)// Is the first signal on?
    {
            iState = STATE_NOT_IN_FRAME;
            return eInsertThisSideEofBefore;
    }
```

Handling Error (DRF, or Data Related Flag) Events

The interpretation of error events depends on the type of communication line that the capture came from in the same way that signal change events do. Error events are encoded in a single byte. As of this printing, the encodings are:

Asynchronous Serial Driver

UART Overrun	0x01
Parity	0x02
Framing	0x04

ComProbe Driver

USART Overrun	0x01
Parity	0x02
Framing	0x04
CRC	0x08
Underrun	0x10

As with signal changes, the only way to be sure that the encoding hasn't changed is to start the protocol analyzer, open the Set I/O Configuration window and click the Names button. The errors will be listed in this window in order, where the top item is the low bit.

Handling Timer Events

The @TIMER block is called once per second. When a call comes to your @TIMER block, you typically want to check the value in i64Timestamp. For example, you can save the timestamp value into a member variable each time a character is received then, when the @TIMER call comes, compare that with the current i64Timestamp, decide if it is appropriate to time out the frame and proceed appropriately. This is facilitated by the fact that i64Timestamp is provided to the @SIGNAL_CHANGE and @EACH_BYTE blocks in addition to @TIMER. You can also use the @TIMER block to check for an idle line by counting the number of ticks since the last byte was received. For example, if the @TIMER block is called 5 times since the last byte was received, you can assume that the line has been idle for 5 seconds.

Loose Ends

The need has arisen to construct Frame Recognizers that depend on conditions such as the state of parity bits. This is an unusual case but can easily be handled since the state of error flags, including parity for a serial connection, is available to the EACH_BYTE code in iDrf.

Many protocols include means to allow data values that match flags to appear within frames. Typically, this is achieved by the use of escape characters or byte stuffing. An escape character is a special code inserted before any sensitive data byte to indicate that it is to be treated as data and not as a control character. Byte stuffing means that sensitive codes in data are converted to a non-sensitive value (or values) and flagged by a special character. The job of stripping escape characters and de-stuffing bytes falls to a Frame Transformer. Frame Recognizers not only are not supposed to do such things but actually have no capability of removing or altering data.

Frame Recognizers Included with the Protocol Analyzer

Frame Recognizers are not exactly methods, but they are necessary components for some decoders. The protocol analyzer includes some standard frame recognizers that may serve your needs. To invoke a frame recognizer for a protocol, put the following statement in the Reference section of your decoder (the section after the decoder ID and before the DECODE keyword):

RECOGNIZER (Recognizer Name)
Following are some recognizers included with the protocol analyzer and how they determine the beginning and end of frames.

Recognizer Name	**Description**
HalfDuplex	Assumes a frame is everything in a stream or channel until a byte comes in from another stream.
ParityError	When a parity error is detected, end the previous frame and begin the next.
TimeGap	Determines framing by comparing the timestamp of the current byte to the previous byte. This recognizer takes a parameter representing the gap between frames. The parameter is expressed in milliseconds. For example, (TimeGap 3) means a gap of greater than 3 milliseconds between bytes will end the previous frame and begin the next.
SignalChange	Framing is controlled by a control signal. This method takes two parameters: an integer representing the control signal used for side A, and an integer representing the control signal used for side B. The Frame Recognizer only looks at the signal states for data bytes, not at signal change events that happen independent of receiving a data byte. Hence, a signal change marking the end of a frame, that occurs independent of a data byte, will be missed by the recognizer.

In the Signal Change recognizer, which signal to look for and whether to look for it going on or off is encoded in 16 bits. The highest bit determines whether to look for the signal going on or off. 1=on, 0=off. The bits for the signals are:

RTS 0x01
CTS 0x02
DSR 0x04
DTR 0x08
CD 0x10
RI 0x20

You can set the recognizer to look for more than one signal, though all signals to watch for must go either on or off. In other words, you can't look for RTS going high and CTS going low as the start of a frame.

For example, to start a frame when RTS goes on and end it when CTS and DSR go off, use the following:

RECOGNIZER (SignalChange 0x80010006 0x80010006)

where each integer is broken down by:

1000		(SOF when the following goes high)
	0001	(RTS is the only signal to pay attention to)
0000		(EOF when the following goes low)
	0110	(pay attention to CTS and DSR).

You have to give the number twice because the first number is for channel A and the second for channel B, where the channels are often DTE and DCE in an asynchronous circuit..

Chapter 8: Frame Transformers

Frame Transformers

Sometimes data is not in a form that is convenient for a decoder to work with. It could be encrypted for example. The protocol analyzer therefore allows you to provide a method that each frame is fed through to prepare it for the decoder. Such a method is called a Frame Transformer.

Typical uses for Frame Transformers are:

To remove escape characters or undo byte stuffing

To decrypt encrypted data

To decompress compressed data

To clean up the data so that it is in suitable condition for CRC/checksum verification

In practice, a Frame Transformer can do any job you find necessary or useful. You could even write a Frame Transformer to make one protocol (which you do not have a decoder for) look like another (which you do have a decoder for).

Note that when decoding a protocol stack, a frame could be fed through multiple transformers; each layer may have its own reasons for transforming the data. Most commonly, however, data-link layers use Frame Transformers.

A Frame Transformer is a method that performs transformations on a frame before or after it is decoded. Most commonly, Frame Transformers are used for such purposes as decrypting data that has been encrypted, decompressing data that has been compressed, undoing byte stuffing, and generally reverting data from a form that it was put in for transmission purposes. You may, however, use Frame Transformers for any purpose you see fit. Here are a couple of examples of how Frame Transformers are used in creative ways:

The decoder for Van Jacobson Uncompressed headers (vju.dec) changes one byte so that the frame then looks like an IP frame.

The decoder for Van Jacobson Compressed headers (vjc.dec) throws away the first several bytes and replaces them with a 40-byte constructed TCP/IP header.

Captured data is normally framed (i.e. passed through a Frame Recognizer) exactly once. Frame transformation, on the other hand, is done every time a frame is decoded. Indeed, it is quite possible for a frame to be transformed several times as it is passed through various decoders.

The term “Frame Transformer” may be a little misleading. When you write a Frame Transformer, the data that you have to work with is strictly the data in your own protocol layer.

A Frame Transformer cannot "see" any data beyond the boundaries of that layer – indeed no method can. Nevertheless, while the view of it is restricted, the entire frame is actually kept through the full decoding process. It is therefore appropriate to say that a transformer does affect the frame as a whole.

A Frame Transformer for FP

To see what a simple Frame Transformer is like, let us look at one for our Fictitious Protocol. You might have noticed a flaw in the FP design (as presented prior to this point). On the one hand, we implied that an FP frame could not contain a byte with value 0xF1 because that matches a Start Flag. On the other hand, a frame may contain a 16-bit actuator/selector value that could certainly include a byte matching 0xF1. Having an End Flag (0xF2) in the body of a frame is a problem for the same reason. The solution we adopt for FP is to define a special encoding for these bytes when they appear within a frame. We reserve a third byte code for this purpose, 0xF0, then encode as follows:

```
F0    ->    F0 00
F1    ->    F0 01
F2    ->    F0 02
```

The rule for decoding is that any time we come across an F0 byte, we discard it and OR the following byte with F0. With that, here is the FP transformer:

```
FRAME_TRANSFORMER
METHOD __FP__ /* Remove byte-stuffings within FP frames */
CODE
{
    // Our task is to convert every two-byte sequence of the
    // form (F0, XX) to the single byte (F0 or XX). This
    // method is used to encode F0, F1, F2.
    enum { STUFF_FLAG = 0xF0 };

    int iIn, iOut;

    // Initialize the arrays. Important: without these lines
    // this routine will access random memory and crash.
    // We don't know how big the output arrays need to be, but
    // we're sure that they won't be bigger than the original,
    // so we'll allocate enough memory to cover the worst case.
    // We will reallocate later when we know the real size.
  ausLowIndex.SetSize(inbytAvailableLayer);
  ausHighIndex.SetSize(inbytAvailableLayer);
  abVirtualFrame.SetSize(inbytAvailableLayer);
    // Process each byte in the frame
    for (iIn = iOut = 0; iOut < inbytAvailableLayer; iOut++)
    {
        // Is this byte a stuff flag?
```

```
        if ((abytLayer[iOut] == STUFF_FLAG)
            && (iOut != inbytAvailableLayer - 1))
        {
            // The previous byte was an introducer and this
            // one is too. Copy this byte into the frame and
            // set the down links so that the byte appears to
            // have come from this byte and the previous one.
            ausLowIndex[iIn]  = (unsigned short) iOut;
            ausHighIndex[iIn] = (unsigned short) (iOut + 1);
            abVirtualFrame[iIn++] = (BYTE)(abytLayer[++iOut] | 0xF0);

            // Set the flag that the frame has been
            // transformed. Note that this is an optimization.
                        // On those frames that have no escape characters,
                        // output frame is the same as the input frame, so
                        // no transformation is necessary.
            bFrameTransformed = true;
        }
        else
        {
            // Just copy this byte to the new frame
            ausLowIndex[iIn]  = (unsigned short) iOut;
            ausHighIndex[iIn] = (unsigned short) iOut;
            abVirtualFrame[iIn++] = abytLayer[iOut];
        }
    }
    // Now we know the correct size of the frame, so if we are
    // transforming, then set the size. If this step is not done,
    // then the end of the frame will contain garbage.
    // If bFrameTransformed is false, then the transformation will
    // be skipped completely, so there is no point using any more
    // time.
    if (bFrameTransformed)
    {
            ausLowIndex.SetSize(iIn);
            ausHighIndex.SetSize(iIn);
            bVirtualFrame.SetSize(iIn);
    }
}
ENDCODE
```

Output Variables

The Frame Transformer creates a new copy of the layer data. It works with the following output variables:

Abyte& **abVirtualFrame** is the array that receives the transformed layer. So abytLayer holds the pre-transform data and abVirtualFrame has the post-transform data.

Aushort& **ausLowIndex** and **ausHighIndex** serve to relate how the old data maps into the new. For each byte abVirtualFrame[i], ausLowIndex[i] holds the index in abytLayer of the first transformed byte and ausHighIndex[i] contains the index in abytLayer of the last transformed byte. These values allow for those bytes that correspond to the selected byte or bytes in the Decode Pane to be highlighted in the Event Pane.

Suppose that, in some hypothetical transformation, nulls are discarded and 10 is an escape character that causes the character following it to be used after 20 has been subtracted from it. Then, if abytLayer comprises [00, 10, 25], abVirtualFrame will end up containing [5], ausLowIndex[0] will be 1 and ausHighIndex[0] will be 2.

A slightly more complicated example:

If abytLayer comprises [00, 10, 25, 10, 26], then abVirtualFrame will contain [5, 6], and the following will be true:

abVirtualFrame[0] = 5, ausLowIndex[0] = 1, ausHighIndex[0] = 2
abVirtualFrame[1] = 6, ausLowIndex[1] = 3, ausHighIndex[1] = 4

bool **bFrameTransformed** is set to true if the method transforms the frame. Depending upon what data is found in the layer, the method may or may not perform any actual transformation. If no transformation is done then bFrameTransformed should be set to false and the other outputs can be left unset. This setting will promote efficiency – two identical copies of a frame will not be stored.

Chapter 9: Decoder Data Objects

In this chapter we discuss Decoder Data Objects, the mechanism by which communication between different methods, different frames, different streams, and different layers is achieved. This chapter is completely self-contained; if you think you will never need to write a Decoder Data Object then you may safely skip it.

A Decoder Data Object (or data object for short) is a named C++ object that holds data for you. You freely define the data area of each data object you use to suit your purpose.

Storing and retrieving data objects is a complex topic that is not intuitive. Before you begin writing a data object, it is important to understand the flow of compiling and reconstructing frames so you understand what is required in a data class and why.

Understanding How The Decoding Process Affects Data Objects

There are two phases in the decoding of a frame.

First, when a capture file is first opened or data is first captured, all frames are processed in the order they are received and certain information is saved in the FRM companion file (this file is stored in the same directory as the capture file, and is given the name of the capture file but with an FRM extension). This is called compiling. Compilation happens only once with each frame as it is received. The *only* output of this phase is the FRM file.

Second, the frame is processed for display as text on the screen. This is called reconstruction. The output of this phase is the text the user sees. Reconstructing the frame may involve retrieving information about the frame that was stored in the FRM file when the frame was compiled. This can happen to any frame, at any time, in any order, any number of times, as the user moves through the screens and clicks on frames.

The compilation stage saves two things: the stack traversal for the particular frame being compiled, and the context data that goes with that frame, which is the data saved by a data object. The reconstruction stage retrieves the context data exactly as it appeared when the frame was first accessed, so that no matter what order the frames are processed in, the context data is the same as it was when the frame was first compiled.

The compilation and reconstruction phases have some different and some similar needs. The program flow is explained in detail below. (References to specific parts of the data class, such as GET_DATA_FROM_A_BYTE_ARRAY, are explained later in this chapter in sections *Decoder Data Object Syntax* and *Using Decoder Data Objects for per-Frame Data*).

Compilation Phase

1. All the possible data objects are created and reset.
2. Each frame is decoded, beginning with the first frame, and sequentially proceeding through each successive frame. Therefore, if a value is changed in your internal data object during the processing of a frame, you know that the value will be retained when the next frame calls your data object.
3. Just before the first time a data object is called in a frame, PUT_DATA_INTO_A_BYTE_ARRAY is called to get a copy of the beginning value of the data object (the data object puts data into a byte array for the frame compiler to read). Note that this is done for each frame.
4. All frame processing is done, including calls to methods that may use and/or change the data object.
5. The next protocol information is determined, possibly with the help of the data object.
6. All of the data objects are given a chance to change or replace the data that they supplied in step 3 through a call to OPTIMIZE_DATA.
7. The frame's protocol stack information and all the data objects are saved in a record in the FRM file.

Reconstruction Phase

1. A particular frame is requested. Frames are requested in response to user actions (filtering, sorting, scrolling through the displays, etc.), which means that this could be any frame at all, in any order.
2. The record corresponding to that frame is loaded from the FRM file. At this point, the stack used in this frame is known, and all the data objects that are needed are known.
3. The needed data objects are initialized, and loaded with the data that was saved in the FRM file. This is done with the GET_DATA_FROM_A_BYTE_ARRAY call, in which the data object gets data from a byte array.
4. All frame processing is done, just like in the compilation phase, except that PROCESSING type methods are not called; however, the FORMAT and VERIFY methods are.
5. The decoded data is now used. The data are the actual strings that appear on the display for a particular frame, along with some supporting information.

There is a pretty big difference in how the compiling and reconstruction phases use their data, and that makes a difference in what needs to be saved. The compiling phase can count on getting frames in order, and therefore knows that any data changed by the previous frame will be the same for the next frame. The reconstruction phase can only count on the fact that the frames will NOT be accessed in any order, and therefore you can only guarantee data that you set yourself each time a frame is accessed. This is the reason for the elaborate PUT DATA... OPTIMIZE DATA… GET DATA... functions. They straighten out the data object before the display instance gets control of it.

For example, if you have a command/response protocol where you need to know which command was sent in order to decode the response properly, the compiling instance can guarantee that the command will have been decoded before the corresponding response frame is encountered. In contrast, the reconstruction instance gets its data from the FRM file and so only has access to what was saved in the record for that frame.

Other considerations:

- Do not save any variables that control what the next protocol is. They will not have any effect in the reconstruction phase.
- Many times a data object will have data that is useful for one frame, but not another. You don't need to save data that is not actually accessed during that frame.
- Sometimes, a lot of data is required to make a single decision. After you make the decision, perhaps you can just save the result instead of all the data used to make the decision.

Implications Of Phases

The compiling and the reconstruction phases can happen simultaneously. For example, the reconstruction phase is being done on frame 10 because the user clicked on it while the compiling phase is occurring for frame 34. This means that there are two instances of each data object: the one used for compiling and the one used for reconstruction.

You should not declare variables in the common section of method files because variables in the common section exist in only one place. This means that the variable will be shared between the compiling and reconstructing instances of a data object. If the user clicks on frame 10 while frame 34 is being compiled, the frame 10 reconstruction instance and the frame 34 compiling instance will be accessing and very likely changing the same variable at the same time.

If the FRM file exists when a capture file is opened and it is up-to-date, the compilation phase doesn't occur. This means that the compiling instance of the data object doesn't exist, so any assumptions made on the value of a variable change whether there is an FRM file or not.

If a data object is written generically, it could be used in two different places by giving a different name to it. This is why you have to supply a data object name as the first parameter to methods that use it. Having one traditional data object for the two different instances will not work.

Ways to share data at different points of the decoding process

The following are the possibilities for sharing data. Your situation will fall into one of these categories. Note that you may mix these types in the same data object if you are careful to keep your member data straight.

1. Sharing data between two fields in the same layer.

You can probably simply use a FIELD to hold the value and not use a data object at all. For instance, if the first field in a decode is a type indicator, later on in the layer you can refer to that field to figure out how to interpret the data. Using the STORE keyword allows you to save whatever data you need in a FIELD.

2. Sharing data between two layers in the same frame.

Sometimes a layer needs to know what a value was in a previously encountered layer in the current frame. For instance, if ABC protocol contains a “frame type” field, and XYZ protocol, which is farther along in the stack, needs the value of that field to know how to decode itself, the ABC layer needs to save the “frame type” for the XYZ layer. In most cases you can simply use the standard methods StoreIntraframeField and IntraframeField to share data between two layers of the same frame. But if the data is complex enough you’ll need to use a data object. However, since all the information is present in the current frame, it can be reconstructed without any outside help. Therefore, the data object you create would not need the PUT_DATA_INTO_A_BYTE_ARRAY, OPTIMIZE_DATA, and GET_DATA_FROM_A_BYTE_ARRAY sections since nothing needs to be saved during compilation and retrieved during reconstruction.

3. Sharing data between two frames.

This is the more complicated case. Sometimes one frame contains a command type, and, when decoding the response frame, you need to know what kind of command was sent because the response frame has a different format. In most cases you can use the standard methods StorePersistentField and PersistentField to share data between two frames. But if you have special needs you need a data object and you need to save the data between frames during compilation with PUT_DATA_INTO_A_BYTE_ARRAY (and possibly OPTIMIZE_DATA) and retrieve that data during reconstruction with GET_DATA_FROM_A_BYTE_ARRAY. See section *Using Decoder Data Objects for per-Frame Data* later in this chapter for details.

Decoder Data Object Syntax

A Decoder Data Object is defined as a special kind of C++ class that is similar to a Frame Recognizer. It takes the following form:

```
DECODER_STATIC name [PER_STREAM] [VERSION number]
<class stuff such as declaration of member variables.>
@CTOR
<code - Constructor>
@DTOR
<code - Destructor>
@RESET
<code - Called when there is a reset. This means forget everything! Called, for example,
when the protocol stack changes.>
@EVENT_HANDLER
<code - Called to NOTIFY type events. See below.>
@PUT_DATA_INTO_A_BYTE_ARRAY
<code - Called to save data from the data object.>
@OPTIMIZE_DATA
<code - Called to save data from the data object.>
@GET_DATA_FROM_A_BYTE_ARRAY
<code - Called to load data into the data object.>
```

```
@IMPLEMENTATION
<code - Anything else that needs to be defined, such as private functions.>
@END_STATIC
```

With one exception, all of the code blocks are optional. If a block is omitted then its tag (e.g. @RESET) should be omitted as well. The exception is that OPTIMIZE_DATA will run only if PUT_DATA_INTO_A_BYTE_ARRAY is defined (PUT_DATA_INTO_A_BYTE_ARRAY can be empty however).

At this point we can explain how a data object works. You define a data class for each set of data that you want to maintain. Then, in your custom methods, you call member functions in the data object to manipulate the data as needed. You can use any number of instances of a data class, each distinguished by a unique name. This is the data name referenced in the section on Intra-frame and Inter-frame Data in Chapter 4.

If PER_STREAM is present in the data class definition, then any instance of the data class is created strictly for use within the current stream. In other words, if you use a data object called ABC then a separate ABC will exist for each stream. The instance for one stream cannot “see” another stream's instance. If you need to pass data between streams then, obviously enough, you use a data class that does not have the PER_STREAM attribute.

If VERSION is not present, the version is zero. Otherwise, it is whatever number is given. This is important because the output of the data object is saved to the FRM file, and is read back in when a capture file is opened. If the data to be saved changes, it won’t be read in correctly from an old FRM file. Therefore, if you change the data format in a data object, you must change the version number so that the FRM file is thrown away and rebuilt.

Just before the first time a data object is called in a frame during the compilation phase, PUT_DATA_INTO_A_BYTE_ARRAY is called to get a copy of the beginning value of the data object. Note that this is done for each frame, and the beginning value is the value the data object had when the frame was first encountered.

OPTIMIZE_DATA is called to get a copy of the value of the data object *after* the frame has been compiled.

GET_DATA_FROM_A_BYTE_ARRAY is used to initialize data objects during the reconstruction phase and load them with the data that was saved in the FRM file.

You cannot define parameters for a data object.

Variables Available to Decoder Data Objects

There are three values available to data objects:

Variable Name	Available To
CArray abStatic	PUT_DATA_INTO_A_BYTE_ARRAY, OPTIMIZE_DATA, and GET_DATA_FROM_A_BYTE_ARRAY
CString szIniFile	EVENT_HANDLER
eFdNotifyEventReasons nerReason	EVENT_HANDLER

abStatic Variable

CArray **abStatic** is a writable/extendable array to the PUT_DATA_INTO_A_BYTE_ARRAY and OPTIMIZE_DATA sections and a constant array to the GET_DATA_FROM_A_BYTE_ARRAY section. To see how to code these sections, see our examples below.

szIniFile Variable

const CString **szIniFile** holds the path of a temporary file that contains certain items that may be useful to an event handler. This file is arranged as a standard Windows INI file and should be accessed using the GetPrivateProfileString and GetPrivateProfileInt functions. An event handler should pick information from this file as needed but should never "remember" the file name.

The contents of the file are illustrated by this example:

```
[Directories]
DecoderDirs=C:\Program Files\MyProductInstallation\App Data\Decoders\;C:\ Program
        Files\MyProductInstallation\My Decoders\;

[Config]
StreamName0=DTE
CharsPerSec0=10416.6666666667
TicksPerChar0=22916
StreamName1=DCE
CharsPerSec1=10416.6666666667
TicksPerChar1=22916
```

DecoderDirs in the Directories section contains a list of directories that may contain decoder (.DEC) files.

In the Config section, StreamName*N* gives the name for the stream when iStream equals *N*. CharsPerSec*N* gives the maximum throughput for stream *N* in characters per second. And TicksPerChar*N* gives the number of 900-nanosecond ticks per character time for stream *N*. Note that the name "CharsPerSec" is used for good reason: this counts characters no matter what size they are (e.g. 7-bit or 8-bit) not necessarily bytes.

nerReason Variable

const ePdaNotifyEventReasons **nerReason** holds the code for an event notification. It is defined as an enum of the following symbols:

```
nerCbStarted          => Capture Buffer Started
nerCbReset                    => Capture Buffer Reset
nerIoConfigChanged    => I/O Configuration Changed
nerInitialize         => Initialization
```

Using Decoder Data Objects for per-Frame Data

As we have indicated, you can use a data object to hold data that has any scope less than or equal to one analysis session. The most common use of data objects, however, is to keep state data that may change with each frame. This is where the PUT_DATA_INTO_A_BYTE_ARRAY, OPTIMIZE_DATA, and GET_DATA_FROM_A_BYTE_ARRAY sections come into play.

PUT_DATA_INTO_A_BYTE_ARRAY is called just before the first time a data object is called in a frame during the compilation phase. It puts the data object's data into byte array abStatic. A good way of saving the data into abStatic is with class CStaticStorage (described below). If nothing needs to be saved then abStatic should be sized to zero bytes. It is perfectly okay to save a different amount of data for different frames.

What data should you save? The answer is the exact data you will be needing to reconstruct the frame. In the PUT_DATA_INTO_A_BYTE_ARRAY section you do not know precisely what data that will be because it is called before the frame is compiled. If you include an OPTIMIZE_DATA section, it is called when the frame has finished compiling. At that point you can then figure out which data is really necessary, and you can change abStatic to make it shorter. Here is an example:

```
@OPTIMIZE_DATA
    /*
    This should not save the items that don't need to be saved.        Create a copy of the
    data so we can manipulate it without changing this instance. We then initialize it with
    the data that would be saved if this routine didn't exist.
    */

CDecoderStaticMyStatic old("");
    old._ReadDataFromAByteArray(abStatic);

    /*
    We now figure out which values were actually used. In this example, let's assume that
    the data consists of an array, only one of which is actually used for any particular frame.
    We then only have to save that one entry. We set a boolean when we process a
    particular entry.
    */

    bool bChanged = false;
```

```
for (int i = 0; i < 255; i++)
{
        if (!old.m_abJustUsed[i])
        {
                old.m_aBigDataItem[i].RemoveAll();
                bChanged = true;
        }
}
if (bChanged)
        old._PutDataInAByteArray(abStatic);
```

When a frame is being reconstructed (such as when a decode for a frame is to be displayed), then the data for the chosen frame is copied from the FRM file to abStatic and GET_DATA_FROM_A_BYTE_ARRAY is called to recreate the state of the data object as it was when that frame was compiled. If abStatic is empty that is always an explicit reset signal; GET_DATA_FROM_A_BYTE_ARRAY will not be called in cases when no data was saved. Instead, the RESET section is executed.

We suggest that when you first write your method, you use only PUT_DATA_INTO_A_BYTE_ARRAY and GET_DATA_FROM_A_BYTE_ARRAY (i.e. don't use OPTIMIZE_DATA) and save your entire object until you have your decoder debugged. You may find, though, that if you have a lot of data in your data object that the FRM file becomes much too large. If you find that to be the case, you may be able to save less data (sometimes *much* less) by using OPTIMIZE_DATA as described above.

OPTIMIZE_DATA is also needed when a FRAME_BEFORE_DECODING statement is being run by the decoder. A FRAME_BEFORE_DECODING statement is executed during compilation but not during reconstruction. Therefore, the data read by GET_DATA_FROM_A_BYTE_ARRAY during reconstruction must include the output of the method invoked by FRAME_BEFORE_DECODING. PUT_DATA_INTO_A_BYTE_ARRAY doesn't help because it runs before FRAME_BEFORE_DECODING. OPTIMIZE_DATA does the trick because it runs after the frame is compiled. Here's an example (the method invoked by FRAME_BEFORE_DECODING is not shown):

```
DECODER_STATIC pima_superframe
public:
        bool m_bSuperframing;
        int m_iBytesLeft;
        Abyte m_abFrameSoFar;

@RESET
        m_bSuperframing = false;
        m_iBytesLeft = 0;
        m_abFrameSoFar.RemoveAll();

@PUT_DATA_INTO_A_BYTE_ARRAY
        abStatic;  // get rid of the "unreferenced" warning
```

```
@OPTIMIZE_DATA
        CStaticStorage ss(m_abFrameSoFar.GetSize()+8);
        ss.Store(m_bSuperframing);
        ss.Store(m_iBytesLeft);
        if (m_abFrameSoFar.GetSize() < 32000)
                ss.Store(m_abFrameSoFar);
        else
        {
                Abyte abTruncatedFrame;
                abTruncatedFrame.Append(m_abFrameSoFar);
                abTruncatedFrame.SetSize(32000);
                ss.Store(abTruncatedFrame);
        }
        ss.RetrieveStream(abStatic);

@GET_DATA_FROM_A_BYTE_ARRAY
        CStaticStorage ss(abStatic);
        ss.Get(m_bSuperframing);
        ss.Get(m_iBytesLeft);
        ss.Get(m_abFrameSoFar);

@END_STATIC
```

Exactly when are these sections called? The short answer is only when necessary. They are not called unless the data is actually used in the current frame. That means that a FIELD must be executed that contains a method that is defined with the USES_STATIC line in it for that particular data object. Normally you won't have to worry about that, but that distinction can be confusing as you are debugging.

Frames are preprocessed as they come in (either from a live source or a capture file) to create the FRM file of per-frame data object snapshots. This file is normally saved between runs so this time-consuming preprocessing need be done only once. We must note one caveat about this process: When you load a capture file, a background thread is launched to perform the preprocessing. Frames that have been captured but not yet preprocessed are marked with green circles in the left margin of the Summary Pane. Such frames are decoded when they are displayed, but since the context provided by preprocessing is absent the decode might be wrong. Once the frame is preprocessed, it is marked with a green dot if it contains the layer in the Summary dropdown, otherwise it's not marked at all (the left margin is blank).

A class to make saving and retrieving easier and more compact

There is a class provided that can make the saving and retrieving of data easy. You do not have to use it, but it useful in most cases. It is called CStaticStorage and is used like this:

```
@PUT_DATA_INTO_A_BYTE_ARRAY
    // write data to abStatic
```

```
    CStaticStorage ss(iApproximateSizeNeeded);
    ss.Store(m_iFirstVariable);
    ss.Store(m_abSecondVariable);
    ss.RetrieveStream(abStatic);

@GET_DATA_FROM_A_BYTE_ARRAY
    // read data from abStatic
    CStaticStorage ss(abStatic);
    ss.Get(m_iFirstVariable);
    ss.Get(m_abSecondVariable);
```

There are two constructors, one for getting the data and one for putting the data. When putting the data, make a guess about how large the array will be and pass that to the constructor. That is for efficiency only. If you guess too low, then extra memory allocation will occur that will slow down the routine, and if you guess too high, then memory is wasted. That memory is recovered soon, though, so it is better to guess a little high.

Then make sure that the Get() and Store() functions are parallel; that is, the variables should be declared in the same order in both routines. Both of these routines are overloaded, so you can call them with many basic types: bool, int, char, __int64, Abyte&, unsigned int, unsigned char, insigned __int64. If you need to save another type, then you will have to break it down yourself and call these functions on the pieces.

The last line when saving the data is always ss.RetrieveStream(abStatic); This takes the saved data from the class and puts it into the real data object variable.

Beyond the ease of use, another advantage of this class is that it compresses the data as it saves it. For instance, if you have a 64-bit integer to save, it would take eight bytes to save it normally. This routine compresses it, so that depending on the contents of that field, only between one and eight bytes are required.

Error Processing

Like a Frame Recognizer, a data object may throw a CfdFrameException exception when inside a routine that is called by a method. A data object should not generate an exception in any other circumstance.

Providing a User Interface to a Decoder Data Object

You can provide an interface to your data object so that users can set the context of the data object in cases where the initial context is not available; for example, in situations where frames that gave the initial start conditions on which decoding of other frames is based were not captured. To provide the user interface, all you need to do is include the following code blocks in your data object. This will automatically cause all needed menu items, dialogs, etc. to appear in the protocol analyzer. These items are described below so you'll know what to look for when using the software.

What the User Sees

If a frame for which the context is missing is encountered AND the method writer has included a user interface (as described in the Code Block section below) to the data object, a dialog (provided by the REQUEST_AUTO_OVERRIDE_DIALOG code block) is displayed that asks the user for the missing frame information will be displayed. The dialog is modeless and always on top so the user can poke around in the protocol analyzer to figure out what information needs to go into the dialog. If the user clicks "I Don't Know", any information that cannot be decoded is dumped to Data (this means that decoding will stop at that point and the remaining bytes will be listed as Data). It is also assumed that the user does not want to see such dialogs again so the *Automatically Request Missing Decoder Information* menu item on the Options menu (see below) is unchecked.

On the Control, Frame Display and Protocol Navigator windows, three menu items appear on the Options menu. They are:

Control window -> Options -> Set Initial Decoder Parameters
Control window -> Options -> Automatically Request Missing Decoder Information
Control window -> Options -> Set Subsequent Decoder Parameters

Set Initial Decoder Parameters pops up a dialog (provided by the REQUEST_PRESET_OVERRIDE_DIALOG code block) which allows the user to supply a starting point for context sensitive data. This is useful when the user knows that the information is missing. The user can preset the information so they are not prompted for it later. One tab will appear for each decoder needing context information.

Automatically Request Missing Decoder Information can be checked or unchecked by the user. If unchecked and the information for a frame is missing, the dialog that asks for the data will not be displayed. The rest of the frame to will be dumped to Data. This item is checked by default.

If the information given in the Set Initial Decoder Parameters dialog is incorrect, the user can correct it from the Frame Display window on a frame-by-frame basis. The user can right-click on a frame and select **Provide <context name supplied by method>**. This will bring up an override dialog (provided by REQUEST_OVERRIDE_DIALOG code block) where the user can supply the desired context for that frame. Alternatively, the user can choose **Set Subsequent Decoder Parameters** from the Options menu, which will bring up the override dialog showing all the places where context data has been overridden.

Code Blocks

You must include all of the following code blocks.

@REQUEST_OVERRIDE_DIALOG

This provides the override dialog used by the right-click menu on the Frame Display and the **Show/Modify Decoding Rules** menu item.

Input

__int64 evnFrameNumber	The number of the currently selected frame in the Summary pane.
IConstructedDialog* dlgTempl	A helper class that can be used to create a dialog without an RC file.

Output

strHelp	String containing any help that should appear when the user clicks "Help" on the override dialog. If the string starts with a ` (a back tick), then what follows is interpreted as a file name and that file will be opened using the program specified by the file extension.

The function returns a CPropertyPage* to the page to be shown.

@REQUEST_PRESET_OVERRIDE_DIALOG

This provides the preset dialog shown when the user selects **Set Decoder Parameters** from the Options menu on the Control, Frame Display and Protocol Navigator windows.

Input

IConstructedDialog* dlgTempl	A helper class that can be used to create a dialog without an RC file.

Output

strHelp	String containing any help that should appear when the user clicks "Help" on the override dialog. If the string starts with a ` (a back tick), then what follows is interpreted as a file name and that file will be opened using the program specified by the file extension.

The function returns a CPropertyPage* to the page to be shown.

@REQUEST_AUTO_OVERRIDE_DIALOG

This is the dialog that automatically appears when missing frame information is discovered. It will not appear if the user unchecks the **Automatically Request Missing Decoder Information** menu item.

Input

__int64 evnFrameNumber	The number of the frame that triggered the override request. This is the frame that the override information will be applied to.
IConstructedDialog* dlgTempl	A helper class that can be used to create a dialog without an RC file.

Output

strHelp	String containing any help that should appear when the user clicks "Help" on the override dialog. If the string starts with a ` (a back tick), then what follows is interpreted as a file name and that file will be opened using the program specified by the file extension.

The function returns a CPropertyPage* to the page to be shown.

@APPLY_OVERRIDE_DATA

Called when the data object should change itself according to the user's wishes. It is called in response to the compilation getting to a frame that the user had previously overridden. The format of the data is unknown. It is up to the object to input and output its internal data in a byte array.

Input

Abyte abOverrideData	The data to apply to the object.

@SET_OVERRIDE_DATA_ON_PAGE

This is called when the Override dialog should be shown and contains the data that was previously set by the user for this frame. You will want to initialize your dialog with this data so the user can see what is currently set.

Input

Abyte abOverrideData	The data to apply to the object.
CPropertyPage* pppPage	The property page you create. This is called before InitDialog(), so the controls are not created yet.

@GET_OVERRIDE_DESCRIPTION

This gets an English description of the override. It is for the dialog that prompts the user to save items that were changed in the capture file.

Input

Abyte abOverrideData	The data to be saved.

Return a CString with a description of the data.

@GET_OVERRIDE_MENU_ITEM_TEXT

This gets the text for the menu item that the user selects to get your Override dialog.

Return a CString with the menu text.

@GET_OVERRIDE_DATA_FROM_PAGE

Called by the protocol analyzer to save the overridden data. You should fill the passed array with the data that is current on your dialog. Whatever you return here is saved and passed back in the above routines.

Output
Abyte abOverrideData

Triggering the Auto Dialog

All methods can set i64UserCanSupplyMissingInfoOnThisFrame to a frame number. If this happens, compilation does not stop, but a request for the automatic dialog is made. The REQUEST_AUTO_OVERRIDE_DIALOG routine may be called. It is probably useful to save some context information so that you can fill in some fields in the dialog for the user.

Writing Methods That Use Decoder Data Objects

Data objects are very flexible because they can be accessed by methods of any kind and can store any kind of data you choose. Probably the most common use is with command-response protocols where any particular response can be fully interpreted only when you know what command it was sent in response to. In such a case, when decoding a command, you want to store away the command code and then retrieve it when decoding the subsequent response. This needs to be done with some care so that you can deal gracefully with situations such as a capture that starts with a response or a response to a command that appeared to be garbled.

Let us suppose that you want to save a field value in a data object as part of the decoding process. You could use a PostProcessing Method along the following lines (the full implementation is in the sample method file):

```
POSTPROCESSING
METHOD __MyPostProcessor__ USES_STATIC __MyStatic__
CODE
    pStatic->SetCommandCode((int) ai64Field[fldParam1]);
ENDCODE
PARAM "Field to be save" field
```

Following the method name you put the keyword USES_STATIC and then the name of the data class. This means that a method may not use more than one data object but that is hardly a significant restriction. Then you automatically get a variable called pStatic that points to the appropriate instance of your data object. Here we obtain the value of the field named as the parameter, cast it to a (32-bit) integer and store it into a member variable called through the data object's member function SetCommandCode.

The last piece of the puzzle is to see how you call the method:

```
RETRIEVE (StoreCommandCode Data command_code)
```

This method invocation has what looks like two parameters even though the method defines only one. What looks like the first parameter (Data) is actually the name of the data object instance that the method is to use. Whenever a method uses a data object, that method must be called with the instance name before the parameters. Here is an example where two instances of a data object are used:

```
FIELD x_field (Fixed 1) RETRIEVE (MyMethod Fred 4)….

FIELD y_field (Fixed 1) RETRIEVE (MyMethod Wilma 5)….
```

An instance of a data object is created automatically on the first occasion that it is referenced. It then persists for the duration of the analysis session. Without careful design, an excessive amount of memory can be used. Don’t use data objects indiscriminately!

Here are some further guidelines on using data objects efficiently. Normally, two in-memory copies of each data object variable are maintained (one for compiling and one for reconstruction), plus a set of copies for each instance name you use. If your data happens to set up, say, a large array that is never changed you can avoid this duplication by declaring the variable to be static in the C++ sense. For example:

```
static CmemoryHog s_memHog;
```

We repeat: This only works if the variable s_memHog is constant. If it changes, you must not declare it static.

Also keep the size of the abStatic array in PUT_DATA_INTO_A_BYTE_ARRAY as small as you can. It might even pay to compress data if you can do that efficiently. Note that the CStaticStorage class automatically compresses data somewhat. The reason for keeping the size of the array small is that this array is saved by writing it to a file.

The one rule about programming data objects is that a its definition and all methods that reference it must be in the same DLL. This means that they must either be in the same .MTH file or at least in .MH files that are all included into one .MTH.

A Simple Example

Let us examine a real use of a data object. This example is the code behind the StorePersistentField and PersistentField methods. These methods are generic inter-frame data methods that allow a decoder writer to save a field value from frame to frame.

```
DECODER_STATIC __Persist__
    // Declare the class just like in C++
    private:
```

```
        __int64 m_i64Value;
        int m_iSize;
        bool m_bIsSet;

    public:
        void Set(__int64 i64Value, int iSize)
        {
            m_i64Value = i64Value;
            m_iSize = iSize;
            m_bIsSet = true;
        };

        bool Get(__int64& i64FieldValue, int& iSize) const
        {
            // be sure the object is initialized
            ASSERT((m_bIsSet == false) || (m_bIsSet == true));
            i64FieldValue = m_i64Value;
            iSize = m_iSize;
            return m_bIsSet;
        };
        void Clear()
        {
            _Reset();
        }

@CTOR
    // Nothing has to be initialized here, so this section can be
    // omitted. This is for initializing one-time items. The RESET
    // section will be called right after the initialization.

@DTOR
    // This is for freeing memory that we've allocated. In this case,
    // there is nothing to free, so this section can be omitted.


@PUT_DATA_INTO_A_BYTE_ARRAY
    // If the data needs to be remembered for new frames, then we
    // need a way to save and restore it to the FRM file. This
    // section saves the class as a BYTE array that can be streamed
    // to the FRM file. It is important to keep the size of the array
    // to the smallest needed, to keep the size of the FRM file down.
    //
    // If this section is omitted, then the RESET section will be
    // called at the beginning of each frame.
    //
    // abStatic is to be filled in
    if (m_bIsSet)
```

```
    {
            int iSize = (m_iSize+7)/8 + 1;
            abStatic.SetSize(iSize);
            BYTE* pb = abStatic.GetData();
            *pb++ = (BYTE)m_iSize;
            memcpy(pb, (BYTE*)&m_i64Value, iSize-1);
    }
    else
            abStatic.RemoveAll();

@GET_DATA_FROM_A_BYTE_ARRAY
    // This is the complement of the above section. It takes the BYTE
    // stream and turns it back into the class data.
    //
    //abStatic contains the static
            m_bIsSet = true;
            const BYTE* pb = abStatic.GetData();
            m_iSize = *pb++;
            m_i64Value = 0;
            int iSize = (m_iSize+7)/8 + 1;
            memcpy((BYTE*)&m_i64Value, pb, iSize-1);

@RESET
    // This is called right after the object is constructed, and
    // through various other events. It should set the object to a
    // known state.
    m_i64Value = -1;
    m_iSize = 0;
    m_bIsSet = false;

@END_STATIC
```

Here is the method that uses this Decoder Data Object:

```
RETRIEVING_DATA
METHOD __StorePersistentField__ USES_STATIC __Persist__
/*
    Saves the value of the current field in the data object
    specified. The item after the method name must be the name of
    the data object in which the field is to be stored.
    Data is retrieved using the __PersistentField__ method.
*/
CODE
    pStatic->Set(i64CurrentField, isizbitCurrentField);

ENDCODE
```

```
RETRIEVING_DATA
METHOD __PersistentField__ USES_STATIC __Persist__
/*
    Retrieves a field stored previously by a __StorePersistentField__ method.
    When using this method, the first item after the method name
    must be the name of the data object in which the field
    was stored.
*/
CODE
    if (!pStatic->Get(i64FieldValue, iFieldSize))
            throw CFdFieldException("Data not present (The field that the data comes from
was not encountered.)");
ENDCODE
```

In one decoder, declare a field like this --

```
FIELD code (Fixed 1) RETRIEVE (StorePersistentField KeepThisCode) (Decimal) "Code"
```

In another decoder, or in a different path in the same decoder, declare a field like this --

```
FIELD retrieved_code (Fixed 0 bits) RETRIEVE (PersistentField KeepThisCode) (Hex)
"Retrieved Code"
```

A Complex Example

Here is an example of a data object that makes use of most of the code sections. It is taken from the decoder for Novell's NetWare Core Protocol (NCP.DEC). NCP is the protocol that carries commands for file and print operations from a client to a NetWare server, and responses in the reverse direction.

The purpose of the NCP data object is a common one, to match responses to commands. Like many protocols, NCP sends commands and responses in a ping-pong fashion. An NCP response does not carry any indication of what type of command it is a response for. For the NCP client, this is no problem: any response is going to be for the last command it sent. Each command that is detected must be noted so that the response can be properly interpreted when it comes. In the case of NCP, the difficulty of doing this is compounded by the fact that a single physical connection may carry multiple logical connections (in other words, several independent NCP sessions).

Let's look at the NCP decoder's use of its data object in a top-down fashion, starting with the decoder statements that make use of it:

```
POSTPROCESSING (NcpDataProcessing Fred conn_numb function_code sub_func type
sequence)
```

```
FIELD reply_function_code (Fixed 0 byte) RETRIEVE (NcpResponseFunction Fred sequence
conn_numb function)(TABLE functions) IN_SUMMARY Function 70 "Function Code"

FIELD reply_sub_function_code (Fixed 0 byte) RETRIEVE (NcpResponseFunction Fred
sequence conn_numb subfunction) (Decimal) IN_SUMMARY "Sub-Function" 70 " Sub-
Function"
```

The first thing you may notice is that just a single instance of the data object is used and that instance is called “Fred”. The PostProcessor Method, NcpDataProcessing, saves two pieces of information when the message is a command. The first of these is a function code, a number that identifies the operation such as “Erase File”. Some functions have subfunctions, for example the function could be “Printer Functions” and the subfunction “Close Spool File”. The first parameter is the logical connection number. Then come the function and subfunction codes, the actual data that needs to be saved. The fourth parameter is the message type (i.e. command, response, or one of a few others) and the last is a sequence number that is embedded in each message.

The Retrieval Method, NcpResponseFunction, retrieves the function or subfunction code for a given connection and sequence number. The last parameter is a list type indicating whether the function or subfunction code is wanted. Here are the methods themselves:

```
POSTPROCESSING
METHOD __NcpDataProcessing__ USES_STATIC __Ncp__
    /* Handles data for the NCP decoder. NCP response
     * frames do not carry the command function they are responding
     * to inside the frame. This saves the command function on
     * request frames so subsequent response frames can be decoded \
     * accurately. */
CODE
    bool bIsCommand = (ai64Field[fldParam4]==0x2222);
    bool bIsResponse = (ai64Field[fldParam4]==0x3333);
    pStatic->Update(bIsCommand, bIsResponse, (BYTE)ai64Field[fldParam5],
              (WORD)ai64Field[fldParam1], (BYTE)ai64Field[fldParam2],
(BYTE)ai64Field[fldParam3]);
ENDCODE
PARAM "Which field has the connection number?" field
PARAM "Which field has the function code?" field
PARAM "Which field has the subfunction code?" field
PARAM "Which field has the cmd/resp?" field
PARAM "Which field has the sequence number?" field

RETRIEVING_DATA
METHOD __NcpResponseFunction__ USES_STATIC __Ncp__
    /* Handles data for the NCP decoder. This matches
     * the response to the previous command. */
CODE
    // using the current value of the TNS field, get the value
```

```
        // from the map. that is what should be returned.
        i64FieldValue = pStatic->Retrieve((BYTE)ai64Field[fldParam1],
                (WORD)ai64Field[fldParam2], (eParam3 == eSubfunction));
        iFieldSize = 8;
    ENDCODE
    PARAM "Which field has the first key?" field
    PARAM "Which field has the second key?" field
    PARAM "Do you want the function or subfunction?" list { Function Subfunction }
```

As you can see, the methods do very little beyond serving as conduits to member functions in the data class. And that data class is defined as:

```
    DECODER_STATIC __Ncp__
        private:
                struct Sky
                {
                        BYTE bySeqNum;
                        WORD wConnNum;

                        operator unsigned long() const
                        {
                                return ((unsigned long)bySeqNum<<16)+wConnNum;
                        };
                        bool operator==(const Sky& rhs) const
                        {
                                if ((bySeqNum == rhs.bySeqNum) &&
                                        (wConnNum == rhs.wConnNum))
                                        return true;
                                else
                                        return false;
                        };
                };
                struct SfCd
                {
                        BYTE byFunctionCode;
                        BYTE bySubfunctionCode;
                };
                struct Spair
                {
                        Sky key;
                        SfCd value;
                };


                CMap<Sky,const Sey&,SfCd,SfCd&> m_map;
        public:
                void Update(bool bIsCommand, bool bIsResponse, BYTE ucSeqNum, WORD
    wConnectionNum,
```

```
                    BYTE ucFunctionCode, BYTE ucSubfunctionCode);
            __int64 Retrieve(BYTE ucSeqNum, WORD wConnectionNum, bool
bWantsSubfunction) const;

@IMPLEMENTATION
    void __Ncp__::Update(bool bIsCommand, bool bIsResponse, BYTE ucSeqNum, WORD
wConnectionNum,
            BYTE ucFunctionCode, BYTE ucSubfunctionCode)
    {
            if (bIsResponse)
            {
                    // This is a response, so let's delete the map entry
                    Sky key;
                    key.bySeqNum = ucSeqNum;
                    key.wConnNum = wConnectionNum;
                    m_map.RemoveKey(key);
            }
            if (bIsCommand)
            {
                    // This is a command, let's add the map entry
                    Sky key;
                    key.bySeqNum = ucSeqNum;
                    key.wConnNum = wConnectionNum;
                    SfCd functionCode;
                    functionCode.byFunctionCode = ucFunctionCode;
                    functionCode.bySubfunctionCode = ucSubfunctionCode;
                    m_map[key] = functionCode;
            }
    }

    __int64 __Ncp__::Retrieve(BYTE ucSeqNum, WORD wConnectionNum, bool
bWantsSubfunction) const
    {
            // using the current value of the TNS field, get the value
            // from the map. that is what should be returned.
            Sky key;
            key.bySeqNum = ucSeqNum;
            key.wConnNum = wConnectionNum;
            SfCd value;
            if (!m_map.Lookup(key, value))
            {
                    return 0xffff;     // a value that could never be really in the map is
returned if the value was not found.
            }
            else
            {
                    if (bWantsSubfunction)
```

```
                        return value.bySubfunctionCode;
                else
                        return value.byFunctionCode;
            }
    }

@GET_DATA_FROM_A_BYTE_ARRAY
    // read data from abStatic
    m_map.RemoveAll();
    CStaticStorage ss(abStatic);
    int iSize;
    ss.Get(iSize);
    Sky sKey;
    SfCd sValue;
    for (int i = 0; i < iSize; i++)
    {
            ss.Get(sKey.bySeqNum);
            ss.Get(sKey.wConnNum);
            ss.Get(sValue.byFunctionCode);
            ss.Get(sValue.bySubfunctionCode);
            m_map[sKey] = sValue;
    }

@PUT_DATA_INTO_A_BYTE_ARRAY
    // write data to abStatic
    int iSize = m_map.GetCount();
    CStaticStorage ss(4*iSize);
    ss.Store(iSize);

    Sky sKey;
    SfCd sValue;
    POSITION pos = m_map.GetStartPosition();
    while (pos != NULL)
    {
            m_map.GetNextAssoc(pos, sKey, sValue);
            ss.Store(sKey.bySeqNum);
            ss.Store(sKey.wConnNum);
            ss.Store(sValue.byFunctionCode);
            ss.Store(sValue.bySubfunctionCode);
    }
 ss.RetrieveStream(abStatic);

@RESET
    m_map.RemoveAll();

@END_STATIC
```

Chapter 10: Tips and Tricks

In this chapter we have a potpourri of tips, tricks, caveats and loose ends pertaining to decoders.

Approach to Decoding a Field

Start by considering the size. If the size is not fixed then review the standard Size Methods to see if one will work. Remember that the FromField method allows you to pick up the size from another field including possibly a variable; this offers deep flexibility in calculating field sizes. If you still cannot obtain the required size then you will need to write your own custom Size Method.

Second, consider the value. Given that you have the size, can the data just be pulled out of the layer data? If it cannot then you need to use a Retrieval Method. Again consider the standard set before deciding that you need to write your own.

Only at this point should you consider formatting and verification. Be particularly careful when formatting numeric values that can be negative.

Empty Frames

There are protocols in which an empty frame has a significance. If you are working with such a protocol then you might be tempted to include a conditional statement in your decoder to detect such a frame and display some relevant information for it. You would find however that the statement appears never to work. The reason is that empty frames are completely skipped in the decoding process.

Red items in the Summary Pane

When something is displayed in red in a decode Summary Pane, it is for one of three reasons:

- A VERIFY failure has occurred in the decoder.
- A hardware error occurred. (Look at the Event Display window for details.)
- The decoder did not terminate at the end of the layer. If it terminated prior to the end then the non-decoded fields will show up in red in the Decode Pane.

How Field Values are stored

Each FIELD (or GROUP FIELD) statement is compiled into a table that includes a 64-bit space for holding a field value. For any field that has a size of 64 or fewer bits, the field value is stored in this space. Note that, since no intrinsic numeric representation is specified for a value, no sign extension is performed when a value that is narrower than 64 bits is stored. Suppose, for example, that you have a FIELD statement for a 16-bit field defined by the protocol as containing a signed integer in twos complement notation. Then an instance of this field that contains –1 would have its value stored as (in hex) 0000FFFF not FFFFFFFF. The way around this is to use the SignExtension method in a RETRIEVE clause.

When a field wider than 64 bits, its value is not extracted. Any method that needs to process the value must read it directly from the raw layer data (the abytLayer array). This is how all the StringOfXxx Format Methods work.

Knowledge of how values are stored may help you in understanding certain intricacies of the protocol analyzer. It will give additional insight as to why some format methods (Decimal, Hex, etc.) do not work appropriately for fields greater than 64 bits. Also it should be clear that if a FIELD is decoded more than once, say in a GROUP REPEAT, then each retrieved value for the field overwrites the previous one.

A subtle difference between GROUP and GROUP FIELD

There is one subtle difference between a GROUP and a GROUP FIELD that you may want to keep in mind. When a group in the Decode Pane is selected, the complete range of bits or bytes that comprise the group in the Binary, Radix and Character Panes is selected. By contrast, when you select a group field, only the bits or bytes that comprise the field are highlighted. To a user viewing the Decode Pane, however, a group and a group field look very similar. In fact the following would look identical except for the highlighted bytes.

```
GROUP FIELD grp_fld (Fixed 0 Bits) (Constant "X") "Data"

GROUP grp "Data: X"
```

The lesson to be learned here is that it is a good idea to maintain a clear distinction between groups and group fields by displaying some “real” formatted data for a group field and avoiding colons in group tags. This is an unlikely lesson because it is unlikely that you would find reason to use GROUP FIELD and GROUP in a confusing way to start with (unless you happen to use GROUP FIELDs where GROUPs are more appropriate).

Large Fields

Suppose you want to write a decoder for a protocol that has fields that can be very large. Any email protocol (POP, SMTP, IMAP, X.400) would serve as a good example; one of these may carry an entire email message as a data field possibly running to several megabytes. Fields of such a size can be managed, but it is hardly practical to display the entire contents in the Frame Display window. The conventional thing to do is to display just the first 128 characters in the Decode Pane; if the user needs to see more he can select the field in the Decode Pane and then open the Event Display window that will come up showing the same data. The SMTP decoder (SMTP.DEC) serves as a good case study of working with such a protocol.

Error correcting in GROUP REPEAT constructions

Sometimes with a repeating group, you know what the size of a single iteration is because there is a length embedded in the data. The data that follows could be of a number of types, including some types that you may not be aware of when you write the decoder. It would be nice to be sure that the decoder does not get misaligned with the data if something unexpected happens. This can be handled in the following way:

```
    GROUP object REPEAT SIZE (ToEndOfLayer)
    {
            FIELD type (Fixed 1) (FromTable obj_table object) "Type"
            FIELD length (Fixed 1) (Decimal) "Length"
            BRANCH (FromTable obj_table object)
            FIELD move_to_next START_BIT (MoveRelativeFromField length 1) (Fixed 0)
(Hex) SUPPRESS_DETAIL
    }
```

Note the last field, that just moves the pointer to the start of the next iteration. Even if there is an error in the decode, or an error in the frame, or there is a new object type defined since the decoder was written, still each group begins at the correct place.

How to implement a repeating counter.

Sometimes it is nice to know which iteration of a repeating group the decoder is currently executing. This can be done in the following way:

```
FIELD rep_count (Fixed 1) (Decimal) "Number of repeats"

FIELD zero_counter ;

GROUP rep REPEAT COUNT (FromField Times rep_count)
{
    FIELD print_counter (Fixed 0) RETRIEVE (StoreField counter) (decimal) "Counter"
    FIELD increment_counter ;
}

END_MAIN_PATH

/* Add one to the count for the next iteration */
FIELD increment_counter (Fixed 0) RETRIEVE (StoreField counter) (Hex) SUPPRESS_DETAIL
STORE counter RETRIEVE (AddInteger 1)

/* Initialize the counter to zero */
FIELD zero_counter (Fixed 0) (Hex) SUPPRESS_DETAIL STORE counter RETRIEVE
(StoreInteger 0)
```

Output for a frame that begins with the byte 0x03:

```
Number of repeats: 3
Counter: 0
Counter: 1
Counter:2
```

(An important point is that the increment_counter field MUST have SUPPRESS_DETAIL and not have IN_SUMMARY. Otherwise, the field is encountered twice and the count increases by two.)

Recalcitrant Summary Columns

You may change the widths of the Summary-Pane columns defined in your decoder and then observe that your changes appear to have no effect on the display. If this happens it is most likely because your column-display data have been stored in an INI file, and override what is in the decoder. Fixing the problem is an easy matter of editing the <My Product>.INI file (which you should find in your main installation directory). Find the section named “SummaryColumnWidths” then the entry named “Decoder <YourProtocol>”, and delete the entry.

To explain this phenomenon more fully, what actually happens is the protocol analyzer uses the column widths defined in the decoder as initial defaults. Any time the user adjusts a column width manually then that selection takes precedence. The user can reset to the defaults at any time by selecting Reset to Default Widths from the Summary menu on the Frame Display window.

Overwriting Summary-Pane Column Entries

Consider the following statements designed to decode an FP Set Actuator command in a rather fancier way than we did in the tutorial chapter:

```
GROUP cmdSetActuator
{
        FIELD actuator_number (Fixed 1 Byte) (Decimal)
                "Actuator Number"       VERIFY (FieldIsBetween 1 32)

        FIELD rpt_actnum START_BIT (Move -1 Byte) (Fixed 1 Byte)
            (Constant "Set Actuator") ALSO (Decimal)
            IN_SUMMARY "Command/Response" 0 SUPPRESS_DETAIL

    FIELD actuator_value ;
}
```

The first two FIELD statements decode the actuator-number field twice. This might seem odd at first but can prove very handy. Specifically the first one decodes it for display in the Decode Pane and the second adds it to the display in the Summary Pane. You may find many other clever reasons to repeat a decode in this kind of way.

Note that the second statement includes a StartBit clause that moves the pointer backwards by one byte to the actuator-number field. The first format method produces a constant string “Set Actuator”. The second format method formats the actuator number in decimal creating a combined result of, for example, “Set Actuator 3”. This is then inserted into the Command/Response column in the Summary Pane. Note that this column will already have an entry that has been generated by the command_code FIELD. Here we are sneakily overwriting that field to add some more information to it. Finally we see the keyword SUPPRESS_DETAIL.

That means nothing should be displayed in the Decode Pane – which is exactly what we want because the previous statement produced the appropriate entry for that.

Two more notes about this pair of FIELDs. First, the second FIELD could be removed and the decoder would work perfectly well, just with slightly less information appearing in the Summary Pane. Second, when we decode the field the second time we are obliged to give it a new name.

Readable Protocols

Is there much point in writing a decoder for a protocol that is very readable in the first place? An example of a readable protocol would be SMTP or IMAP where commands are English words and data is mostly text (i.e. email messages or fragments thereof). The need is certainly not so critical as with a protocol such as PPP or X.25 but the answer is nevertheless a definite yes.

With an unreadable protocol, emphasis is placed on the decode pane to view what is going on. Readable protocols rely more on the summary pane. The summary pane gives a comprehensive view of what is taking place in a session. You can filter frames based on type, watch the delays between commands and responses and quickly spot faults by scanning for entries displayed in red.

Using Variables

Variables are only intended for use in odd situations. An example is where the length of a string is given sometimes as a byte and sometimes as a word. Do not create variables just because you want to reference field values while processing other fields; in most cases you can just refer directly to the field name for this purpose.

Be careful to avoid any possibility of referencing a variable before it is defined, especially in a case where there are multiple paths through your decoder each of which is supposed to store a particular variable whose value is then used at the end. One technique that may help in this regard is to initialize variables at the start of the executable section of the decoder. This also serves as an excellent opportunity to document them, as in:

```
/*
** The variable string_len is set to the length of the
** varialbe string if one exists. If there is no such
** string then its value will be left at minus one as
** set here.
*/
FIELD init_var_string_len (Fixed 0) STORE string_len RETRIEVE (StoreInteger –1) (Decimal)
SUPPRESS_DETAIL
```

Note that you can use similar means to perform arithmetic operations on variables. Consider, for example:

```
FIELD arithmetic_1 (Fixed 0) STORE var1 RETRIEVE (StoreField var1) ALSO (MultiplyField
var2) ALSO (SubtractInteger 1) (Decimal) SUPPRESS_DETAIL
```

This multiplies variable var1 by variable var2, subtracts one from that, and then stores the result back into var1. Sure, it would be easier if one could write, say:

```
var1 = var1 * var2 – 1
```

But it may only be one decoder in fifty that needs to do arithmetic on variables in the first place.

You can even display the value of a variable in the decode pane, as with the statement:

```
FIELD display_var1 (Fixed 0) RETRIEVE (StoreField var1) (Decimal) “Variable var1”
```

Storing Data

Three facilities are provided for managing data such that it can be saved during one method invocation and retrieved during another:

1. DecoderScript variables are really just a special case of fields that allow you to store pieces of data of up to 64 bits in size. The unique value of such variables is that they can be referred to in DecoderScript statements and passed to methods. Their scope is limited to the decoding of one particular protocol layer.

2. Standard C++ module-level variables allow you to store arbitrary data between calls to methods that reside in the same DLL. The scope of such variables is at least the decoding of one instance of a layer. We mention this because it is something a programmer might think of doing. We consider the safety of this to be questionable, and do not recommend its use. The problem is that there is no guarantee about the order in which methods will be called. It may vary from release to release, or on how the user manipulates the Frame Display, or several other factors.

3. If you need to store any data that must persist beyond these boundaries then you must use a Decoder Data Object. Decoder Data Objects endure for an entire analysis session.

Storing Data into Decoder Data Objects

There are two ways to save field values into Decoder Data Objects. One is with a PostProcessing Method, and the other is to use a custom Retrieval Method. Note that you are not obliged to set either output of a Retrieval Method (i64FieldValue or iFieldSize); these are preset with the existing values when the method is called. Using a Retrieval Method to store data would be preferable in various circumstances, for example when a field is repeated and each instance of the value needs to be saved.

Chapter 11: One complete example

Here we present the decoder for IP version 4 with added commentary to explain what is going on.

Note that the DecoderScript here is not written totally in line with the recommendations given in this manual. Many single-word field tags, for example, are written without enclosing quotes. This may serve a useful purpose in reminding you that many of our recommendations are just that and not hard-and-fast rules.

```
/*  Internet Protocol version 4. Copyright 2000-2001 Frontline Test Equipment, Inc. All
rights reserved. */
IPv4 0x7f000401

/*
** This identifies the next protocol.
*/
NEXT_PROTOCOL (FromField protocol)
/*
** The length for the next protocol is calculated as the
** length of the entire IP packet (decoded as the field
** named tot_length) minus the length of
** the IP header.
*/
NEXT_PROTOCOL_SIZE (ThisLayerLength tot_length)

/*
** This is a custom method to handle fragmentation.
*/
FRAME_AFTER_DECODING (IpReconstructFragments a)

DECODE

/*
** This table is indexed by the IP version number. It
** identifies certain IP variants such as TUBA (TCP and UDP
** with Bigger Addresses). Full decoding of these variants
** is not supported. Basically version 4 is expected.
*/
TABLE ver_nums
{ 0 "Reserved"}
{ 4 "Internet Protocol ver. 4"}
{ 5 "ST" "ST Datagram Mode"}
{ 6 "SIP" "Simple Internet Protocol"}
{ 7 "TP/IX" "The Next Internet"}
{ 8 "PIP" "P Internet Protocol"}
```

```
{ 9 "TUBA"}
{ DEFAULT "Unassigned"}
ENDTABLE

/*
** This table lists the precedence levels defined for IP.
** The precedence is given by a three-bit field within
** the Type-of-service byte.
*/
TABLE prec
{ 7 "Network Control"}
{ 6 "Internetwork Control"}
{ 5 "CRITIC / ECP"}
{ 4 "Flash Override"}
{ 3 "Flash"}
{ 2 "Immediate"}
{ 1 "Priority"}
{ 0 "Routine"}
ENDTABLE

/*
** The table lists the options specified by the Delay
** bit in the Type-of-service byte.
*/
TABLE delay
{ 0 Normal }
{ 1 Low }
ENDTABLE

/*
** The table lists a pair of options that pertain to both
** the Throughput and the Reliability bit flags in the
** Type-of-service byte.
*/
TABLE norm_hi
{ 0 Normal }
{ 1 High }
ENDTABLE

/*
** The table lists the options specified by the
** may/don’t fragment bit in the Flags field.
*/
TABLE df
{ 0 "May Fragment"}
{ 1 "Do not Fragment"}
ENDTABLE
```

```
/*
** The table lists the options specified by the
** last/more fragment(s) bit in the Flags field.
*/
TABLE mf
{ 0 "Last Fragment"}
{ 1 "More Fragments"}
ENDTABLE

/*
** This table lists the next-layer protocol codes.
** Apart from ICMP, GGP, TCP, EGP, IGP and UDP,
** most of these are rarely encountered.
*/
TABLE protocol
{ 0 "IPv6 Hop-by-Hop Option"}
{ 1 "ICMP" "Internet Control Message Protocol"}
{ 2 "IGMP" "Internet Group Management Protocol"}
{ 3 "GGP" "Gateway-to-Gateway Protocol"}
{ 4 "IP in IP (encapsulation)"}
{ 5 "Streaming protocol"}
{ 6 "TCP" "Transmission Control Protocol"}
{ 7 "CBT"}
{ 8 "EGP" "Exterior Gateway Protocol"}
{ 9 "IGP" "Interior Gateway Protocol"}
{ 10 "BBN RCC Monitoring"}
{ 11 "NVP" "Network Voice Protocol"}
{ 12 "PUP"}
{ 13 "ARGUS"}
{ 14 "EMCON"}
{ 15 "Cross Net Debugger"}
{ 16 "Chaos"}
{ 17 "UDP" "User Datagram Protocol"}
{ 18 "Multiplexing"}
{ 19 "DCN Measurement Subsystems"}
{ 20 "HMP" "Host Monitoring Protocol"}
{ 21 "Packet Radio Measurement"}
{ 22 "XEROX NS IDP"}
{ 23 "TRUNK-1"}
{ 24 "TRUNK-2"}
{ 25 "LEAF-1"}
{ 26 "LEAF-2"}
{ 27 "RDP" "Reliable Data Protocol"}
{ 28 "IRTP" "Internet Reliable Transaction"}
{ 29 "ISO Transport Protocol Class 4"}
{ 30 "Bulk Data Transfer Protocol"}
{ 31 "MFE Network Services Protocol"}
```

```
{ 32 "MERIT Internodal Protocol"}
{ 33 "SEP" "Sequential Exchange Protocol"}
{ 34 "3PC" "Third Party Connect"}
{ 35 "IDPR" "Inter-Domain Policy Routing Protocol"}
{ 36 "XTP"}
{ 37 "DDP" "Datagram Delivery Protocol"}
{ 38 "IDPR Control Message Transport Protocol"}
{ 39 "TP++ Transport Protocol"}
{ 40 "IL Transport Protocol"}
{ 41 "IP ver6" "Internet Protocol version 6"}
{ 42 "SDRP" "Source Demand Routing Protocol"}
{ 43 "Routing Header for IPv6"}
{ 44 "Fragment Header for IPv6"}
{ 45 "Inter-Domain Routing Protocol"}
{ 46 "Reservation Protocol"}
{ 47 "General Routing Encapsulation"}
{ 48 "Mobile Host Routing Protocol"}
{ 49 "BNA"}
{ 50 "Encap Security Payload for IPv6"}
{ 51 "Authentication Header for IPv6"}
{ 52 "Integrated Net Layer Security"}
{ 53 "IP with Encryption"}
{ 54 "NBMA Address Resolution Protocol"}
{ 55 "IP Mobility"}
{ 56 "Transport Layer Security Protocol"}
{ 57 "SKIP"}
{ 58 "ICMP for IP ver6"}
{ 59 "No Next Header for IP ver6"}
{ 60 "Destination Options for IP ver6"}
{ 61 "Any Host Internal Protocol"}
{ 62 "CFTP"}
{ 63 "any local network"}
{ 64 "SATNET and Backroom EXPAK"}
{ 65 "Kryptolan"}
{ 66 "MIT Remote Virtual Disk Protocol"}
{ 67 "Internet Pluribus Packet Core"}
{ 68 "any distributed file system"}
{ 69 "SATNET Monitoring"}
{ 70 "VISA Protocol"}
{ 71 "Internet Packet Core Utility"}
{ 72 "Computer Protocol Network Executive"}
{ 73 "Computer Protocol Heart Beat"}
{ 74 "Wang Span Network"}
{ 75 "Packet Video Protocol"}
{ 76 "Backroom SATNET Monitoring"}
{ 77 "SUN ND PROTOCOL"}
{ 78 "WIDEBAND Monitoring"}
```

```
{ 79 "WIDEBAND EXPAK"}
{ 80 "ISO IP" "ISO Internet Protocol"}
{ 81 "VMTP"}
{ 82 "SECURE-VMTP"}
{ 83 "VINES"}
{ 84 "TTP" }
{ 85 "NSFNET-IGP"}
{ 86 "Dissimilar Gateway Protocol"}
{ 87 "TCF"}
{ 88 "EIGRP"}
{ 89 "OSPF-IGP"}
{ 90 "Sprite RPC Protocol"}
{ 91 "Locus Address Resolution Protocol"}
{ 92 "Multicast Transport Protocol"}
{ 93 "AX.25 Frames"}
{ 94 "IP-within-IP Encapsulation Protocol"}
{ 95 "Mobile Internetworking Control Protocol"}
{ 96 "Semaphore Communications Sec. Protocol"}
{ 97 "Ethernet-within-IP Encapsulation"}
{ 98 "Encapsulation Header"}
{ 99 "Any private encryption scheme"}
{ 100 "GMTP"}
{ 101 "Ipsilon Flow Management Protocol"}
{ 102 "PNNI over IP"}
{ 103 "PIM" "Protocol Independent Multicast"}
{ 104 "ARIS"}
{ 105 "SCPS"}
{ 106 "ONX"}
{ 107 "Active Networks"}
{ 108 "IP Payload Compression Protocol"}
{ 109 "Sitara Networks Protocol"}
{ 110 "Compaq Peer Protocol"}
{ 111 "IPX in IP"}
{ 112 "Virtual Router Redundancy Protocol"}
{ 113 "PGM Reliable Transport Protocol"}
{ 114 "0-hop" "Any 0-hop protocol"}
{ 255 Reserved }
{ Default "???" "Protocol Not Found"}
ENDTABLE

/*
** This table lists codes for the IP options that may
** follow the main IP header. Note that this table
** contains names of statements for processing each
** option.
*/
TABLE ip_options
```

```
{ 0 "End of List"}
{ 1 "No Operation"}
{ 7 "" "Record Route" 1  misc_pointer}
{ 10 "" "Experimental Measurement" 1  misc_opt}
{ 11 "" "MTU Probe" 1  misc_opt}
{ 12 "" "MTU Reply" 1  misc_opt}
{ 15 "" ENCODE 1  misc_opt}
{ 68 "" "Time Stamp" 1  ts}
{ 82 "" Traceroute 1  misc_opt}
{ 130 "" Security 1  security }
{ 131 "" "Loose Source Route" 1  misc_pointer}
{ 133 "Extended Security"}
{ 134 "Commercial Security"}
{ 136 "Stream ID"}
{ 137 "Strict Source Route"}
{ 142 "Expermental Access Control"}
{ 144 "IMI Traffic Descriptor"}
{ 145 "EIP" }
{ 147 "Address Extension"}
{ 148 "" "Router Alert" 1  rtr_opt}
{ 149 "Selective Directed Broadcast"}
{ 150 "NSAP Addresses"}
{ 205 "Experimental Flow Control"}
{ Default "Unknown IP Option"}
ENDTABLE

/*
** This table lists security levels. This is used
** when processing the Security option.
*/
TABLE sec_TABLE
{ 0x00000001 "Reserved 4"}
{ 0x0000003d "Top Secret"}
{ 0x0000005a "Secret"}
{ 0x00000096 "Confidential"}
{ 0x00000066 "Reserved 3"}
{ 0x000000cc "Reserved 2"}
{ 0x000000ab Unclassified }
{ 0x000000f1 "Reserved 1"}
{ Default "Unknown Security Classification"}
ENDTABLE

/*
** This table is used in processing the Router Alert
** option.
*/
TABLE rtr_options
```

```
{ 0x00000000 "Examine all packets"}
{ Default "Reserved"}
ENDTABLE

/*
** The IP header begins with a four-bit field that contains
** the protocol version number. If we find a version below
** 4 then we will have a failed VERIFY resulting in the frame
** being marked in red.
*/
FIELD version (Fixed 4 Bits) (TABLE ver_nums) "Version" VERIFY (FieldIs
GreaterThanOrEqualTo 4)

/*
** The IHL field contains the IP Header Length as a count of
** 32-bit words. The base header consists of a fixed 5 such
** words, hence the check that this value is at least 5.
** Anything beyond 5 words comprises options.
*/
FIELD ihl (Fixed 4 Bits) (Decimal) IN_SUMMARY IHL 30 "Header Length" VERIFY (FieldIs
GreaterThanOrEqualTo 5 ) ALSO (IsNoMoreThanLayerInDwords)

/*
** Here we decode the Type Of Service byte. Note that this
** could have been done with a GROUP FIELD. For completeness
** we display the two reserved bits rather than covering them
** up with a RESERVED statement.
*/
GROUP tos "Type of Service"
{
  FIELD preced (Fixed 3 Bits) (Binary) ALSO (Table prec)
                 Precedence
  FIELD delay (Fixed 1 Bit) (Binary) ALSO (Table delay) Delay
  FIELD thru (Fixed 1 Bit) (Binary) ALSO (Table norm_hi)
                 Throughput
  FIELD reli (Fixed 1 Bit) (Binary) ALSO (Table norm_hi)
                 Reliability
  FIELD reserv (Fixed 2 Bits) (Binary) Reserved
}

/*
** This field contains the length of the total IP packet in
** bytes including the IP header. Thus this is the length of
** the IP layer plus all subsequent layers.
*/
FIELD tot_length (Fixed 2 Byte) (Decimal) IN_SUMMARY Length 50
        "Total Length"
```

```
/*
** The ID field contains the packet's serial number. This
** is used for reassembling fragmented packets.
*/
FIELD id (Fixed 2 Byte) (Hex) IN_SUMMARY ID 50 Identification

/*
** The Control Flags field contains one unused bit plus two
** bit flags:
**  may/don't fragment
**  last fragment/more fragments coming
*/
GROUP ctrl_flags "Control Flags"
{
   FIELD reser (Fixed 1 Bit) (Binary) Reserved VERIFY
           (FieldIs EqualTo 0 )
   FIELD df (Fixed 1 Bit) (Table df) DF
   FIELD mf (Fixed 1 Bit) (Table mf) MF
}

/*
** For a fragment, this field contains the offset of its
** starting point within the full packet.
*/
FIELD offset (Fixed 13 Bits) (Decimal) "Fragment Offset"

/*
** The Time-To-Live field contains the number of seconds
** before a router should discard the packet.
*/
FIELD ttl (Fixed 1) (Decimal) IN_SUMMARY Time 40
           "Time to live sec."

/*
** This field contains a code that identifies the next-layer
** protocol. Note that we verify that the code is present
** in our table of protocols.
*/
FIELD protocol (Fixed 1) (Table protocol) IN_SUMMARY Protocol 100
           Protocol VERIFY (IsInTable protocol)

/*
** Next comes a 16-bit checksum on the header so far, that is
** 10 bytes worth.
*/
FIELD chksum (Fixed 2 Byte) (Hex) IN_SUMMARY Checksum 65 "Header
```

```
          Checksum" VERIFY (IpChecksum ihl)

/*
** The source and destinations addresses of the packet are
** decoded here. These are formatted in dot-quad notation
** (e.g. "64.147.250.27"). Note that the addresses are stored
** in intra-frame (within the current frame only) data
** so that they can be used in the Summary pane for other protocols,
** such as HTTP.
*/
FIELD src_addr (Fixed 4 Byte) RETRIEVE (StoreIntraframeField source_ip_address)
(IPAddress) IN_SUMMARY Source 90
          "Source Address"

FIELD dest_addr (Fixed 4 Byte) RETRIEVE (StoreIntraframeField destination_ip_address)
(IPAddress) IN_SUMMARY Destination 90 "Destination Address"

/*
** Here we have a loop to process any IP options.
** The option_loop GROUP is repeated until all the header
** data is processed (i.e. the count of doublewords in IHL
** minus 5).
*/
GROUP option_loop IF (FieldIs GreaterThan 5 ihl) REPEAT SIZE (FromField DWords ihl 5)
{
    /*
    ** Here's a good use of a GROUP FIELD to decode and display
    ** the option type.
    */
    GROUP FIELD ip_opt_type (Fixed 1) (Table ip_options)
                    "IP Option"
    {
            /*
            ** Most options include a one-byte length field.
            ** The extra data in the options table indicates
            ** whether this should be present or not.
            */
            FIELD opt_length (Fixed 1) IF (MatchTableData
                        ip_options ip_opt_type 1) (Decimal)
                        "Option length"

            /*
            ** We branch to option-specific code
            ** for the rest of the job.
            */
            BRANCH (FromTable ip_options ip_opt_type)
    }
```

```
}

/*
** The following two fields deal with the possibility
** that the frame is fragmented.
*/

FIELD set_fragmentation (Fixed 0)
    PROCESSING (IpSetFragmentation a mf offset src_addr dest_addr decoder ID
set_fragmentation)
    (Decimal) SUPPRESS_DETAIL

FIELD consume_fragment (ToEndOfLayer) IF (IpWholePayloadNotAvailable a) (Constant
"Decode is in another frame") "Payload is fragmented"

/*
** Here ends the main path of this decoder. The statements
** that follow are for decoding specific IP options and
** we have not commented them.
*/
END_MAIN_PATH

GROUP misc_pointer
{
   FIELD miscellaneous_pointer (Fixed 1) (Decimal) Pointer
   FIELD miscellaneous_data (FromField Bytes opt_length 3)
          (StringOfHex 6) Data
}

FIELD misc_opt (FromField Bytes opt_length 2) (StringOfHex 6)
          Data

GROUP ts
{
   FIELD ts_point (Fixed 1) (Decimal) Pointer
   FIELD ts_flags (Fixed 1) (Binary) Flags
   FIELD ts_data (FromField Bytes opt_length 4) (StringOfHex 6)
          "TS Data"
}

GROUP security
{
   FIELD sec_class (Fixed 1) (Table sec_TABLE) Classification
   FIELD sec_flags (FromField Bytes opt_length 3)
          (StringOfHex 6) Flags
}
```

```
GROUP rtr_opt
{
   FIELD rtr_opt_data (Fixed 2) (Table rtr_options)
          "Router Option"
}
```

Chapter 12: A Formal Grammar for DecoderScript

Here we provide precise specifications for the syntax of decoder executable statements using a BNF-type grammar. We have simplified the grammar a little and, we trust, made it much easier to read by skipping definition of the blank space that separates items. The rule for such space is that any consecutive items must be separated by at least one unit of white space (a space character or an end of line) and may be separated by several.

You may want to print out this chapter and stick the hardcopy to your wall as a reference.

Some notes about the forms used to define the grammar:

> A parenthesized expression within a non-terminal is a comment. Thus, in “<(size)method>” the “size” is a note to tell you that a Size Method is required in the context but the syntax is defined simply by “<method>”.

A phrase such as “1*<x>” means a string of one or more <x>’s. Similarly, “*1-16<y>” means a sequence of between one and sixteen <y>’s.

An ellipsis means the preceding item may be repeated any number of times.

A name/string must be enclosed in double quotes if it contains spaces or characters other than letters, numbers, and underscores.

And here is the grammar:

```
decoder ::= <header> DECODE <table>... <statement>...

header ::= [<comment>] <name> <decoder_id> [<data_order>] [<comment>] [<header_item>]...

decoder_id ::= <number>

data_order ::= NETWORK_DATA_ORDER | HOST_DATA_ORDER

header_item ::= PREPROCESSING <(preprocessing)method> |
                POSTPROCESSING <(postprocessing)method> |
                NEXT_PROTOCOL <(next_protocol)method> |
                NEXT_PROTOCOL_OFFSET <(next_protocol_offset)method> |
                NEXT_PROTOCOL_SIZE <(next_protocol_size)method> |
                FRAME_BEFORE_DECODING <(frame_transformer)method> |
                FRAME_AFTER_DECODING <(frame_transformer)method> |
                RECOGNIZER <(recognizer)method> |
                PAYLOAD_IS_BYTE_STREAM |
                BYTE_STREAM_FRAMER <(ByteStreamFramer)Method> |
                SUMMARY_COLUMNS <column list> |
```

column list ::= "{" <summary_label> <column_width> ... "}"

summary_label ::= <string>

column_width ::= <number>

table ::= TABLE <name> <table_entry>... ENDTABLE |
TABLE <name> "@" <filename>

table_entry ::= "{" <match> [<match_hi>] <summary_string> [<detail_string>] [<user_data>] [<branch_field>] "}"

match ::= <number> | DEFAULT

match_hi ::= <number>

summary_string ::= <string>

detail_string ::= <string>

user_data ::= <number>

branch_field ::= <(field)name>

statement ::= <branch> | <field> | <group> | <group-field> | <reserved>

branch ::= BRANCH <(branch)method>

field ::= <field_definition> | <field_reference>

field_definition ::= FIELD <(field)name> [<startbit_clause>] <(size)method>
[<conditional_clause>] [<retrieve_clause>] [<processing_clause>] <format>
[<summary_tag_clause>] <(decode_pane)string> [<tag_clause>]
[<margin_text_clause>][<info>][<nomove>] [<data_order>] [<verify_clause>][<store-clause>]

summary_tag ::= IN_SUMMARY | IN_SUMMARY_ONLY | IN_COLLAPSED

field_reference ::= FIELD <(field)name> ;

group ::= GROUP <(group)name> [<if_clause>] [<repeat_clause>] [<(detail)string>] {
1*<statement>}

group-field :: = GROUP <field_definition>

<reserved> ::= RESERVED <(field)name> <(size)method>

atom_char ::= <letter> | <digit> | <underscore>
conditional_clause ::= <if_clause> | <print_if_clause>
decimal ::= [<minus>]1*<digit>
data_order ::= HOST_DATA_ORDER | NETWORK_DATA_ORDER
digit ::= 0 – 9
float ::= [<minus>]0*<digit>.1*<digit> | [<minus>]1*<digit>.0*<digit>
format ::= <(format)method> [ALSO <(format)method>]
hex ::= 1*<hexit>
hexit ::= <digit> | a – f, A - F
if_clause ::= IF [NOT] <(boolean)method> [ELSE_SKIP <(size)method>]
IN_SUMMARY_clause ::= IN_SUMMARY <(summary_column)string> <(column_width)number>
info ::= INFO
letter ::= a – z, A – Z
margin_text_clause ::= MARGIN_TEXT <(format)method>
method :: = (<(method)name>) | (<(method)name> *1-15<param>)
minus ::= -
name ::= 1*<atom_char>
name_char ::= <space> | <atom_char>
nomove ::= NO_MOVE
number ::= <decimal> | <hex> | <float>
param ::= <string> | <number> | <(field/variable)name> | <table_name>
print_if_clause ::= PRINT_IF [NOT] <(verify)method>
repeat_clause ::= REPEAT UNTIL <(boolean)method> | REPEAT COUNT <(size)method> | REPEAT
SIZE <(size)method> [TRUNCATED_FIELD_OK]
retrieve_clause ::= RETRIEVE <(retrieve)method> [ALSO <(retrieve)method>]…
processing_clause ::= PROCESSING <(processing)method> [ALSO <(processing)method>]…
space :: = space character
startbit_clause ::= START_BIT <(startbit)method>
store_clause ::= STORE <(variable>name> [<retrieve_clause>]
string ::= 1*<atom_char> | ”1*<name_char>”
tag_clause ::=TAG <(tag)method>
underscore ::= _
verify_clause ::= VERIFY | SUPPRESS_IF_VERIFIED [NOT] <(verify)method> | VERIFY
<(crc)method> [ALSO [NOT] <(verify)method>| ALSO <(crc)method>]

Appendix 1: Decoder Summary

Here we present a skeleton decoder that you may find useful as a base to work from:

```
/* Header comments */
"Protocol Name" [Decoder ID in Hex] [Data Order]

NEXT_PROTOCOL (NextProtocolMethod param)

DECODE

TABLE tablename
{ low_value [high_value] "summary string" "decode string"
        extradata field/groupname}
ENDTABLE

FIELD fieldname (fieldsize) (format method) "decode label"

FIELD fieldname (fieldsize) conditional_statement (format method)
        IN_SUMMARY ColumnLabel defaultWidth "decode label"
           VERIFY (verify method)

FIELD fieldname ;

GROUP name "label"
{
    FIELDS
}
GROUP name conditional_statement
{
    FIELDS
}

END_MAIN_PATH

<GROUP and FIELD definitions for stuff called above>
```

Appendix 2: Using the Protocol Analyzer with the Option for Creating Decoders

Reload Decoders

When writing a decoder, it is often very useful to be able to view the results of changes right away. To reload the decoders to determine the effect of decoder changes, load a capture file into the protocol analyzer. Change your decoder in a text editor and save your changes. In the protocol analyzer, open the Frame Display window and click the icon on the far right of the toolbar that looks like two arrows.

Building Methods

Your custom methods will be automatically built for you, provided you have Microsoft's Visual C++.NET version 7.1 installed on your system. Assuming that you do, all you need to do is start the protocol analyzer. It should detect that you have a method that needs to be built and ask you if you want to build it now.

You can also build your methods at any time by using the toolbar buttons. Start the protocol analyzer. If you are in live mode (i.e. you have the data capture icons on your toolbar), go to the File menu on the Control window and choose Close. Any open capture files will be closed, and the protocol analyzer will enter View Mode. You will find two icons in addition to the File-Open icon.

The icon with the red arrow, will recompile all methods that have changed. The second icon, will rebuild all methods for which there are source files. In the <installation path>\Executables\Core directory, a Method.Log will be created. This file has the inputs and outputs to every method called for every field in the file. You want to use this with a file of just a few frames because otherwise the log file gets very large and it's very difficult to match the log data against the capture file.

How to Insert Fictitious Protocol into a Fictitious Product

If you are writing decoders for industrial automation analyzers, you may want to add your decoder as a separate analyzer in the launcher. This way your users can simply click on the "Analyze My Custom Industrial Protocol" link instead of choosing a generic analyzer and setting the stack within the software themselves.

The example product is called "Fictitious Product". The example protocol is called "Fictitious Protocol". All references to file names and tags in files will use that name for convenience.

There are two files that must be modified so that an option to analyze data with the new decoder or stack will be presented by the launch application. Those files are "SelectorChoices.txt" and, in this case, "FP.personality". The decoder-writer will determine the name of the personality file. In general, to insert the new protocol, change "Fictitious Protocol"

or “FictitiousProtocol”, the placeholders in the examples below, to the name of the desired protocol. Spaces are not important in items that are for display only or in items that are used in determining the application command line. Items in *italics* are just display items, and are completely at the decoder-writer’s discretion. Items in **bold**-type **must** be consistent both inside a file and across the files. Note that angle brackets, ‘<’ and ‘>’, are used to indicate variable entries, and are not a part of the actual syntax.

The resulting file paths will be:

FP.personality will be put into My Decoders, and
SelectorChoices.txt will be put into <installation path>\App Data\Decoders.

You will need to include these files in your distribution package along with the method(s) and decoder(s).

The contents of file “FP.personality” are:

[Rules]
0x7f000404,0x7f000000,0x7f000000 TCP - Fictitious Protocol

[Filters]
Include Fictitious Protocol,Predefined\TCP IP\Include,,Include,port 9998 or port 9999

[Port Assignments:TCP]
9998-9999,0x7f000000

[Predefined Stacks]
Fictitious Protocol with autotraverse,Auto,0x7f000000,

[Common Personality]
CbSizeInKbytes=16398
FileSizeInKbytes=32793
WrapBuffer=FALSE
CompanionSizeMultiplier=3
StartupWindows=CVStaDisp
EnableNodeFilters=true

[Personality`Fictitious Product`**Ethernet`FictitiousProtocol**]
ShortcutName=Ethernet*A Fictitious Protocol*
SetupCommand=.\Fts /"Fictitious Product=**Ethernet**" /Setup /OEMKEY="**FictitiousProtocol**"
OemTitle=Fictitious Product Protocol Analyzer for *Fictitious Protocol*
Description=Double-click to analyze your *Fictitious Protocol* network, or click the [Go!] button.

The Rules, Filters, Port Assignments and Predefined Stacks sections are described in Chapter 6: [ProtocolName].personality files. The Common Personality section does not need to be included in your custom personality file. If no Common Personality section is found, the software will use the one found in the personality file for the product, which is as it should be.

You will need to change the Personality section. Line order within the section is not important, with the exception of the first line.

The first line is the personality section header, and it must be the first line. The format is "[Personality`<Product Name>`<Driver Key>`<OEM Key>]". In this example, the data are being collected from an Ethernet network, so we must set the Driver Key to "**Ethernet**". We have decided to use a tag "**FictitiousProtocol**" as the OEM Key (to differentiate from other Ethernet personalities). The decoder-writer will determine that key value, but it must be consistent within and across the files. You can find the appropriate values for the Product Name and Driver Key by looking in the personality file for your product.

The next line is the Shortcut name. The format is "ShortcutName=<Driver Group>\<*Protocol Display Name*>". The driver group will generally already be displayed in the selector, so to put the protocol in an existing group, use one of those names – in this case, "Ethernet" has been used. Another name could be used, but that will create a new group. The Protocol Display Name is at the discretion of the decoder-writer.

The next line is the SetupCommand. This is the command line that will be used to launch the setup mode of the protocol analyzer. The format is "SetupCommand=.\Fts /"<Product Name>=<**Driver Key**>" /Setup /OEMKEY="<**OEM Key**>".

The next line is the OemTitle. This is the title that will be displayed on the Control Window of the protocol analyzer. The format is "OemTitle=<*Decoder-writer's choice of title*>".

The next, and in this case last, line is Description. This is the description of the action that the "Go" button will perform in the selector application. The format is "Description=<*Decoder-writer's choice of description*>".

The file "SelectorChoices.txt" also need to be modified. This is the file that gives the names listed in the launcher window. The sample contents are:

`Generic`DeviceNet
`Async`DF1_Half_BCC
`Ethernet`CSP
`View
`Async`DF1_Half_CRC
`Async`DH485
`Ethernet`Ethernet_IP
`Async`DF1_Full_BCC
`ComProbeClassic
`Async
`Generic`CAN2A
`Async`DF1_Full_CRC
`Async`DH+
`Ethernet
`Ethernet`FictitiousProtocol

The last line, "**`Ethernet`FicititiousProtocol**", has been added to validate the new analyzer personality, which was created with the personality file. The format is "`<**Driver Key**>`<**OEM Key**>".

Pay careful attention to the leading character in the personality section of the personality file, which is also the leading character of the entry in selectorchoices.txt. This character is used as a separator, and is a grave accent, and is found on the upper-left key on a keyboard, co-located with the '~'.

Appendix 3: Keyword List

ALSO	Chains together RETRIEVE, FORMAT or VERIFY statements
ALWAYS_ASK	Used in conjunction with the PARAMETER keyword. Forces the user to enter the parameter every time the decoder is loaded.
ASK_IF_NOT_DEFINED	Used in conjunction with the PARAMETER keyword. Asks the user for the parameter the first time the decoder is loaded, and doesn't ask again after that.
BRANCH	Causes control to pass to one of several places depending on the results of the Branch method.
BYTE_STREAM_FRAMER	Reference section keyword followed by a byte stream framer method. Works in a similar fashion to a frame recognizer, except that it operates only on the data found at the current layer.
COUNT	Used with REPEAT clauses to repeat the group X number of times.
DECODE	Begins the decoder.
DEFAULT	Specifies the default entry for tables.
ELSE_SKIP	Used with a Size method to skip a specified amount of data if a conditional fails. If the ELSE_SKIP is not present, no data is skipped.
END_MAIN_PATH	Ends the decoder. At this statement decoding of this layer stops.
ENDTABLE	Specifies the end of a table.
FIELD	The main component of the decoder.
FRAME_AFTER_DECODING	Reference section keyword followed by a frame transformer method that processes the frame after it has been decoded.

FRAME_BEFORE_DECODING	Reference section keyword followed by a frame transformer method that processes the frame before it is decoded.
GROUP	Used to group together FIELDs for a variety of reasons. Groups can be the destination of a branch, they can have conditionals placed on them or REPEAT clauses such that the fields in the group are executed multiple times.
HOST_DATA_ORDER	Can be placed either after the decoder ID at the top of the decoder, in which case all fields are interpreted in host data order, or at the end of a field statement, in which case just that field is interpreted in host data order.
IF	Conditional. Used with Boolean methods.
IN_COLLAPSED	Places the results of the field just in the summary line of a collapsed protocol in the Protocol Navigator.
IN_SUMMARY	Places the results of the field in the Summary pane on the Frame Display and in the summary line of a collapsed protocol in the Protocol Navigator.
IN_SUMMARY_ONLY	Places the results of the field in just the Summary pane on the Frame Display
INCLUDE	Inserts the named file at this location in the decoder.
INFO	Causes the field output to be displayed in an Information line at the top of the Decode pane in addition to its normal location in the decode.
LIST	Defines a list of legal values that a parameter can take. Used in conjunction with the PARAMETER keyword.

MARGIN_TEXT	Causes the field output to be appended to the margin text in the Protocol Navigator.
NETWORK_DATA_ORDER	Can be placed either after the decoder ID at the top of the decoder, in which case all fields are interpreted in network data order, or at the end of a field statement, in which case just that field is interpreted in network data order.
NEVER_ASK	Used in conjunction with the PARAMETER keyword. Never asks the user for the parameter value. Useful when the parameter has a good default value that may need to be changed only rarely.
NEXT_PROTOCOL	Reference section keyword followed by a Next Protocol method. Determines the next protocol in the stack.
NEXT_PROTOCOL_OFFSET	Reference section keyword followed by a Next Protocol Offset method. Determines the number of bytes between the end of the current layer and the beginning of the next.
NEXT_PROTOCOL_SIZE	Reference section keyword followed by a Next Protocol Size method. Determines the size of the next layer in bytes.
NO_MOVE	Tells the protocol analyzer not to move the bit pointer after decoding the field.
NOT	Reverses the results of a Boolean or Verify method. Often used in IF clauses.
PARAMETER	Defines a runtime parameter for the decoder. Allows the user to give information to the decoder that can be used for decision making or for field definitions.

PAYLOAD_IS_BYTE_STREAM	Reference section keyword. Tells the next protocol in the stack to treat the payload of all subsequent frames as a continuous byte stream. Used in conjunction with the BYTE_STREAM_FRAMER keyword.
POSTPROCESSING	Reference section keyword followed by a method that does things after the layer is decoded. Note that these methods cannot change the data in the frame.
PREPROCESSING	Reference section keyword followed by a method that does things before the layer is decoded. Note that these methods cannot change the data in the frame.
PRINT_IF	Followed by a Boolean method. Prints the value of the field in the Decode pane only if the method returns true.
RECOGNIZER	Reference section keyword. Identifier the frame recognizer for the protocol.
REPEAT	Followed by a SIZE, COUNT or UNTIL keyword. Used with groups to cause the group to be repeated.
RESERVED	Used with Size methods to skip a specified amount of data.
RETRIEVE	Used with Retrieve methods, which usually manipulate the contents of the field before displaying it.
SIZE	Followed by a Size method. Used with the REPEAT keyword on groups to repeat the group until the number of bits specified in the Size method have been used up.
START_BIT	Followed by a Start bit method to alter the location of the bit pointer.

STORE	Followed by a name. Stores the field contents under this name so other methods can reference it.
SUMMARY_COLUMNS	Followed by a list of Summary Pane Column names and their default widths. Allows you to define the order of the Summary Pane entries.
SUPPRESS_DETAIL	Prevents the field from being displayed in the Decode pane.
SUPPRESS_IF_VERIFIED	Prevents the field from being displayed in the Decode pane only if the Verify passes. The field is shown if the Verify fails.
TABLE	Begins a table.
TAG	Followed by a Tag method. Used to tag a field for filtering, tracking and other purposes.
UNTIL	Followed by a Boolean method. Used with the REPEAT keyword on groups to repeat the group until the Boolean method returns true.
	Followed by a Verify method. Usually used to determine if the contents of a field are valid.

Appendix 4: Creating Custom Frame Recognizer Return Values

The recognizer returns are 32-bit bit masks that represent which events to add and when and where to add them. We've created enumerations representing the most common returns we think will be needed, but you may need to create your own if an existing enumeration does not meet your needs. (Existing enumerations are listed in the chapter on writing Frame Recognizers.)

We've created some basic enumerations/macros so you don't need to know the exact format of the bit mask. This way if the format ever changes, your recognizer will still work.

To add framing events, use one or more of the following recognizer result enumerations:

ThisSideBefore(Recognizer Event)
ThisSideAfter(Recognizer Event)
OtherSide(Recognizer Event)

Possible Recognizer Events are:

StartOfFrame
EndOfFrame
BrokenFrame

You can specify more than one recognizer event per recognizer result by using the OR operator, with one exception. You cannot use both a BrokenFrame and EndOfFrame in the same recognizer return. A broken frame event means the frame didn't end when the recognizer expected it to. The broken frame is inserted to let the end-user know that something went wrong and end the current frame so the recognizer can start over with trying to find the beginning of the next frame.

For example, to insert a start of frame event before the current byte (the equivalent of the eInsertSofBefore enumeration), you would use the syntax:

```
iOutput = (eRecognizerResult)(ThisSideBefore(StartOfFrame));
```

To frame a single byte as one frame, you would use:

```
iOutput = (eRecognizerResult)(ThisSideBefore(StartOfFrame)|ThisSideAfter(EndOfFrame));
```

To end the current frame and begin a new one on the same side (the equivalent of eNewFrameEndThisSide), you would use:

```
iOutput = (eRecognizerResult)(ThisSideBefore(StartOfFrame|EndOfFrame));
```

How does the recognizer know that the end of frame event needs to be inserted in the buffer before the start of frame event? The recognizer events are added to the capture buffer in the following order:

ThisSideBefore
    BrokenFrame OR EndOfFrame (these are mutually exclusive)
    StartOfFrame

Data Event

ThisSideAfter
    BrokenFrame OR EndOfFrame
    StartOfFrame

OtherSide
    BrokenFrame OR EndOfFrame
    StartOfFrame

Appendix 5: Method Reference

Group: BOOLEAN

Method: Check_CrcCCITTrevDataBytes

Description:
The Check_CrcCCITTrevDataBytes method calculates the CCITT checksum after reversing the data bytes. It then compares the result with the value in the "Field with CRC". It returns TRUE if the values match, else it returns FALSE.

Parameters:

[Required]	Int	"First byte to include (0=first byte in layer)"
[Required]	Int	"Seed"
[Optional]	List	"Swap Result" (Default value: NoSwap) List values: NoSwap, Swap
[Required]	Field	"Field with CRC"

Example:
FIELD bad_crc_packet (ToEndOfLayer) IF NOT (Check_CrcCCITTrevDataBytes 0 0xFFFF NoSwap crc_field) (Hex) "Bad Packet"

Method: CompareFields

Description:
CompareFields compares two fields and returns TRUE if the relationship is true.

Parameters:

[Required]	Field	"First field to compare"
[Required]	List	"Operator" List values: GreaterThan, GreaterThanOrEqualTo, LessThan, LessThanOrEqualTo, EqualTo, NotEqualTo
[Required]	Field	"Second field to compare"

Example:
GROUP doifconditiontrue IF (CompareFields field1 EqualTo field2)

Method: CurrentDataMatches

Description:
This method returns TRUE if the current data has the value of the "Value to match" parameter. This allows you to check the value of data at the pointer without explicitly decoding it with a FIELD statement. It requires two parameters: 1) the number of bits (1 to 64) to be extracted from the data for comparison and 2) the value to be matched against.

This method is particularly useful in two situations. The first is in a GROUP statement to check a field before decoding it. You might, for example, have a field that is defined as being a four-byte field if the first byte contains hex FF and a two-byte field otherwise. You can see examples in the SMB decoder.

The second case is where you want to decode a field only if it contains a certain value. Consider, for example, this pair of statements from the NCP decoder:

```
FIELD printer_off_line (Fixed 1) IF (CurrentDataMatches 8 0xff) (Constant "Halted")
"Printer Status"
FIELD printer_on_line (Fixed 1) IF (CurrentDataMatches 8 0x00) (Constant "Not Halted")
"Printer Status"
```

Here a one-byte field is deemed to exist and is decoded only when its value is zero or hex FF.

Parameters:

[Required]	Int	"Number of bits to match (from 1-64)"
[Required]	Int64	"Value to match"

Example:
```
GROUP string_bindings REPEAT UNTIL (CurrentDataMatches 8 0x01)
```

Method: Dce

Description:
This method returns TRUE if the frame comes from the DCE side. This method has relevance only for data received from a serial type connection.

Parameters:
None

Example:
```
FIELD streamDCE (Fixed 4) IF (DCE) (StringOfHex) “DCE DATA”
```

Method: Dte

Description:
Dte returns TRUE if the frame comes from the DTE side. This method has relevance only for data received from a serial type connection.

Parameters:
None

Example:
FIELD dte_stream (Fixed 4) IF (DTE) (StringOfHex) “DTE Data”

Method: FieldIs

Description:
This method returns TRUE if the statement is true. The statement compares the value in "Field to test" against a specified value using the given operator.

Parameters:

[Required]	List	"Operator"	
		List values:	GreaterThan
			GreaterThanOrEqualTo
			LessThan
			LessThanOrEqualTo
			EqualTo
			NotEqualTo
[Required]	Int64	"Value"	
[Required]	Field	"Field to test"	

Example:
Here’s a typical example taken from the X.25 decoder (example has been edited):
FIELD called_addr(Fixed2) IF (Fieldls GreaterThan called_addr_lgh 0)(StringOfBcd) “Called Address”
In this case a field exists only when its length(called_addr_lgh) is non-zero.

Method: FieldIsBetween

Description:
FieldIsBetween returns TRUE if the value in the "Field to test" is between the specified low and high values. This test is inclusive. In other words, if the value in the "Field to test" is equal to either the low or high value, the method will return true.

Parameters:

[Required]	Int64	"Low value"
[Required]	Int64	"High value"
[Required]	Field	"Field to test"

Example:
GROUP if_not_sco_pkt IF NOT(FieldIsBetween 5 8 packet_type)

Method: FieldLengthIs

Description:
This method returns TRUE if the statement is true. The statement compares the value in "Field containing length value" against the length of the current field using the given operator.

Parameters:

[Required]	List	"Operator"		
			List values:	GreaterThan
				GreaterThanOrEqualTo
				LessThan
				LessThanOrEqualTo
				EqualTo
				NotEqualTo
[Required]	Field	"Field containing length value"		
[Required]	Field	"Field whose length to test"		

Example:
FIELD segment (FromField Bytes LOS) IF (FieldLengthIs GreaterThanOrEqualTo LOS get_segment_length) (StringOfHex) "Segment"

Method: IndexedPersistentStaticExists

Description:
This method checks to see if the specified inter-frame data exists, which is true only if it has been encountered in any decoder in any frame so far. In the case of the IndexedPersistentField, it checks to see if the index value is good. It returns TRUE if the value is good and FALSE if the value isn't there. This is useful in cases where you have saved a field value with an index, but the index field hasn't been encountered yet. Using the example from the StoreIndexedPersistentField method, this method would check to see if a command to the slave specified in the parameter has been seen. If not, you can do something appropriate.

Parameters:

[Required]	StaticName	"Inter-frame data name"
[Required]	Field	"Field that contains the index"

Example:
FIELD retrieve_type (Fixed 0) IF (IndexedPersistentStaticExists usb_type_with_address frame_address) RETRIEVE (IndexedPersistentField usb_type_with_address frame_address) (TABLE types) SUPPRESS_DETAIL

Method: IntraframeStaticExists

Description:
This method checks to see if the specified intra-frame data exists, which is true only if it has been encountered in any decoder so far in the current frame. It returns true if the value is good, and false if the value isn't there. This is useful in cases where you have saved a field value, but that field may not be encountered in every frame. You can use this method in conjunction with the IntraframeField method to see if the data exists and do something appropriate if it doesn't, like skip the field, or decode something else instead.
The method doesn't take a parameter, but you do need to specify which field value you're checking for.

Parameters:

[Required]	StaticName	”Intra-frame data name”

Example:
This example uses the boolean to skip the field if the intra-frame data doesn't exist.
FIELD my_field_2 (Fixed 0) IF (IntraframeStaticExists field_name) RETRIEVE (IntraframeField field_name) (Decimal) "Field Name"

Method: IsInTable

Description:
This method returns TRUE if the value of the current field exists in the specified table and is not the default.

Parameters:

[Required]	Table	"Table"
[Required]	Field	"Field"

Example:
GROUP valid_types IF (IsInTable point_types is_valid_type)

Method: IsOdd

Description:
This method returns TRUE if the field passed in is an odd number

Parameters:

[Required] Field "Field to test"

Example:

GROUP end_of_path_test IF (IsOdd symbolic_length)

Method: IsProbablyAtType

Description:

This method returns TRUE if the frame is probably an AT command. The criteria is: If the frame ends in <CR>, and the last two bytes are not a legal FCS, and all characters in the frame are printable ASCII, then it is an AT frame.

Parameters:

None

Example:

GROUP decode_at_command IF (IsProbablyAtType)

Method: MatchTableData

Description:

This method returns TRUE if the table data matches the parameter "Value to match". A typical use of this method is to distinguish various classes of message, for example commands, responses and data.

Parameters:

[Required] Table "Which table"
[Required] Field "Which field"
[Required] Int64 "Value to match"

Example:

Examples of its use can be seen in the HTTP and SMTP decoders. One instance from the latter decoder is (the table has been edited for clarity):

```
TABLE cmd_reply

{0x48454c4f "HELO" "Hello" 1 hi}
{0x4d41949c "MAIL" "Mail" 1 from}
{0x51554954 "QUIT" "Quit" 1 end}
{0x32313120 "Status" System status, or system help reply" 2}
{0x32313420 "Help" "Help message" 2}
{Default ""default"3}
```

ENDTABLE

FIELD type (Fixed 4) (Hex) SUPPRESS_DETAIL NO_MOVE

Group cmd IF (MatchTableData cmd_reply type 1)

This takes the value in the field “type” and looks it up in the table “cmd_reply”. Then the table data for that entry is checked against the constant given in the parameter “Value to match”. For example if the value of “type” were 0x48454c4f. the test would return true because this value matched the “HELO” entry in the table, and the table data for “HELO” is 1, which matches the 1 given as the parameter.

Method: MoreBytes

Description:
This method returns TRUE if there are more bytes to be decoded in the layer, and false if the current bit is at the end of the layer. If the current pointer is not at a byte boundary it is moved up to the next byte boundary for the purposes of the calculation. The parameter "Offset from end in bytes" specifies how many bytes at the end of the layer to disregard when determining if there are more bytes remaining. This is useful if you want to ignore a CRC, for example.
This method is often used to test for optional fields at the end of a layer.

Parameters:

[Optional]	Int	"Offset from end in bytes"	(Default value: 0)

Example:
GROUP more_ieee_format_data IF (MoreBytes)

Method: MoreBytesInSegment

Description:
This method figures out how much data has been consumed between the start of the field in the "Field that contains length" parameter and the current bit position. Then it compares the value in the "Field that contains length" parameter with the amount of data consumed and returns TRUE if there is more data to decode and FALSE if not.

Note that this method is inclusive, which means if you need to find out how much data has been consumed not including the data in the first parameter field, you will need to include the size of that field as an offset. See Size method RelativeFromField for one use of this method.

Parameters:

[Required]	Field	"Field that contains length"	
[Optional]	Int	"Size of Parameter Field In Bytes"	(Default value: 0)

Example:
FIELD consume_data (RelativeFromField length 1) IF (MoreBytesInSegment length 1) (StringOfHex 64) "Data"

Method: MoreBytesToDecode

Description:
This method returns TRUE if there are more bytes to decode based on the length field and FALSE if it does not have any bytes to decode.

Parameters:

[Required]	Field	"Field that contains length"	
[Optional]	Int	"Size of Parameter Field in Bytes"	(Default value: 0)

Example:
FIELD unknown_or_undecoded_bytes (RelativeFromField length 2) IF (MoreBytesToDecode length 2)(StringOfHex) "Unknown Bytes"

Method: PersistentStaticExists

Description:
This method checks to see if the specified inter-frame data exists, which is true only if it has been encountered in any decoder in any frame so far. It returns true if the value is good, and false if the value isn't there. You can use this method in conjunction with the PersistentField method to see if the data exists and do something appropriate if it doesn't, like skip the field, or decode something else instead.

Parameters:
[Required] StaticName ”Inter-frame data name”

Examples:
FIELD retrieved_addr_error (Fixed 0) IF NOT (IndexedPersistentStaticExists IR srcslave)

Method: TestTableData

Description:
This method returns TRUE if the table data is anything other than zero, and false if the table data is zero or doesn't exist.

Parameters:

[Required]	Table	"Which table"
[Required]	Field	"Which field"

Example:
Here is an example from the TCP decoder:

```
TABLE tcp_options
{0 “” “End of Option List” }
{2 “” “Maximum Segment Size” 1 seg_size}
{6 “” “Echo” 1 misc_opt}
{8 “” “Time Stamp” 1 timestamp}
{Default “” "Unknown TCP option” }
ENDTABLE

GROUP FIELD tcp_opt_type (Fixed 1) (Table tcp_options) "TCP Option"

FIELD opt_length (Fixed 1) IF (TestTableData tcp_options tcp_opt_type) (Decimal)
"Length"

BRANCH (FromTable tcp_options tcp_opt_type)
```

This sequence deals with TCP options. The field "tcp_opt_type" identifies an option. Most options, but not all, are followed by parameters starting with a one-byte length field. The extra table data in the option table is either 1, meaning that a length field follows, or doesn't exist, meaning that nothing follows. The value of the field "tcp_opt_type" is looked up in the "tcp_options" table. Then the extra table data is checked for that entry. If table data exists, the Field "opt_length" is processed. If no table data exists, the field is not processed.

Group: Branch

Method: DTEorDCE

Description:
This method chooses one of two branch targets depending on whether the current frame came from the DTE or the DCE. Clearly this has meaning only in contexts where there are two data streams distinguished as DTE and DCE. The first parameter names the statement to be branched to for a DTE frame and the second names the target for a DCE frame.

Parameters:

[Required]	Field	"Field present if this frame is DTE"
[Required]	Field	"Field present if this frame is DCE"

Example:
In the X.25 protocol, everything is defined in terms of DCE and DTE and so near the toe of the X.25 decoder is the statement:

BRANCH (DTEorDCE dte_pkt dce_pkt)

This says: If the frame came from the DTE side, branch to the “dte_pkt” field, and if the frame came from the DCE side, branch to the “dce_pkt” field.

Method: FromTable

Description:
FromTable causes a branch to the statement named in the branch item of a table entry. This is very commonly used and examples can be seen in many decoders. As in all similar cases, the field in question must have been previously decoded.

If the matching table entry has no branch item then no branch is made and control continues at the next statement in sequence. Omitting branch targets from tables is considered perfectly good practice when it is appropriate. If there is no matching entry in the table then the Default entry, if any, is used.

Parameters:

[Required]	Table	"Which table"
[Required]	Field	"Which field"

Example:
BRANCH (FromTable type type)

Method: PersistentStaticExists

Description:
This method chooses one of two branch targets depending on whether the specified inter-frame data exists. The first branch is taken if it does exist, and the second is taken if it doesn't.

Parameters:

[Required]	StaticName	”Inter-frame data name”
[Required]	Field	"Field present if the data exists"
[Required]	Field	"Field present if the data doesn't exist"

Example:
GROUP dce_check_context IF (PersistentStaticExists HostSide)

Group: FORMAT

Method: Binary

Description:
This method formats the current field value in binary notation. The method takes 3 optional parameters. If you want to set only the 2nd or 3rd parameter, you must include values for previous parameters as placeholders.

The "Separator" parameter allows you to specify a character to separate multi-byte fields. The default value of "0" will cause multi-byte fields to run together with no separator. To put a space between each byte, use (Binary " ").

The "Amount of Data to Display" parameter determines the maximum amount of data to display in the Decode pane. A value of 0 will display all data in the field, regardless of length. This parameter is used for limiting the amount of data displayed in the Decode pane for very large fields or fields where the size is unknown ahead of time.

The "Suppress leading zeroes" parameter tells the method whether or not to suppress leading zeroes. The default is "KeepPad", which means all digits in the number will be displayed, while "SuppressPad" causes leading zeroes to be converted to blanks.

FIELD flag_byte (Fixed 8 Bits) (Binary) "Flags"
output: "Flags: 00000101"

FIELD flag_byte (Fixed 8 Bits) (Binary SuppressPad) "Flags" output: "Flags: 101"

Parameters:

[Optional]	Str	"Separator"	(Default value: 0)
[Optional]	Int	"Amount of data to display (0=unlimited)" (Default value: 0)	
[Optional]	List	"Suppress leading zeroes?" (Default value: KeepPad)	
		List values:	KeepPad
			SuppressPad

Example:
FIELD multicast_test START_BIT (Move 7 Bits) (Fixed 1 Bit) (Binary) SUPPRESS_DETAIL NO_MOVE

Method: Constant

Description:
The method Constant is an exception among format methods in that it does not actually format anything or even make use of the field value. Rather it passes on for display a string passed as a parameter.

One time Constant is handy is when you want to add a units (or other) qualifier to a number as in:

FIELD port_speed (Fixed 2 Bytes) (Decimal) ALSO (Constant "bps")

One more valuable use for Constant is when we want to note if a particular field is present or not without needing to display its value. This typically fits when a PRINT_IFclause is used as in:

FIELD archive (Fixed 1 Bit) PRINT_IF (FieldIs EqualTo 1) (Constant "Archive") "Attribute"

FIELD hidden (Fixed 1 Bit) PRINT_IF (FieldIs EqualTo 1) (Constant "Hidden") "Attribute"

Parameters:

[Required] Str "String to print"

Example:
FIELD command (Fixed 2) (Constant "AT Command") IN_SUMMARY "Address/Control

---

Method: Decimal

Description:
This method formats the field value as a signed decimal number. The field must be 64 bits or less in size. An error will be displayed if Decimal is used on fields greater than 64 bits in size.

Field values are inherently unsigned, so if there is a possibility that the field could have a negative value, you should use "RETRIEVE (SignExtension)" in addition to the Decimal format method. You can use this method to format a field value as an unsigned decimal number by not doing the RETRIEVE.

Parameters:

None

Example:
FIELD signed_decimal (Fixed 2 Bytes) RETRIEVE (SignExtension 16) (Decimal) “Signed Decimal”

FIELD unsigned_decimal (Fixed 2 Bytes) (Decimal) “Unsigned Decimal”

For the value of 0xCAFE, signed _decimal displays- 13570 while unsigned_decimal displays 51966.

Method: DecimalWithFieldLabel

Description:
This method adds a value from another field to the format, like this: "other_field = this_field". The two fields are output in decimal.

Parameters:

[Required] Field "Other field to format with"

Example:

FIELD state_bit_w_cnt (Fixed 1 bit) (DecimalWithFieldLabel counter) "State"

Method: DosDate

Description:

This method formats a 16-bit field value interpreted as a date expressed in DOS directory-entry format. An error will be displayed if DosDate is used with fields that are not 16 bits in size.

A DosDate is composed as follows:

Bits	Contents
0 - 4	Day of month (1-31)
5 - 8	Month (1-12)
9 - 15	Year relative to 1980

Parameters:

None

Example:

FIELD creation_date (Fixed 2) (DosDate) "Creation Date

Method: DosTime

Description:

This method formats a 16-bit field value as a time expressed in DOS directory-entry format. An error will be displayed if DosTime is used with fields that are not 16 bits in size.

A DosTime is made up as:

Bits	Contents
0 - 4	Number of two-second increments (0-29)
5 - 10	Minutes (0-59)
11 - 15	Hours (0-23)

Parameters:

None

Example:

FIELD update_time (Fixed 2) (DosTime) "Update Time

Method: Double

Description:
This method formats the current field value as an IEEE-standard double-precision (64 bits or 8 bytes) floating-point number. Such a number is comprised of the following bits: 1 for sign, 11 for the exponent, and 52 for the mantissa. The optional parameter specifies the number of decimal places to be formatted with a default of two. An attempt to use Double with a value that has been sized at other than 64 bits will result in an error.

Parameters:
[Optional] Int "Number of places after the decimal point" (Default value: 2)

Example:
FIELD lreal (Fixed 8) (Double) "LREAL" HOST_DATA_ORDER

Method: Float

Description:
This method formats the current field value as an IEEE-standard single-precision (32 bits or 4 bytes) floating-point number. Such a number is comprised of the following bits: 1 for sign, 8 for the exponent, and 23 for the mantissa. The optional parameter specifies the number of decimal places to be formatted with a default of two. An attempt to use Float with a value that has been sized at other than 32 bits will result in an error.

Parameters:
[Optional] Int "Number of places after the decimal point" (Default value: 2)

Example:
FIELD float (Fixed 4) (Float) "Value"

Method: Hex

Description:
This method formats the current field value as a hexadecimal number. The method takes 3 optional parameters. If you want to set only the 2nd or 3rd parameter, you must include values for previous parameters as placeholders. See the description of the Binary method for information on the parameters.

The result of a Hex format is preceeded by "0x" to indicate hex notation.

Parameters:

[Optional] Str "Separator" (Default value: 0)
[Optional] Int "Amount of data to display (0=unlimited)" (Default value: 0)
[Optional] List "Suppress leading zeroes?" (Default value: KeepPad)
List values: KeepPad
SuppressPad

Example:
FIELD id_code (Fixed 1 Byte) (Hex) “Id Code”

Method: IpAddress

Description:
This method formats the current field value as an IP address. The field must be exactly 4 bytes in size. An error is displayed if IpAddress is used with fields that are not 4 bytes in size.
The display format is 123.123.123.123, where each number is a decimal number.

Parameters:
None

Example:
FIELD ipaddress_4_bytes_all_0 (Fixed 4 bytes) (IpAddress) "4 Byte 00..."

Method: MacAddress

Description:
This method formats a 6 byte field as a MAC Address. An error will be displayed if MacAddress is used with fields that are not 6 bytes in size. The format is 00:01:02:03:04:05, where each digit is a hex digit. Note that this method honors the host/network data order setting.

The "OUI table" parameter specifies a table with OUI identifiers in it. If an identifier is found, it will be used in place of the first 3 digits. This means that if you use the MacAddress method, you must have an OUI table specified in the decoder.

Parameters:
[Required] Table "OUI table"

Example:
FIELD dest_mac_address (Fixed 6 Bytes) (MacAddress oui) "Destination Address"

Method: MillisecondsFrom1970

Description:

This method interprets the field value as the number of milliseconds since 1970. The field/1000 must be 32 bits in size or less. An error will be displayed if this method is used with fields/1000 greater than 32 bits in size.

Parameters:

None

Example:

FIELD time (Fixed 6) (MillisecondsFrom1970) "Time"

Method: NetBiosName

Description:

This method formats the current field value as a NetBios name. NetBios names are formed by taking a hex byte and converting it to the equivalent ASCII character. Then take the hex value of the ASCII character, and subtract hex 41 (ASCII 'A'). This number forms the first nibble of the actual hex value. Repeat. Take the resulting hex value and interpret it as an ASCII character.

Parameters:

None

Example:

FIELD node_name (Fixed 4) (NetBiosName) "Net bios name"

Method: Octal

Description:

Octal method formats the current field value as an octal number. The method takes 3 optional parameters. If you want to set only the 2nd or 3rd parameter, you must include values for previous parameters as placeholders. See the description of the Binary method for information on the parameters.

No indication is included in the formatted number that it is expressed in octal. When it is not clear from the context that octal is being used you may want to note it in the same sort of way as in the example given for the Hex method.

Parameters:

[Optional] Str "Separator" (Default value: 0)
[Optional] Int "Amount of data to display (0=unlimited) (Default value: 0)
[Optional] List "Suppress leading zeroes?" (Default value: KeepPad)
List values: KeepPad
SuppressPad

Example:

FIELD successor (Fixed 2) (Octal 0 0 SuppressPad) “Successor”

Method: OtherField

Description:
This method takes a field as input and formats it. Normally, format methods only format the current field, but this one actually formats a different one.

It can be used to display an enumerator field.

Parameters:

[Required]	Field	"Field to display"		
[Optional]	List	"Format"	(Default value: Decimal)	
			List values:	Decimal
				Hex

Example:
FIELD channel_map_bits (Fixed 2 bits) IF NOT (FieldIs EqualTo 79 map_counter) (OtherField map_counter Decimal) "Channel "

Method: Scale

Description:
This method multiplies the field by the scale factor, and formats as a decimal using the "digits after the decimal point" parameter. This method is useful because it multiplies the field value by a float, rather than an integer.

Parameters:

[Required]	Float	"Factor"	
[Required]	Int	"Digits after the decimal point"	(Default value: 2)

Example:
FIELD data_rate (Fixed 1) (Scale .5 3) "Data Rate"

Method: SecondsFrom1970

Description:
This method interprets the field value as the number of seconds since 1970. The field must be 32 bits in size or less. An error will be displayed if this method is used with fields greater than 32 bits in size.

Parameters:

None

Example:
FIELD time_4Byte (Fixed 4) (SecondsFrom1970) "Time"

Method: StringOfAscii

Description:
This method formats the current field as a string of ASCII characters. Non-printable characters are displayed as dots.

The optional parameter indicates how many characters to display in the Decode pane of the Frame Display window. Setting a value here does not affect the actual size of the field, merely prevents very long character strings from being displayed. If there is more data in the field when the parameter value is reached, an ellipsis (...) is displayed in the Decode pane. The default value of zero means that the entire contents of the field will be shown, regardless of length.

Parameters:
[Optional] Int "Amount of data to display (0=unlimited)" (Default value: 0)

Example:
FIELD payload_data (ToEndOfLayer) (StringOfAscii) “Payload”

Method: StringOfBCD

Description:
This method formats the current field as a string of binary-coded decimal. See the description for the StringOfASCII Method for information on the "Amount of data to display" parameter.

Parameters:
[Optional] Int "Amount of data to display (0=unlimited)" (Default value: 0)

Example:
FIELD release_version (Fixed 8) (StringOfBCD) "Release version"

Method: StringOfBinary

Description:
This method formats the current field as a string of binary digits. See the description for the StringOfASCII Method for information on the "Amount of data to display" parameter.

The “Separator” parameter allows you to specify a string to separate multi-byte fields. The default value of “0” will cause each field to be separated by a space.

Parameters:
[Optional] Int "Amount of data to display (0=unlimited)" (Default value: 0)
[Optional] Str “Separator” (Default value: 0)

Example:
FIELD onoff_flags (Fixed 1) (StringOfBinary) "Flags"

Method: StringOfDecimal

Description:
This method formats the current field as a string of decimal numbers. Each hex byte is formatted in decimal, rather than interpreting all the bytes in the field as a single number. See the description for the StringOfASCII Method for information on the "Amount of data to display" parameter.

The “Separator” parameter allows you to specify a string to separate multi-byte fields. The default value of “0” will cause each field to be separated by a space.

Parameters:

[Optional] Int "Amount of data to display (0=unlimited)" (Default value: 0)
[Optional] Str “Separator” (Default value: 0)

Example:
FIELD events (Fixed 4) (StringOfDecimal) "Events"

Method: StringOfHex

Description:
This method formats the current field as a string of hexadecimal numbers, with a space between each byte. See the description for the StringOfASCII Method for information on the "Amount of data to display" parameter.

The “Separator” parameter allows you to specify a string to separate multi-byte fields. The default value of “0” will cause each field to be separated by a space.

Parameters:

[Optional] Int "Amount of data to display (0=unlimited)" (Default value: 0)
[Optional] Str “Separator” (Default value: 0)

Example:
FIELD cont_data (ToEndOfLayer) (StringOfHex 20) "Continuation Report Data"

Method: StringOfUTF8

Description:
This method formats the current field as a string of UTF8 characters. Non-printable characters are displayed as dots. See the description for the StringOfASCII Method for information on the "Amount of data to display" parameter.

Parameters:

[Optional] Int "Amount of data to display (0=unlimited)" (Default value: 0)

Examples:

FIELD stringofutf8_1_bit_off (Fixed 1 bit) (StringOfUTF8) "1 bit off"

Method: StringOfUnicode

Description:

This method formats the current field as a string of Unicode characters. Because the software can be run on systems that do not support Unicode, this method ignores the upper byte of each byte pair and interprets the lower byte as if it were an ASCII character. This works for the Latin alphabet but will not work for other alphabets. Non-printable characters are displayed with a dot. See the description for the StringOfASCII Method for information on the "Amount of data to display" parameter.

Parameters:

[Optional] Int "Amount of data to display (0=unlimited)" (Default value: 0)

Example:

FIELD unicode_string (ToEndOfLayer) (StringOfUnicode 30) "Unicode String"

Method: Table

Description:

Table method extracts a string from a table for display. The method extracts either of two strings according to the context in which it is to be displayed. For display in the Summary pane, the first string in the table entry is retrieved. For display in the Decode pane, the second string is used. If only one string is present, it is used in both the Summary and Decode panes. If the field value fails to match an entry in the table, the Default entry is used and the field value is displayed in decimal or optionally in hex (as specified by the second parameter).

When using the Table method, the field value must be 64 bits or less in size.

Parameters:

[Required]	Table	"Which table"	
[Optional]	List	"Radix for default case"	(Default value: Decimal)
		List values:	Decimal
			Hex

Example:

FIELD usb_request_type (Fixed 2) (Table request_types_table) “Request”

Method: UUID

Description:
This method formats the field as a 16 byte Universally Unique Identifier (UUID). Also called Globally Unique Identifier (GUID). An error will be displayed if UUID is used with fields that are not exactly 16 bytes in size.

The display format is (where 0x represents a hex digit):
0x0x0x0x-0x0x-0x0x-0x0x-0x0x0x0x0x0x

When the UUID's are transmitted in little-endian format of the DCE/RPC specification, the byte order is changed. To display UUID's in this format, use the optional parameter. The optional parameter is set to default value of 0. The default value is used in all the regular cases. (See the first example statement below.) in order to use the little-endian display, pass an integer parameter value greater than zero. (See the second example statement below.) When this format is used, the 16 bytes are displayed in this order:
03020100-0504-0706-0809-101112131415

Parameters:
[Optional] int "Does this use the DCE/RPC little-endian NDR?"

Example:
Field uuid (Fixed 16) (UUID) IN_SUMMARY "UUID" "UUID"

Field uuid_dcerpc_littleendian (Fixed 16) (UUID 1) IN_SUMMARY "UUID" "UUID"
HIST_DATA_ORDER

Group: FRAME_TRANSFORMER

Method: RealignOnByteBoundary

Description:
This method creates a new frame with the data realigned. This is called when the layer stops in the middle of a byte, and the next layer needs to pick up in the same byte.

Parameters:

[Required] Field "Field to transform to"

Example:
FRAME_AFTER_DECODING (RealignOnByteBoundary crc_value)

Method: StripDoubledChar

Description:
This method looks for two of the same character in a row and strips one out.

Parameters:

[Required] Int "Escaped char"

Example:
FRAME_BEFORE_DECODING (StripDoubledChar 0x10)

Group: NEXT_PROTOCOL

Method: FromField

Description:
This method returns the value in the field named in the parameter. The software makes no check that the field contains a valid value; if the field has not been processed then the result will be an indeterminate value. This lack of a check is not due to laziness; it is because making such a check would be time consuming to a degree unacceptable when decoding data arriving from a fast line.

Parameters:
[Required] Field "Field that contains the next protocol"

Example:
FIELD rej_pack (FromField Bytes length 4) (StringOfHex 16) "Rejected Packet"

Method: FromPortAssignment

Description:
This method returns the value from the personality file for one of the two fields. The port assignment is user-defined in the Decoder Parameters dialog.

Parameters:
[Required] StaticName ”Inter-frame data name”
[Required] Field "Field to check first for the next protocol"
[Required] Field "Field to check second for the next protocol"

Example:
NEXT_PROTOCOL (FromPortAssignment UDP src_port dst_port)

Method: IsAlways

Description:
The method IsAlways is used when the next protocol is invariable. It takes one parameter which is the value to be returned.

Parameters:
[Required] Int64 "Value to return"

Example:
NEXT_PROTOCOL (IsAlways 0x7f008006)

Group: NEXT_PROTOCOL_SIZE

Method: FromField

Description:
The method FromField returns the value from the field named in the parameter. The value is used to determine the size of the next protocol layer.

The "Offset" parameter adds or subtracts from the value in the "Field" parameter. Positive values are added, negative values are subtracted.

Parameters:

[Required]	Field	"Field that contains the next protocol size"
[Required]	Int	"Offset"

Example:
NEXT_PROTOCOL (FromField nlpid)

Method: FromFieldIfLessThan

Description:
If the value of the field specified is less than the "Threshold" parameter, it is returned as the size of the next protocol layer. Otherwise, the size of the next protocol layer is the rest of the payload, minus the value of the "Offset" parameter.

Parameters:

[Required]	Field	"Field that contains the next protocol size"
[Required]	Int	"Threshold for length"
[Required]	Int	"Offset"

Example:
NEXT_PROTOCOL_SIZE (FromFieldIfLessThan type_length 0x5dd 4)

Method: OffsetToEnd

Description:
This method sets the size of the next protocol layer as the rest of the frame minus the value of the "Offset" parameter. The size of the next layer is thus the number of bytes in the rest of the frame minus two to allow for the 16-bit frame-check sequence (FCS) at the end.

Parameters:

[Required]	Int	"Offset from the end of frame"

Example:
The Async PPP decoder, for example, includes the statement:
NEXT_PROTOCOL_SIZE (OffsetToEnd 3)

Method: OffsetToEndTakesFieldParam

Description:
This method sets the size of the next protocol layer as the rest of the frame minus the value of the "Offset" parameter.

The size of the next layer is thus the number of bytes in the rest of the frame minus mic_size to allow for the mic_size bytes MIC at the end.

Parameters:
[Required] Field "Offset from the end of frame"

Example:
The ZigBee NWK decoder, for example includes the statement:
NEXT_PROTOCOL_SIZE(OffsetToEndTakesFieldParam mic_size)

Method: ThisLayerLength

Description:
This method sets the size of the next protocol layer to the value in the "Field" parameter minus the value of the "Offset" parameter. ThisLayerLength is used when a field in the current layer indicates the combined size of the current layer plus the payload, where the payload contains the next layer(s).

Parameters:
[Required] Field "Field that contains the length in bytes of this layer"
[Optional] Int "Offset" (Default value: 0)

Example:
NEXT_PROTOCOL_SIZE (ThisLayerLength total_length)

Group: POSTPROCESSING

There are no generic Postprocessing methods.

Group: PREPROCESSING

Method: InitField

Description:
This method initializes the passed field before starting the decoder. This is useful if you want to do a test on a field, but it is not always encountered in the frame. This forces a known value into it.

Parameters:

[Required]	Field	"Field to set"	
[Optional]	Int64	"Value to set it to"	(Default value: 0)

Example:
PREPROCESSING (InitField conn_numb -1)

Group: **PROCESSING**

There are no generic Processing methods.

Group: RETRIEVING_DATA

Method: AddField

Description:
AddField method adds the value of another field to the value of the current field, storing the result in the current field.

Parameters:

[Required]	Field	"Field to add"

Example:
FIELD total_length (Fixed 2) RETRIEVE (AddField optional_fields) (Decimal) “Length”

Method: AddFieldToField

Description:
AddField To Field method adds the values of two other fields, storing the result in the current field.

Parameters:

[Required]	Field	"First field"
[Required]	Field	"Second field"

Example:
FIELD pword_address (Fixed 18 bits) RETRIEVE (AddFieldToField first14_bits last6_bits) (Decimal) "P-word"

Method: AddFieldToInteger

Description:
This method adds the value of another field to a constant, storing the result in the current field.

Parameters:

[Required]	Field	"Field"
[Required]	Int64	"Constant value to add"

Example:
FIELD increment_points (Fixed 0) RETRIEVE (AddFieldToInteger num_points 32) (Decimal) “Number of Points”

Method: AddInteger

Description:
This method adds an integer constant to the value of the current field.

Parameters:

[Required] Int64 "Constant value to add"

Example:
FIELD increment_count (Fixed 2) RETRIEVE (AddInteger 10) (Decimal) “MapCounter”

Method: AddIntegerToField

Description:
This method adds a constant to the value of another field, storing the result in the current field.

Parameters:

[Required] Int64 "Constant value"
[Required] Field "Field to add"

Example:
FIELD new_count (Fixed 0) RETRIEVE (AddIntegerToField old_count 10) (Decimal) “New Count”

Method: AndMask

Description:
AndMask method performs a bit-wise AND operation with the field value and a given mask. The single parameter is the mask, which can be up to 64 bits (8 bytes) long.

A bit-wise AND compares each bit in the field value to the corresponding bit in the mask. If both are 1, the result is 1. Otherwise the result is 0. For example, if a byte field contained 0x77 and a RETRIEVE was performed using (AndMask 0xF0), the result would be 0x70.

Parameters:

[Required] Int64 "Mask to apply to data"

Example:
FIELD connection (Fixed 2) RETRIEVE (AndMask 0xffff) (Decimal) "Address”

Method: BitOffset

Description:
This method returns the bit offset of the start of the current field.

Parameters:
None

Example:
FIELD field_begin_data (Fixed 0) RETRIEVE (BitOffset) (Hex) SUPPRESS_DETAIL

Method: ClearPersistentField

Description:
This method clears the specified inter-frame data that was previously set.

Parameters:

[Required]	StaticName	”Inter-frame data name”

Example:
RETRIEVE (ClearPersistentField tid) (Decimal) IN_SUMMARY "Trans ID" "Transaction ID"

Method: DivideField

Description:
DivideField method divides the current field value by the value of another field, storing the result in the current field. An error will be displayed and a result of -1 returned if the divisor is zero.

Parameters:

[Required]	Field	"Field to divide by"

Example:
Field quotient (Fixed 2) RETRIEVE (DivideField divisor) (Decimal) “Result”

Method: DivideFieldByField

Description:
The method divides the value of the first field by the second field, storing the result in the current field. An error will be displayed and a result of -1 returned if the divisor is zero.
Parameters:

[Required]	Field	"First field"
[Required]	Field	"Second field"

Example:
Field quotient (Fixed 2) RETRIEVE (DivideFieldByField dividend divisor) (Decimal) “Result”

Method: DivideFieldByInteger

Description:
This method divides the value of another field by a constant, storing the result in the current field. An error will be displayed and a result of -1 returned if the divisor is zero.

Parameters:

[Required]	Field	"Field"
[Required]	Int64	"Constant value to divide by"

Example:
Field quotient (Fixed 2) RETRIEVE (DivideFieldByInteger dividend 8) (Decimal) “Result”

Method: DivideInteger

Description:
This method divides the current field value by an integer constant, storing the result in the current field. An error will be displayed and a result of -1 returned if the divisor is zero.
Parameters:

[Required]	Int64	"Constant value to divide by"

Example:
Field quotient (Fixed 2) RETRIEVE (DivideInteger 16) (Decimal) “Result”

Method: DivideIntegerByField

Description:
This method divides a constant by the value of another field, storing the result in the current field. An error will be displayed and a result of -1 returned if the divisor is zero.

Parameters:

[Required]	Int64	"Constant value"
[Required]	Field	"Field to divide by"

Example:
Field quotient (Fixed 2) RETRIEVE (DivideIntegerByField 24 divisor) (Decimal) “Result”

Method: FromFlag

Description:
This method returns a single "data related flag" selected by flag number. The flags are numbered 0 to 31.

Parameters:

[Required] List "Flag number"

List values: 0
1
2
3
etc.
...
30
31

Example:
FIELD drf_flag (Fixed 0) RETRIEVE (FromFlag 15) (Hex) SUPPRESS_DETAIL

Method: FromFlags

Description:
This method returns the 32 "data related flags" packed into a 32 bit value. Flag number 0 is the least significant bit, flag number 31 is the most significant bit.

Parameters:

None

Example:
FIELD fcs_flag (Fixed 0) RETRIEVE (FromFlags) ALSO (AndMask 0x00000100) (Hex) SUPPRESS_DETAIL

Method: IndexedPersistentField

Description:
This method retrieves a field stored previously by a StoreIndexedPersistentField method. This method requires the name of the inter-frame data to retrieve, and a field giving the index to use. See the description of StoreIndexedPersistentField for a full explanation of how this works.

Parameters:

[Required] StaticName ”Inter-frame data name”
[Required] Field "Field that contains the index"

Example:
RETRIEVE (IndexedPersistentField usb_type_with_address previous_frame_address)

Method: IndexedPersistentFieldAndDelete

Description:
This method retrieves a field stored previously by a StoreIndexedPersistentField method. This method requires the name of the inter-frame data to retrieve, and a field giving the index to use. See the description of StoreIndexedPersistentField for a full explanation of how this works.

Parameters:
[Required] StaticName "Inter-frame data name"
[Required] Field "Field that contains the index"

Example:
FIELD request_class (Fixed 0) IF (IndexedPersistentStaticExists cls sender_context) RETRIEVE (IndexedPersistentFieldAndDelete cls sender_context) (Hex) "(Request Class)"

Method: IntraframeField

Description:
IntraframeField method retrieves a field stored previously by the StoreIntraframeField method. If the value is unavailable because the stored field hasn't been encountered in this frame, an error is returned saying so.

This method requires the name of the intra-frame data to retrieve. The syntax is:
RETRIEVE (IntraframeField intra_frame_data_name).

Parameters:
[Required] StaticName "Intra-frame data name"

Example:
FIELD piconet (Fixed 0) RETRIEVE (IntraframeField piconet) (Decimal) SUPPRESS_DETAIL

Method: ModuloInteger

Description:
This method calculates the modulus of another field with respect to an integer constant and stores the result in the current field.

Parameters:
[Required] int64 "Constant value to divide by to calculate modulus (divisor)"

Example:
Field modulo_value (Fixed 0) RETRIEVE (StoreInteger 256) ALSO (ModuloInteger 7) (Decimal) SUPPRESS_DETAIL

Method: ModuloField

Description:
This method calculates the modulus by dividing the current field value by the value of another field and stores the result in the current field.

Parameters:

[Required]	Field	"Field to divide by to calculate modulus (divisor)"

Example:
Field modulo_value (Fixed 0) RETRIEVE (StoreInteger 256) ALSO (ModuloField modulo_divisor) (Decimal) SUPPRESS_DETAIL

Method: ModuloFieldByField

Description:
This method calculates the modulus by dividing the value of the first field by the second field and stores the result in the current field.

Parameters:

[Required]	Field	"First Field (quotient)"
[Required]	Field	"Second Field (divisor)"

Example:
Field modulo_value (Fixed 0) RETRIEVE (StoreField modulo_quotient) ALSO (ModuloFieldByField modulo_quotient modulo_divisor) (Decimal) SUPPRESS_DETAIL

Method: ModuloFieldByInteger

Description:
This method calculates the modulus by dividing the value of another field by a constant integer and stores the result in the current field.

Parameters:

[Required]	Field	"Field (quotient)"
[Required]	int64	"Constant value to divide by to calculate modulus (division)"

Example:
Field modulo_value (Fixed 0) RETRIEVE (StoreField modulo_quotient) ALSO (ModuloFieldByInteger modulo_quotient 11) (Decimal) SUPPRESS_DETAIL

Method: ModuloIntegerByField

Description:
This method calculates the modulus by dividing a constant by the value of another field and store the result in the current field. PARAM "Constant value" int64
PARAM "Field to divide by to calculate modulus" field

Parameters:

[Required]	int64	"Constant value (quotient)"
[Required]	Field	"Field (divisor)"

Example:
Field modulo_value (Fixed 0) RETRIEVE (StoreInteger 124) ALSO (ModuloIntegerByField 124 modulo_divisor) (Decimal) SUPPRESS_DETAIL

Method: MultiplyField

Description:
This method multiplies the current field value by the value of another field.

Parameters:

[Required]	Field	"Field to multiply by"

Example:
FIELD group_length (Fixed 4) RETRIEVE (MultiplyField num_groups) (Decimal) "Total segment length"

Method: MultiplyFieldByField

Description:
This method Multiplies the values of two fields, storing the result in the current field.

Parameters:

[Required]	Field	"First field"
[Required]	Field	"Second field"

Example:
FIELD total_length (Fixed 0) RETRIEVE (MultiplyFieldByField num_groups group_length) (Decimal) “Total segment length”

Method: MultiplyFieldByInteger

Description:
This method multiplies the value of another field by a constant, storing the result in the current field.

Parameters:

[Required]	Field	"Field"
[Required]	Int64	"Constant value to multiply by"

Example:
FIELD total_length (Fixed 4) RETRIEVE (MultiplyFieldByInteger number_groups 24) (Decimal) "Total segment length"

Method: MultiplyInteger

Description:
This method multiplies the current field value by an integer constant.

Parameters:

[Required]	Int64	"Constant value to multiply by"

Example:
FIELD group_length (Fixed 4) RETRIEVE (MultiplyInteger 6) (Decimal) "Total segment length"

Method: MultiplyIntegerByField

Description:
This method multiplies a constant by the value of another field, storing the result in the current field.

Parameters:

[Required]	Int64	"Constant value"
[Required]	Field	"Field to multiply by"

Example:
FIELD group_length (Fixed 4) RETRIEVE (MultiplyIntegerByField 24 num_groups) (Decimal) "Total segment length"

Method: OrMask

Description:
This method performs a bit-wise inclusive OR operation with the field value and a given mask. The single parameter is the mask, which can be up to 64 bits (8 bytes) long.

A bit-wise inclusive OR compares each bit in the field value with the corresponding bit in the mask. If either bit is 1, the result is 1. If both bits are 1, the result is 1. Otherwise the result is 0.

Parameters:

[Required]	Int64	"Mask to apply to data"

Example:
If a byte field contained 0x77 and a RETRIEVE was performed using (OrMask 0xF0), the result would be 0xF7

Method: PersistentField

Description:
This method retrieves a field stored previously by a StorePersistentField method. When using this method, the first item after the method name must be the name of the inter-frame data in which the field was stored.

Parameters:

[Required] StaticName "Inter-frame data name"

Example:
To continue the example used in the StorePersistentField method description, if you wanted to retrieve the value, you would use this syntax:

FIELD retrieve_fieldname (Fixed 0 Bits) RETRIEVE (PersistentField KeepThisData) (Decimal) “Value of Persistant Field”

Method: ReplaceWithExtraData

Description:
ReplaceWithExtraData method looks up the value of the current field in the table specified in the parameter, and then replaces the value of the current field with the extra data from the table. If the table entry does not have extra data, zero is returned. Usually used in connection with the STORE keyword.

For example, assume a protocol has a field where the value maps to another value indicating a size. The method allows you to put the actual size in the field, where it can be used for other purposes.

Parameters:

[Required] Table "Which table"

Example:

In this example, storing the retrieved value under a different field name allows you to display the correct string from the table, and still use the correct value for the "next_field" field.

```
TABLE sizes
{1 "1 Bit" "1 Bit" 1}
{2 "4 Bits" "4 Bits" 4}
{3 "1 Byte" "1 Byte" 8}
ENDTABLE
```

FIELD size_lookup (Fixed 1 Byte) (Table sizes) "Next Field Size" STORE field_size RETRIEVE (ReplaceWithExtraData sizes)

FIELD next_field (FromField Bits field_size) (Decimal) "Number"

Method: ShiftLeft

Description:
ShiftLeft method shifts the data to the left by the number of bits specified in the parameter. If a field contained the value 0x7f and a RETRIEVE (ShiftLeft 4) was done, the resulting value would be 0xf0.

Parameters:

[Required] Int "Number of bits to shift"

Example:
FIELD service_key (Fixed 0) RETRIEVE (AddIntegerToField 0xFFFF instance_id) ALSO (ShiftLeft 8) (Hex) SUPPRESS_DETAIL

Method: ShiftRight

Description:
ShiftRight method shifts the data to the right by the number of bits specified in the parameter. If a field contained the value 0x7F and a RETRIEVE (ShiftRight 4) was done, the resulting value would be 0x07.

Parameters:

[Required] Int "Number of bits to shift"

Example:
FIELD minutes (Fixed 2) RETRIEVE (AndMask 0x07E0) ALSO (ShiftRight 5) (Decimal) "Minutes"

Method: SignExtension

Description:
SignExtension method extends the sign bit to 64 bits. This allows you to treat a field of any size between 1 and 63 bits as a signed integer rather than an unsigned one. The "Number" parameter indicates the number of bits in the value; usually this is the same as the field size. It must be explicitly provided in case a RETRIEVE alters the field.

Parameters:

[Required] Int "Number of bits in value"

Example:
FIELD int_value (Fixed 1) RETRIEVE (SignExtension 8) (Decimal) "Test Attribute"

Method: SplitField

Description:
This method removes some bits from the middle of the current field and moves the left part of the field over to the remaining bits on the right. The first parameter is the number of bits on the right side to keep, and the second parameter is the number of bits in the middle to delete.

If the raw data in a 16-bit field is 0x5A8F (binary 010110101000111) and it is retrieved using (SplitField 7 1), the low 7 bits are preserved, the high bit of the low byte is discarded and the remaining bits are right-shifted one place to produce a result of 0x2D0F (binary 00101101 00001111).

Parameters:

[Required]	Int	"Number of bits on right side to preserve"
[Required]	Int	"Number of bits to delete"

Example:
FIELD parity (Fixed 5) RETRIEVE (SplitField 2 6) (Hex) "Parity"

Method: StoreField

Description:
This method stores the value of another field to the current field.

Parameters:

[Required]	Field	"Field to store"

Example:
FIELD increment_address (Fixed 0) RETRIEVE (StoreField retrieved_addr_var) ALSO (AddInteger 1) (Hex) “New Address”

Method: StoreIndexedPersistentField

Description:
This method stores the current field value to an inter-frame data array, indexed by the value in the field specified in the parameter. This method is similar to the StorePersistentField method in that the data is saved forever rather than just for the length of the layer or frame. The difference between this method and StorePersistentField is that you can store multiple values to the same inter-frame data name.

Parameters:

[Required] StaticName "Inter-frame data name"
[Required] Field "Field that contains the index"

Example:
Say you have a master device talking to multiple slaves. The master sends a Read command to Slave 1, a Write Configuration command to Slave 4, and a Read Configuration command to slave 7. The command frames contain both the command and the slave the command is directed to. The responses from the slaves, however, don't reference the command, though they do say which slave the response is from. You can't correctly decode the responses unless you can match them up to the commands.

You can do this if you save the command value indexed by the slave number. The command part of the decoder might look like this:

FIELD slave_number (Fixed 1) (Decimal) "Slave Number"

FIELD command_code (Fixed 1) RETRIEVE (StoreIndexedPersistentField command slave_number) (Hex) "Command"

The response section might look like this:

FIELD slave_number (Fixed 1) (Decimal) "Slave Number"

FIELD retrieved_command_code (Fixed 0) RETRIEVE (IndexedPersistentField command slave_number) SUPPRESS_DETAIL

BRANCH (FromTable command_code retrieved_command_code)

Using the above example, after the first command frame is decoded, the array holds:
1 Read

After the second and third command frames are decoded, it holds:
1 Read
4 Write Configuration
7 Read Configuration

When a response comes back from Slave 7, the response section of the decoder looks up 7 in the array and retrieves the value "Read Configuration". It then branches to the Read Configuration section of the decoder.

Method: StoreInteger

Description:

Stores a constant value to the current field.

Parameters:

[Required] Int64 "Constant value to store"

Example:

FIELD dummy (Fixed 0) RETRIEVE (StoreInteger 255) (decimal) “Variable”

Method: StoreIntraframeField

Description:

StoreIntraframeField method saves the value of the current field for the duration of the current frame. This makes the field's value available to other layers within the frame (i.e. other decoders used in the frame). When the decoding of the frame is finished, the value is cleared prior to decoding the next frame. You can store as many fields as you wish, provided you use a different name for each one.

The method doesn't take a parameter, but you do need to specify a name to store the value under. The syntax is:

RETRIEVE (StoreIntraframeField intra_frame_data_name).

Parameters:

[Required] StaticName "Intra-frame data name"

Example:

FIELD src_addr (Fixed 4 Byte) RETRIEVE (StoreIntraframeField source_ip_address) (IPAddress) IN_SUMMARY Source "Source Address" TAG (Tag "Addr:IP:Src")

Method: StorePersistentField

Description:

StorePersistentField method saves the value of the current field in the specified inter-frame data. The item after the method name must be the name of the inter-frame data in which the field is to be stored. Data is retrieved using the PersistentField method.

Parameters:

[Required] StaticName "Inter-frame data name"

Example:

GROUP request IF (MatchTableData code_table req_or_resp 1)

FIELD code (Fixed 1) RETRIEVE (StorePersistentField KeepThisCode) (Decimal) "Code"

GROUP response IF (MatchTableData code_table req_or_resp 2)

FIELD response_code (Fixed 7 Bits) (TABLE response_code_table) IN_SUMMARY "Resp Code" 100 "Response Code

FIELD retrieved_code (Fixed 0 bits) RETRIEVE (PersistentField KeepThisCode) (Hex) "Retrieved Code

FIELD no_response (Fixed 1 Byte) IF (FieldIsBetween 0x00 0x1F retrieved_code) (Constant "No Response Required") "Response"

FIELD resp_req (Fixed 2 Bytes) IF NOT (FieldIsBetween 0x00 0x1F retrieved_code) (Table resp_code_table) "Response"

Method: SubtractField

Description:
This method subtracts the value of another field from the value of the current field, storing the result in the current field.

Parameters:

[Required]	Field	"Field to subtract"

Example:
FIELD calculation (Fixed 0) RETRIEVE (StoreField BufferLength) also (SubtractField descriptor_length) (Decimal) SUPPRESS_DETAIL

Method: SubtractFieldMinusField

Description:
This method subtracts the value of the second field from the value of the first field, storing the result in the current field.

Parameters:

[Required]	Field	"First field"
[Required]	Field	"Second field"

Example:
FIELD data_len (Fixed 0) RETRIEVE (SubtractFieldMinusField end_data begin_data) (Decimal) SUPPRESS_DETAIL

Method: SubtractFieldMinusInteger

Description:
SubtractFieldMinusInteger method subtracts a constant from the value of another field, storing the result in the current field.

Parameters:

[Required]	Field	"Field value"
[Required]	Int64	"Constant value to subtract"

Example:
FIELD number_bytes (Fixed 0) RETRIEVE (SubtractFieldMinusInteger structure_length 2) (Decimal) SUPPRESS_DETAIL

Method: SubtractInteger

Description:
This method subtracts an integer constant from the value of the current field, storing the result in the current field.

Parameters:

[Required]	Int64	"Constant value to subtract"

Example:
FIELD data_length (Fixed 0) RETRIEVE (StoreField tlvlength) ALSO (SubtractInteger 1) (Decimal) SUPPRESS_DETAIL

Method: SubtractIntegerMinusField

Description:
This method subtracts the value of another field from a constant, storing the result in the current field.

Parameters:

[Required]	Int64	"Constant value"
[Required]	Field	"Field to subtract"

Example:
FIELD calculate_reserved_length (Fixed 0) RETRIEVE (SubtractIntegerMinusField 25 identifier_length) (Decimal) "Reserved length"

Group: SIZE

Method: ArrayOf

Description:
This field contains an array of identical bytes, terminated by the first byte that is different.

Parameters:
[Required] Int64 "Byte that repeats"

Example:
FIELD start_flag (ArrayOf 0x7e) (StringOfHex) "Start Flag"

Method: Bit

Description:
Bit methods returns a size of 1.

Parameters:
None

Example:
FIELD over_flow_bit (Fixed 1 Bit) (Binary) "Overflow Bit"

Method: Byte

Description:
Byte methods returns a size of 8 (8 bits).

Parameters:
None

Example:
FIELD info_state (Fixed 1 Byte) (Decimal) "Info State"

Method: Fixed

Description:
Fixed methods return the number of bits specified. Two parameters are used, a count and a unit specifier. The latter may be any of:

"Bit" or "Bits" = 1-bit units

"Nibble" or "Nibbles" = 4-bit units
"Byte" or "Bytes" = 8-bit units
"Octet" or "Octets" = 8-bit units
"Word" or "Words" = 16-bit units
"Dword" or "Dwords" = 32-bit units (double words)
"Qword" or "Qwords" = 64-bit units (quad words)
"Times" - used in REPEAT COUNT (Fixed 3 Times) constructions. Equivalent of saying (Fixed 3 Bits) but is more clear in the context of a REPEAT

The singular and plural forms are distinguished only for readability; for instance, the method will return 8 whether you write "(Fixed 1 Byte)" or "(Fixed 1 Bytes)". The units parameter is optional and, if omitted, will default to "Bytes".

Parameters:

[Required]	Int	"Number (in units)"		
[Optional]	List	"Units" (Default value: Bytes)		
			List values:	Bit
				Bits
				Times
				Nibble
				Nibbles
				Byte
				Bytes
				Octet
				Octets
				Word
				Words
				Dword
				Dwords
				Qword
				Qwords

Example:

```
FIELD info_num (Fixed 4 Bytes) (Decimal) "Info Number"
```

Method: FixedUpTo

Description:
FixedUpTo methods return the fixed value given in the first parameter "Number", unless that would put the pointer past the absolute end point given by the "Length field" and "Offset" parameters. The absolute end point is found by starting immediately after the field in "Length field", adding the value of "Length field" and subtracting the value of "Offset". If the value in "Number" puts the pointer past the absolute end point, the size is truncated to the absolute end point.

Parameters:

[Required]	Int	"Number (in units)"

[Required]	Field	"Length field"		
[Optional]	Int	"Offset"	(Default value: 0)	
[Optional]	List	"Units" (Default value: Bytes)		
			List values:	Bit
				Bits
				Times
				Nibble
				Nibbles
				Byte
				Bytes
				Octet
				Octets
				Word
				Words
				Dword
				Dwords
				Qword
				Qwords

Example:
FIELD length (Fixed 1 Byte) (Decimal) “Length”

FIELD address (FixedUpTo 6 length 5) (Hex) “Address”

Assume the value in “length” is 10. Starting after the length field, the absolute end point is 10-5=5 bytes from the length field. The address field, which is right after the length field, is 6 bytes long, except that this is 1byte past the absolute end point. Instead of being set at 6 bytes, the size of “address” is set to 5. If the value of “length” was 14, the absolute end point would be 14-5 =9 bytes from the end of “length”. In this case, the size of the address field would be set to 6, because this does not go past the absolute end point.

This method handles cases where bad data may cause a group to overflow, which is particularly a problem with repeated groups.

Method: FromExtraData

Description:
This method derives a size from the extra data in a table, referenced by the value in a field that has already been processed. Three parameters are used, the first two being required. The first parameter is the table that contains the extra data, the second parameter is the field that will be used to index the table, the optional third parameter is the unit, that is one of the terms listed for the Fixed method ("Bits", "Bytes", etc.). The default is "Bytes".

Suppose the field "type" contains the value 6, and the "cmd" table contains 6 "type 6" "" 3. (FromExtraData cmd type Bytes) would return the size of 3 bytes (the actual value would be translated to 18 bits).

Parameters:

[Required]	Table	"Which table"		
[Required]	Field	"Which field"		
[Optional]	List	"Units" (Default value: Bytes)		
			List values:	Bit
				Bits
				Times
				Nibble
				Nibbles
				Byte
				Bytes
				Octet
				Octets
				Word
				Words
				Dword
				Dwords
				Qword
				Qwords

Example:

FIELD indices (FromExtraData not_qcode_11 resp_index_size Bits) (Decimal) "Index"

Method: FromField

Description:

FromField method derives a size from the value in a field that has already been processed. Three parameters are used, the first two being required. The first parameter is the unit, that is one of the terms listed for the Fixed method ("Bits", "Bytes", etc.). The second is the name of the field that contains the size. The optional third parameter is an adjustment, a (possibly negative) integer that is subtracted from the field value.

Suppose the field "data_count" contains the value 6. (FromField Words data_count 1) would extract the value 6, subtract 1, multiply by 16 (remember all Size methods return the length in bits), and return the result 80.

Parameters:

[Required]	List	"Units"		
			List values:	Bit
				Bits
				Times
				Nibble
				Nibbles
				Byte

			Bytes
			Octet
			Octets
			Word
			Words
			Dword
			Dwords
			Qword
			Qwords
[Required]	Field	"Which field"	
[Optional]	Int	"Offset (in units)"	(Default value: 0)

Example:

FIELD var_id (FromField Bytes var_len) (StringOfAscii 20) IN_SUMMARY ID 50 ID

Method: MinFieldOrEndOfLayer

Description:

This method derives a size from the value in a field that has already been processed. Three parameters are used, the first two being required. The first parameter is the unit, that is one of the terms listed for the Fixed method ("Bits", "Bytes", etc.). The second is the name of the field that contains the size. The optional third parameter is an adjustment, a (possibly negative) integer that is subtracted from the field value.

Suppose the field "data_count" contains the value 6. (FromField Words data_count 1) would extract the value 6, subtract 1, multiply by 16 (remember all Size methods return the length in bits), and return the result 80.

This will not exceed the end of the layer.

Parameters:

[Required]	List	"Units"	
		List values:	Bit
			Bits
			Times
			Nibble
			Nibbles
			Byte
			Bytes
			Octet
			Octets
			Word
			Words
			Dword
			Dwords
			Qword

			Qwords
[Required]	Field	"Which field"	
[Optional]	Int	"Offset (in units)"	(Default value: 0)

Example:
FIELD type_value (MinFieldOrEndOfLayer Bytes two_byte_length 3) (StringOfASCII 30) "Type"

Method: OddTerminated

Description:
OddTerminated method counts bytes up to and including the first odd byte found.

Parameters:
None

Example:
FIELD protocol (OddTerminated) (Decimal) SUPPRESS_DETAIL NO_MOVE

Method: OneToken

Description:
OneToken method find the first space, tab, CR, LF, or null character. The returned size is never larger than the maximum passed in. If no maximum is given, the maximum returned size is assumed to be the size of the rest of the frame.

Parameters:

[Optional]	Int	"Maximum size (in bytes)"	(Default value: -1)

Example:
FIELD at_normal_type (OneToken) (Table at_normal_type) "Command"

Method: RelativeFromField

Description:
This method figures out how much data has been consumed between the start of the field in the "Field that contains length" parameter and the current bit position. Note that this method is inclusive, which means if you need to find out how much data has been consumed not including the data in the first parameter field, you will need to include the size of that field as the second parameter.

RelativeFromField is useful when decoding protocols that carry multiple objects or functions in a frame and there may be extra data at the end of an object that you want to put into a field but you don't know how much there is. For example, each object starts with a 1 byte length field. There are variable length fields between the length field and the current bit position where you

want to put your data field. You can use this construction to find out the correct size for the field:

FIELD consume_data (RelativeFromField length) IF (MoreBytesInSegment length) (StringOfHex 64) "Data"

Usually used in conjunction with the Boolean method MoreByteInSegment so that the field doesn't show up if there isn't any extra data.

Parameters:

[Required]	Field	"Field that contains length"	
[Optional]	Int	"Size of Parameter Field in Bytes"	(Default value: 0)

Example:
FIELD unknown_or_undecoded_bytes (RelativeFromField length 2) IF (MoreBytesToDecode length 2)(StringOfHex) "Unknown Bytes"

Method: String

Description:
String method is provided for sizing null-terminated ASCII strings. The method searches forward for a null character and returns the size of the string including the null. If the search reaches the end of the layer without encountering a null then no error is given and the field is sized to the layer's end.

Parameters:
None

Example:
FIELD password (String) (StringOfASCII) "Password"

Method: ToEndOfLayer

Description:
This method return the number of bits from the pointer to the end of the layer, excluding the number of bytes in the "Bytes" parameter. For example, (ToEndOfLayer 2) would return the number of bits left in the frame minus 16. This method is useful when a field subsumes all data left in the layer except for a CRC or checksum at the end. The parameter may be omitted in which case no adjustment is made.

Parameters:

[Optional]	Int	"Bytes from end"	(Default value: 0)

Example:
FIELD bus_idle (ToEndOfLayer) (Binary) SUPPRESS_DETAIL

Method: ToTerminatingValue

Description:
ToTerminatingValue method scans ahead for the value specified in the "Value" parameter, and returns the number of bits between the value and the current bit. If the terminating value is not found, the size is set to the number of bytes remaining in the layer. The "Inclusive/Exclusive" parameter specifies whether to include the terminating value in the count or not. Thus (ToTerminatingValue Inclusive 0xFF) would count bytes until one was found containing 0xFF and return a count that included the 0xFF. If the pointer is not at a byte boundary when the scan starts, then the pointer is moved to the beginning of the next byte and the scan starts from there.

Parameters:

[Required] List "Inclusive/Exclusive"
List values: Inclusive
Exclusive
[Required] Int64 "Value that signifies the start of the next field or end of this one"

Example:
FIELD s_reg (ToTerminatingValue Exclusive 0x3d) (Table s_reg) "S-register"

Method: UnicodeString

Description:
Determines the length of a string of Unicode characters by scanning alternate bytes until it finds one that is NULL. The null byte is included in the count. Note that the data order specified for the decoder is not used here; the comparison is performed in host data order. For example, if the pattern at the pointer is (in hex): 00 48 00 45 00 4C 00 4C 00 50 00 00 … then (UnicodeString) would find 6 characters, 12 bytes and return 96.

Parameters:
None

Example:
FIELD file_name (UnicodeString) IF (FieldIs EqualTo 1 unicode) (StringOfUnicode) "Original File Name"

Method: UpTo

Description:
This method returns the number of bits specified. Two parameters are used, a count and a unit specifier. The latter may be any of:

"Bit" or "Bits" = 1-bit units
"Nibble" or "Nibbles" = 4-bit units
"Byte" or "Bytes" = 8-bit units
"Octet" or "Octets" = 8-bit units
"Word" or "Words" = 16-bit units
"Dword" or "Dwords" = 32-bit units (double words)
"Qword" or "Qwords" = 64-bit units (quad words)

The singular and plural forms are distinguished only for readability; for instance, the method will return 8 whether you write "(Fixed 1 Byte)" or "(Fixed 1 Bytes)". The units parameter is optional and, if omitted, will default to "Bytes".

This is the same as the "Fixed" method, except that if there are not enough bits remaining in the frame, the total number of bits remaining is used.

Parameters:

[Required]	Int	"Number (in units)"
[Optional]	List	"Units" (Default value: Bytes)

List values: Bit, Bits, Times, Nibble, Nibbles, Byte, Bytes, Octet, Octets, Word, Words, Dword, Dwords, Qword, Qwords

Example:

FIELD peek_self_contained (UpTo 4) IF (FieldIs EqualTo 1 is_first) (Hex) SUPPRESS_DETAIL NO_MOVE

Method: Zero

Description:
This method returns a size of 0.

Parameters:
None

Example:

```
FIELD asdu_addr_size (Zero) RETRIEVE (StoreField SizeOfASDUAddressFieldInBytes) ALSO
(AddInteger 1) (Hex) SUPPRESS_DETAIL
```

Group: START_BIT

Method: FromEnd

Description:
Moves the pointer to a given offset from the end of the layer. The first parameter is the number of units to move and the second is the unit (see the Fixed Size Method). Thus (FromEnd 4 Bits) would move the pointer 4 bits prior to the end of the layer while (FromEnd 4 Bytes) would move the pointer 32 bits from the end of the layer. Both parameters are optional and, if omitted, will default to zero and Bytes.

Parameters:

[Optional]	Int	"Offset"	(Default value: 0)
[Optional]	List	"Units"	(Default value: Bytes)

List values: Bit, Bits, Times, Nibble, Nibbles, Byte, Bytes, Octet, Octets, Word, Words, Dword, Dwords, Qword, Qwords

Example:
FIELD fcs START_BIT (FromEnd 1 Byte) (Fixed 1) (Hex) "FCS" NO_MOVE VERIFY (RFCommCrc crc_end_not_uih)

Method: FromEndSizeFromField

Description:
Moves the pointer to a given offset from the end of the layer, where the size of the offset is provided in another field. The first parameter is the field which contains the value of the offset and the second is the unit (see the Fixed Size Method). Thus (FromEndSizeFromField size_of_crc Bytes) would get the value stored in the field size_of_crc and move the pointer that many bytes prior to the end of the layer. The second parameter is optional and, if omitted, will default to Bytes.

Parameters:

[Required]	Field	"Field that contains offset"		
[Optional]	List	"Units" (Default value: Bytes)		
			List values:	Bit
				Bits
				Times
				Nibble
				Nibbles
				Byte
				Bytes
				Octet
				Octets
				Word
				Words
				Dword
				Dwords
				Qword
				Qwords

Example:

FIELD crc_value START_BIT (FromEndSizeFromField size_of_crc Bytes) (FromField Bytes size_of_crc) “CRC”

Method: FromStart

Description:
Moves the pointer to a given offset from the start of the layer. Parameters are the same as for the FromEnd method.

Parameters:

[Optional]	Int	"Offset"	(Default value: 0)	
[Optional]	List	"Units" (Default value: Bytes)		
			List values:	Bit
				Bits
				Times
				Nibble
				Nibbles
				Byte
				Bytes
				Octet
				Octets
				Word
				Words
				Dword
				Dwords
				Qword
				Qwords

Example:
FIELD can_id START_BIT (FromStart 5 Bits) (Fixed 11 Bits) (Hex) “CAN ID”

Method: Move

Description:
Moves the pointer the requested number of units from the current position. Use negative numbers to move backwards.
The first parameter is the number of units to move and the second is the unit (as listed for the Fixed Size Method). Thus, (Move -1 byte) would move the pointer back one byte.

Parameters:

[Required]	Int	"Amount to move"
[Optional]	List	"Units" (Default value: Bytes)

List values:
- Bit
- Bits
- Times
- Nibble
- Nibbles
- Byte
- Bytes
- Octet
- Octets
- Word
- Words
- Dword
- Dwords
- Qword
- Qwords

Example:
FIELD CheckOpen START_BIT (Move 5 Bits) (Fixed 3 Bits) (Hex) "SUPPRESS_DETAIL" NO_MOVE

Method: MoveAbsoluteFromField

Description:
This method takes the value in the "Field that contains offset" parameter and starting from the beginning of the layer, moves the pointer that many bytes plus the number of bytes in the "Additional Offset" parameter. A positive value in "Offset" moves the pointer forward, while a negative value moves back. An error will be triggered if the result points outside the frame. This method is useful for ensuring that decoding begins again at the proper location in case bad data prior in the frame causes trouble in a REPEAT loop or similar construction.

Parameters:

[Required]	Field	"Field that contains offset"

[Optional] Int "Additional Offset In Bytes" (Default value: 0)

Example:
FIELD DataCrc START_BIT (MoveAbsoluteFromField len 8) (Fixed 2) (Decimal) “CRC”

Method: MoveRelativeFromField

Description:
This method takes the value in the "Field that contains offset" parameter and starting from that field, moves the pointer that many bytes plus the number of bytes in the "Additional Offset" parameter. A positive value in "Offset" moves the pointer forward, while a negative value moves back. An error will be triggered if the result points outside the frame. This method is useful for ensuring that decoding begins again at the proper location in case bad data prior in the frame causes trouble in a REPEAT loop or similar construction.

Parameters:
[Required] Field "Field that contains offset"
[Optional] Int "Additional Offset In Bytes" (Default value: 0)

Example:
FIELD file_name START_BIT (MoveRelativeFromField fname_offset) (FromField Bytes name_size) (StringOfASCII) "File Name"

Method: MoveToField

Description:
This method moves the pointer to the beginning of the field specified. This must be a field that has already been decoded. MoveToField is handy when you need to process a field twice.

Parameters:
[Required] Field "Field to move to"

Example:
FIELD string_length START_BIT (MoveToField object_variation) (Fixed 1) (Decimal) "Octet String Length"

Method: MoveToFieldPlusOffset

Description:
This method moves the pointer to the beginning of the first field specified, and then moves forward the number of bytes in the second field parameter. This must be a field that has already been decoded.

Parameters:
[Required] Field "Field to move to"
[Required] Field "Offset beyond that to move in bytes"

Example:
FIELD mr_data START_BIT (MoveToFieldPlusOffset anchor multi_req_offset) (Fixed 16) (StringOfHex) "Data"

Method: NextByteBoundary

Description:
This method moves the pointer to the beginning of the next byte. If the pointer is already at a byte boundary, it doesn't move.

Parameters:

None

Example:
FIELD flags START_BIT (NextByteBoundary) (Fixed 1) (Hex) "Flags"

Method: NextWordBoundary

Description:
This method moves the pointer to the next word boundary. If the pointer is already at a word boundary, it doesn't move.

Parameters:

None

Example:
FIELD pd_unicode START_BIT (NextWordBoundary) (UnicodeString) (StringofUnicode) "Primary Domain"

Method: PreviousByteBoundary

Description:
This method moves the pointer to the beginning of the current byte. If the pointer is already at a byte boundary, it doesn't move.

Parameters:

None

Example:
FIELD read_flags_again START_BIT (PreviousByteBoundary) (Fixed 1) (Hex) "Flags"

Group: TAG

Method: Tag

Description:
This method returns the constant string passed to it

Parameters:

[Required]   Str   "Constant"

Example:
FIELD src_port (Fixed 2) (Table ports) IN_SUMMARY "Source Port""Source Port" TAG (Tag "Addr:TCP/UDP:Src")

Group: VERIFY

Method: CheckCrc802153MacHcs

Description:
Crc802153MacHcs is calculated after reversing the data bits in each byte and X-ORing the result with 0xFFFF after reversing the bits of the result.

Parameters:

[Required]	Int	"First byte to include (0=first byte in layer)"		
[Required]	Int	"Seed"		
[Optional]	List	"Swap Result" (Default value: NoSwap)		
			List values:	NoSwap
				Swap

Example:
FIELD crc_802153 (Fixed 2 Bytes) (Hex) "CRC" VERIFY (Check_Crc802153MacHcs 0 0x0000)

Method: Check_CrcCCITT

Description:
This method calculates the CRC checksum value according to the 16-bit CRC-CCITT specification.

Parameters:

[Required]	Int	"First byte to include (0=first byte in layer)"		
[Required]	Int	"Seed"		
[Optional]	List	"Swap Result" (Default value: NoSwap)		
			List values:	NoSwap
				Swap

Example:
FIELD ccitt_crc (Fixed 2 Bytes) (Hex) "CCITT CRC" VERIFY (Check_CrcCCITTrev 0 0x0000)

Method: Check_CrcCCITTrev

Description:
This method calculates the reverse CCITT CRC.

Parameters:

[Required]	Int	"First byte to include (0=first byte in layer)"		
[Required]	Int	"Seed"		
[Optional]	List	"Swap Result" (Default value: NoSwap)		
			List values:	NoSwap
				Swap

Example:

FIELD crc_revccitt START_BIT (FromEnd 3) (Fixed 2 Bytes) (Hex) "CRC" VERIFY (Check_CrcCCITTrev 0 0x0000)

Method: Check_CrcCCITTrevDataBytes

Description:
This method calculates the CRC checksum value according to the 16-bit CRC-CCITT specification after reversing all the data bytes.

Parameters:

[Required]	Int	"First byte to include (0=first byte in layer)"
[Required]	Int	"Seed"
[Optional]	List	"Swap Result" (Default value: NoSwap) List values: NoSwap Swap

Example:
FIELD crc START_BIT (FromEnd 5) (Fixed 4) (Hex) "CRC" VERIFY (Check_CrcCCITTrevDataBytes 0 NoSwap)

Method: Check_CrcCRC16

Description:
This method calculates the 16-bit CRC checksum.

Parameters:

[Required]	Int	"First byte to include (0=first byte in layer)"
[Required]	Int	"Seed"
[Optional]	List	"Swap Result" (Default value: NoSwap) List values: NoSwap Swap

Example:
FIELD crc START_BIT (FromEnd 2)(Fixed 2) (Hex) "CRC" VERIFY (Check_CrcCRC16 0)

Method: Check_CrcCRC16RevAndInvert

Description:
This method calculates the CRC after reversing and inverting the data bytes.

Parameters:

[Required]	Int	"First byte to include (0=first byte in layer)"
[Required]	Int	"Seed"
[Optional]	List	"Swap Result" (Default value: NoSwap) List values: NoSwap Swap

Example:
FIELD crc START_BIT (FromEnd 2) (Fixed 2) (Hex) "CRC" VERIFY (Check_CrcCRC16RevAndInvert 0)

Method: Check_CrcCRC16rev

Description:
This method calculates the 16-bit CRC after reversing the data bytes.

Parameters:

[Required]	Int	"First byte to include (0=first byte in layer)"	
[Required]	Int	"Seed"	
[Optional]	List	"Swap Result" (Default value: NoSwap)	
		List values:	NoSwap
			Swap

Example:
FIELD crc START_BIT (FromEnd 2) (Fixed 2) (Hex) "CRC" VERIFY (Check_CrcCRC16rev 0 0x0000)

Method: Check_CrcCRC32

Description:
This method calculates the 32-bit CRC checksum.

Parameters:

[Required]	Int	"First byte to include (0=first byte in layer)"	
[Required]	Int	"Seed"	
[Optional]	List	"Swap Result" (Default value: NoSwap)	
		List values:	NoSwap
			Swap

Example:
FIELD fcs START_BIT (FromEnd 5) (Fixed 4) (Hex) "Frame Check Sequence" VERIFY (Check_CrcCRC32 0 0xffffffff NoSwap)

Method: Check_CrcHDLC

Description:
This method calculates 2-byte CRC value according to the HDLC specification.

Parameters:

[Required]	Int	"First byte to include (0=first byte in layer)"	
[Required]	Int	"Seed"	
[Optional]	List	"Swap Result" (Default value: NoSwap)	
		List values:	NoSwap
			Swap

Example:

FIELD crc START_BIT (FromEnd 2) (Fixed 2) (Hex) CRC NO_MOVE VERIFY (Check_CrcHDLC)

Method: Check_CrcHDLCerr

Description:
This method calculates the Error checksum value according to the HDLC specification

Parameters:

[Required]	Int	"First byte to include (0=first byte in layer)"
[Required]	Int	"Seed"
[Optional]	List	"Swap Result" (Default value: NoSwap)
		List values: NoSwap
		Swap

Example:

Method: Check_CrcLRC

Description:
This method calculates the Longitudinal Redundancy Check CRC. The method XORs each byte in the layer, starting with the byte specified in the first parameter. The result is an 8-bit value.

Parameters:

[Required]	Int	"First byte to include (0=first byte in layer)"
[Required]	Int	"Seed"
[Optional]	List	"Swap Result" (Default value: NoSwap)
		List values: NoSwap
		Swap

Example:
FIELD bcc (Fixed 1) (Hex) "BCC" VERIFY (Check_CrcLRC 1 0)

Method: Check_CrcModBus

Description:
This method calculates a 2-byte CRC value according to the MODBUS protocol specification.

Parameters:

[Required]	Int	"First byte to include (0=first byte in layer)"
[Required]	Int	"Seed"
[Optional]	List	"Swap Result" (Default value: NoSwap)
		List values: NoSwap
		Swap

Example:

FIELD crc START_BIT (FromEnd 2) (Fixed 2) (Hex) IN_SUMMARY CRC 40 CRC NO_MOVE VERIFY (Check_CrcModBus 0 0xffff)

Method: Check_CrcModBus_LRC

Description:
This method calculates a 1-byte LRC value according to the MODBUS protocol specification.

Parameters:

[Required]	Int	"First byte to include (0=first byte in layer)"
[Required]	Int	"Seed"
[Optional]	List	"Swap Result" (Default value: NoSwap) List values: NoSwap Swap

Example:
FIELD crc START_BIT (FromEnd 1 Bytes) (Fixed 1) (Hex) CRC NO_MOVE VERIFY (Check_CrcModBus_LRC 0 0)

Method: Check_CrcSum

Description:
This method calculates the summed checksum. It adds each byte to the next byte and returns a 2-byte result.

Parameters:

[Required]	Int	"First byte to include (0=first byte in layer)"
[Required]	Int	"Seed"
[Optional]	List	"Swap Result" (Default value: NoSwap) List values: NoSwap Swap

Example:
FIELD crc (Fixed 2) (Hex) “CRC” VERIFY (Check_CrcSum 0 0)

Method: Check_CrcSum1comp

Description:
This method calculates the CRC by summing the data bytes and taking the 1’s complement of the sum.

Parameters:

[Required]	Int	"First byte to include (0=first byte in layer)"
[Required]	Int	"Seed"
[Optional]	List	"Swap Result" (Default value: NoSwap) List values: NoSwap

Swap

Example:
FIELD crc (Fixed 2) (Hex) “CRC” VERIFY (Check_CrcSum1comp 0)

Method: Check_CrcSum2comp

Description:
This method calculates the CRC by summing the data bytes and taking the 2’s complement of the sum.

Parameters:

[Required] Int "First byte to include (0=first byte in layer)"
[Required] Int "Seed"
[Optional] List "Swap Result" (Default value: NoSwap)
List values: NoSwap
Swap

Example:
FIELD crc (Fixed 2) (Hex) “CRC” VERIFY (Check_CrcSum2comp 0)

Method: Check_CrcXOR1comp

Description:
This method calculates the CRC by XORing the data bytes and taking the 1’s complement of the XOR-sum.

Parameters:
[Required] Int "First byte to include (0=first byte in layer)"
[Required] Int "Seed"
[Optional] List "Swap Result" (Default value: NoSwap)
List values: NoSwap
Swap

Example:
FIELD crc (Fixed 2) (Hex) “CRC” VERIFY (Check_CrcXOR1comp 0)

Method: Check_CrcXOR2comp

Description:
This method calculates the CRC by XORing the data bytes and taking the 2’s complement of the sum.

Parameters:
[Required] Int "First byte to include (0=first byte in layer)"
[Required] Int "Seed"

[Optional] List "Swap Result" (Default value: NoSwap)
List values: NoSwap
Swap

Example:
FIELD crc (Fixed 2) (Hex) "CRC" VERIFY (Check_CrcXOR2comp 0)

Method: Fail

Description:
This method always returns an error. It is useful when the decoder takes a path that is always an error

Parameters:
[Required] Str "Error message"

Example:

Method: FieldIs

Description:
This method returns OK if the statement comparing the field value with a given constant using the operator specified is true. If the comparison fails, the field value will be displayed in red in the Decode pane and the expected value(s) will be displayed next to it. The optional "Output Radix" parameter specifies whether to show the expected value(s) in decimal or hex.

Parameters:
[Required] List "Operator"
List values: GreaterThan
GreaterThanOrEqualTo
LessThan
LessThanOrEqualTo
EqualTo
NotEqualTo
[Required] Int64 "Match value"
[Optional] List "Output radix" (Default value: AsDecimal)
List values: AsDecimal
AsHex

Example:
Here is a typical example taken from the IP decoder. This serves to check that the version of IP used is at least 4:

```
FIELD version (Fixed 4 Bits) (TABLE ver_nums) "Version" VERIFY (FieldIs
GreaterThanOrEqualTo 4)
```

Method: FieldIsBetween

Description:
This method returns OK if the value of the current field is between the minimum and maximum values. This method is inclusive. In other words, if the value of the current field is exactly equal to the value of either the maximum or minimum, the method returns OK.

The comparison can be reversed by using the NOT keyword after the VERIFY. For example, VERIFY NOT (FieldIsBetween 5 10) will return OK if the field value is less than 5 or greater than 10, and fail if the value is 5, 6, 7, 8, 9 or 10.

Parameters:

[Required]	Int64	"Minimum value"
[Required]	Int64	"Maximum value"

Example:
GROUP cmdGroup2Explicit IF (FieldIsBetween 2 6 message_type)

Method: FieldLengthIs

Description:
This method makes sure that the field length is correct. This is useful for those fields that have variable lengths, but not all of the lengths are legal. The parameters and functionality are the same as for FieldIs, except that the method is verifying the length of the field rather than the contents.

Parameters:

[Required]	List	"Operator"		
			List values:	GreaterThan
				GreaterThanOrEqualTo
				LessThan
				LessThanOrEqualTo
				EqualTo
				NotEqualTo
[Required]	Int64	"Match value"		
[Optional]	List	"Units"	(Default value: Bytes)	
			List values:	Bit
				Bits
				Times
				Nibble
				Nibbles
				Byte
				Bytes

Octet
Octets
Word
Words
Dword
Dwords
Qword
Qwords

[Optional] List "Output radix" (Default value: AsDecimal)
List values: AsDecimal
AsHex

Example:
FIELD unexpected_data (ToEndOfLayer) IF (MoreBytes) (StringOfHex) "Unexpected Data" VERIFY (FieldLengthIs EqualTo 0)

Method: HdlcFcs

Description:
This method is used as a Checksum used for Async PPP frames.

Parameters:

[Optional] Int "Starting Offset" (Default value: 0)
[Optional] Int "Ending Offset" (Default value: 0)

Example:
FIELD FCS START_BIT (FromEnd 3) (Fixed 2) (Hex) IN_SUMMARY CRC 50 CRC NO_MOVE VERIFY (HdlcFcs 1 1)

Method: IsInTable

Description:
IsInTable method returns OK if the value of the current field exists in the specified table and is not the default. This method is used in the IP decoder to check that the next layer protocol is defined:

Parameters:

[Required] Table "Table"

Example:
FIELD protocol (Fixed 1) (Table protocol) IN_SUMMARY "Protocol" 100 "Protocol" VERIFY (IsInTable protocol)

Method: IsLengthOfLayer

Description:

This method returns OK if the value in the current field is equal to the length of the layer minus the value of the "Offset" parameter. This method is handy for verifying the internal size of a layer against the amount of data collected; the adjustment can be used to exclude a frame check or anything else of the sort.

Parameters:

[Optional] List "Operator" (Default value: EqualTo)
List values: GreaterThan
GreaterThanOrEqualTo
LessThan
LessThanOrEqualTo
EqualTo
NotEqualTo
[Optional] Int "Offset" (Default value: 0)
[Optional] List "Units" (Default value: Bytes)
List values: Bit
Bits
Times
Nibble
Nibbles
Byte
Bytes
Octet
Octets
Word
Words
Dword
Dwords
Qword
Qwords

Example:

FIELD length_of_layer (Fixed 1) (Decimal) "=0 Bytes" VERIFY (IsLengthOfLayer EqualTo 0 Bytes)

Method: MatchTableData

Description:

MatchTableData method returns TRUE if the table data matches the parameter "Value to match". A typical use of this method is to distinguish various classes of message, for example commands, responses and data.

Parameters:

[Required] Table "Which table"
[Required] Int64 "Value to match"
[Optional] Str "Error message" (Default value:)

Example:

Examples of its use can be seen in the HTTP and SMTP decoders. One instance from the latter decoders is:

```
TABLE cmd
{ 1 “Send” “Send” 1 }
{ 2 “Receive” “Receive” 1 }
{ 3 “Reserved” “Reserved” 0 }
{ 4 “Set” “Set” 1 }
{ Default “???” “???” 0 }
ENDTABLE

FIELD type (Fixed 4) (Table cmd) “Type” VERIFY (MatchTableData cmd 1 “Unsupported”)
```

This takes the value in the filed “tpe” and looks it up in the table “cmd”. Then the table data for that entry is checked against the constant given I nthe parameter “Value to match.” For example, if the value of “type” were 2, the test would return true because this value matches the “Receive” entry in the table, which matches the 1 given as the parameter.

Method: TcpChecksum

Description:
This method is the checksum defined in RFC 793 for TCP. See Pg 16-17 for an explanation. Note: If the input field is 0, then we bypass the check. We assume the field wasn't set.

Parameters:

[Required]	Field	"IP Source Addr"
[Required]	Field	"IP Dest Addr"

Example:
FIELD chksum (Fixed 2) (Hex) IN_SUMMARY Checksum "Checksum" VERIFY (TcpChecksum source_ip dest_ip)

Glossary

Boolean Method: A method used in an IF clause on a FIELD or GROUP statement, or in a REPEAT UNTIL clause on a GROUP statement. A Boolean Method returns True (in which case the statement is executed) or False (in which case the statement is skipped).

BRANCH: A **DecoderScript** statement that causes control to be transferred out of line to another statement.

Branch Method: A method for use in a **BRANCH** statement that determines the target of the branch.

Byte: In this manual the term "byte" is always used in the sense of an 8-bit value (sometimes referred to as an "octet").

Custom Method: A method written by a user to serve a protocol-specific need. (cf **Standard Method**.)

Data object: Short for Decoder Data Object.

Decode: A decode is the display of decoded protocol **frames** provided by protocol analyzer programs.

Decoder: A decoder is the program used to decode a protocol. The decoder, written in **DecoderScript** takes protocol layers as its input and outputs the various bits and pieces needed that make up a **decode**.

DecoderScript: The language used to write **decoders**.

Decoder Package: A decoder package comprises everything necessary to carry a **decoder** from one computer to another. The heart of the package is of course the decoder itself. It may also contain a **Frame Recognizer**, a **Frame Transformer**, and other **Custom Method**s, etc.

Decoder Data Object: An object used to store complex inter-frame data.

Drf: Data related flag. These are flags that convey information about a byte of data. They are usually errors, but may be other things depending on the type of driver used to capture the data. For example, with serial asynchronous products, there are three Data related flags: parity, framing and UART Overrun. For data captured with the ComProbe, which uses the ComProbe's driver, there are five Data related flags: USART Overrun, parity, framing, CRC and underrun.

DWord: Short for "Double Word". Used in the PC sense of a 32-bit (4-byte) quantity.

END_MAIN_PATH : A **DecoderScript** statement that marks the end of the main thread of **executable statements** in a **decoder**. Every **decoder** must include an **END_MAIN_PATH**

statement and control should reach it coincidentally with the pointer reaching the end of the layer.

Event: A protocol-analyzer capture consists of a sequence of events. Such events include data of course but also start-of-frame and end-of-frame markers, error indications, records of signal changes, etcetera.

Executable Statement: A DecoderScript statement of a type that plays an active role in decoding fields. Executable statements are BRANCH, FIELD, GROUP, GROUP FIELD, and RESERVED.

Field: A field in a protocol **layer**.

FIELD: A **DecoderScript** statement that processes (decodes) a single protocol **field**. (See also **GROUP FIELD**.)

Format Method: A method used to format data or otherwise create text for display in the **decode**.

Frame: The basic communication package in which data is shipped.

Frame Recognizer: A program, actually a C++ class, that scans captured data, recognizes where frames begin and end, and inserts start-of-frame and end-of-frame **events**.

Frame Transformer: A method that does whatever is necessary to prepare a frame for further processing. This could involve removing escape characters, decrypting or decompressing the data.

GROUP: A **DecoderScript** statement that brackets a group of other **executable statements**.

GROUP FIELD: A **DecoderScript** statement that both decodes a protocol **field** and brackets a further set of **executable statements**.

Host Data Order: One of two systems by which multi-byte numeric values are interpreted. The first byte in the field is treated as the byte of highest significance while the last byte is given the lowest significance. (cf **Network Data Order**.)

Inter-frame Data: Data that is saved while decoding one frame for use in interpreting subsequent frames. Such data is saved with the StorePersistentField or StoreIndexedPersistentField methods, or, if it is more complex, with a Decoder Data Object.

Intra-frame Data: Data that is saved while decoding one layer for use in interpreting subsequent layers within the same frame. Such data is saved with the StoreIntraframeField method.

Layer: The portion of a **frame** governed by one protocol.

Method: A subprogram that can be called from a **Decoder** to process data in some way. We distinguish two classes of method, **Standard Methods** and **Custom Methods**. We also categorize methods according to the context in which they are used. Categories are: **Boolean Methods**, **Branch Methods**, **Format Methods**, **Frame Transformers**, **NextProtocol Methods**, **Retrieval Methods**, **Size Methods**, **StartBit Methods** and **Verify Methods**.

Ndrf: Non-data related flag. This is a two-byte bit-encoded value containing information about the state of the circuit that may or may not be related to a data byte. A good example is control signal changes in serial circuits, which can occur independent of a data byte being received.

Network Data Order: One of two systems by which multi-byte numeric values are interpreted. The first byte in the **field** is treated as the byte of least significance while the last byte is given the highest significance. (cf **Host Data Order**.)

NextProtocol Method: A **method** that determines information about the protocol used at the next higher **layer** in a protocol stack.

Nibble: A 4-bit value.

Pointer: An internal pointer that points to the first bit of the **field** being processed.

Decoder ID: A 32-bit number that uniquely identifies a protocol within the protocol-analysis system.

Qword: Short for "Quad Word". Used in the PC sense of a 64-bit (8-byte) quantity.

RESERVED: A **DecoderScript** statement used to define a reserved protocol **field**.

Size Method: A **method** used to determine the size of a **field**.

Standard Method: A **method** supplied with the protocol analyzer. (cf **Custom Method**.)

StartBit Method: A **method** used to determine a position within the current **layer** where the decoding of a **field** should start.

Stream: Data arrives in streams. Ethernet has only one stream, while asynchronous serial has two streams, named DTE and DCE. Knowing the number of streams is important for frame recognizers and inter-frame data methods that need to track data on a per stream basis.

Verify Method: A method used either to verify a frame check sequence or to validate the value in a field.

Word: Used in the PC sense of a 16-bit (2-byte) value.

Index

A

ALSO 57, 58, 60, 77, 78, 79, 143, 144, 152, 159, 165
ALWAYS 41
APPLY_OVERRIDE_DATA 130
Automatically Request Missing Decoder Information 128
Autotraversal 3, 40

B

Boolean Methods 76
BooleanMethod 55
braces 17
Braces 18
BRANCH 13, 28, 29, 30, 31, 32, 36, 50, 52, 63, 68, 69, 71, 74, 76, 154, 158, 165, 244, 245
Branch Methods 76
BYTE_STREAM_FRAMER 11, 46, 157
byte-stuffing 47

C

CFA Maker 8
character escaping 47
Checksum/CRC Methods 79
comment 10, 17
comments 17
Conditional-Clause 55
COUNT 64, 165, 168
CStaticStorage 126

D

DECODE 11, 16, 20, 21, 22, 23, 30, 35, 74, 111, 146, 157, 160, 165
decoder 1, 17
    Header Section 10
    required statements 10
    Table section 11
Decoder Data Object
    User Interface to 127
Decoder ID 16, 39
DecoderScript 1
DecoderScript Sample Methods.mth 76
decompression 47
decryption 47
Detail Tag 65
    use in Groups 65
detail_tag 55

E

ELSE_SKIP 55
Encrypted Decoders 39
END_MAIN_PATH 13, 16, 20, 21, 24, 29, 31, 32, 36, 46, 52, 63, 71, 74, 155, 160, 165

ENDTABLE ... 12, 22, 23, 24, 30, 35, 51, 52, 62, 74, 147, 148, 150, 151, 152, 158, 160, 165
events ... 9
executable statements ... 13, 52

F

Fictitious Protocol ... 13
FIELD .. 12, 13, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 71, 72, 73, 74, 76, 77, 78, 79, 80, 83, 89, 94, 131, 132, 135, 140, 141, 143, 144, 145, 152, 153, 154, 155, 156, 158, 160, 165, 244, 245
  definition of ... 52
  naming ... 17
field_name ... 53
Field-Validation Methods ... 79
Filters ... 43
Format Methods ... 76
FormatMethod ... 54, 58
FP.cfd ... *See* CFA Maker
FP.dec ... *See* Fictitious Protocol
frame ... 9
frame recognition ... 4
Frame Recognizer ... 4, 44, 48, 102, 103, 105, 106, 107, 108, 114, 121, 127, 244, 245
Frame Recognizers
  included with the Protocol Analyzer ... 111
Frame Transformer ... 5, 46, 47, 92, 111, 114, 115, 116, 244, 245
FRAME_AFTER_DECODING ... 11, 46, 157
FRAME_BEFORE_DECODING ... 11, 47, 48, 157
FRM ... 73, 118, 126, 133

G

GET_OVERRIDE_DATA_FROM_PAGE ... 130
GET_OVERRIDE_DESCRIPTION ... 130
GET_OVERRIDE_MENU_ITEM_TEXT ... 130
GROUP ... 13, 17, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 36, 37, 38, 50, 51, 52, 64, 65, 66, 67, 68, 74, 76, 78, 89, 94, 140, 141, 143, 152, 153, 154, 155, 156, 158, 160, 166, 244, 245
  naming ... 17
GROUP FIELD ... 13, 33, 34, 52, 67, 68, 76, 141, 245

H

HalfDuplex ... 112
Header section
  protocol ID number ... 10
Header Section ... 10
  protocol name ... 10
hexadecimal ... 10
  protocol IDs ... 10
HOST_DATA_ORDER ... 11, 60, 157, 159, 166

I

i64UserCanSupplyMissingInfoOnThisFrame ... 131
id number ... 10
IF 55
IN_COLLAPSED ... 59, 158, 166

IN_SUMMARY..12, 21, 22, 23, 24, 25, 27, 28, 29, 30, 31, 32, 36, 38, 51, 56, 58, 59, 60, 63, 72, 73, 135, 143, 152, 153, 154, 158, 159, 160, 166
IN_SUMMARY_ONLY ... 59, 158, 166
INFO ... 59
Inter-frame Data ... 72
Intra-frame ... 72
Intra-frame Data ... 72
  retrieving ... 72
  storing ... 72

K

key
  as used in Port Assignments ... 43
  as used in Rules section ... 41
Keyword ... 17
keywords ... 9

L

layer ... 2

M

MARGIN_TEXT ... 59, 166
method invocation ... 18
Method Reference ... 54
methods
  naming ... 17
Methods
  Boolean ... 26, 27, 55, 56, 65, 76, 82, 83, 85, 92, 93, 99, 166, 167, 168, 169, 244, 246
  Branch ... 76
  Format ... 76
  NextProtocol ... 77
  Retrieval (Retrieve) ... 78
  Size ... 78
  StartBit ... 78
  Verify ... 78
MyProtocolName.personality ... 44

N

name/string ... 17
NETWORK_DATA_ORDER ... 11, 60, 157, 159, 167
NEXT_PROTOCOL ... 11, 47
NEXT_PROTOCOL_OFFSET ... 11, 46, 47, 157
NEXT_PROTOCOL_SIZE ... 11, 46, 47, 96, 146, 157
NextProtocol ... 77
NextProtocol Methods ... 77
NextProtocolOffset ... 77
NextProtocolSize ... 77
NO_MOVE ... 31, 32, 33, 34, 36, 37, 38, 59, 60, 63, 67, 83, 159, 167
NOT ... 55
number ... 18

P

PARAMETER....13, 52, 69
ParityError....112
payload....3
PAYLOAD_IS_BYTE_STREAM....11, 46, 47, 94, 157
Personality Files....39
  Filters section....43
  Port Assignments section....42
  Predefined Stacks section....42
  Protocol IDs used in....41
  Rules Section....40
Plain Text Decoders....39
Port Assignments....42
port number....42
POSTPROCESSING....11, 48, 96, 131, 135, 136, 157
Predefined Stacks....42
PREPROCESSING....11, 48, 96, 157
PRINT_IF....56
PROCESSING....58, 159
Processing-Clause....58
protocol header....3
protocol ID....10
Protocol ID....47
protocol name....10
ProtocolName.personality....39
Publisher ID....39

R

RECOGNIZER....11, 48, 102, 103, 106, 107, 108, 111, 112, 157
REPEAT....64, 65, 76, 78, 141, 154, 159, 165, 166, 168, 169, 244
REPEAT COUNT....64, 65, 78, 159
  definition of....64
REPEAT SIZE....64, 65, 78, 154, 159
  definition of....65
REPEAT UNTIL....64, 65, 76, 159, 244
  definition of....65
REQUEST_AUTO_OVERRIDE_DIALOG....129
REQUEST_OVERRIDE_DIALOG....128
REQUEST_PRESET_OVERRIDE_DIALOG....129
RESERVED....13, 33, 34, 38, 52, 63, 64, 152, 158, 168, 245, 246
Retrieval Methods....78
RETRIEVE....57, 62, 72, 73, 74, 77, 78, 86, 131, 132, 135, 140, 144, 145, 159, 165, 168
Retrieve-Clause....56
Rules section....40

S

sample files....1
SelectorChoices.txt....161
Set Decoder Parameters....128
SET_OVERRIDE_DATA_ON_PAGE....130
Show/Modify Decoding Rules....128
SignalChange....112

SIZE 46, 64, 87, 97, 168
Size Methods 78
SizeMethod 54
Standard Methods 76
START_BIT 34, 38, 55, 59, 60, 64, 78, 90, 98, 143, 159, 168
StartBit Methods 78
StartBitMethod 55
statement 17
statements 17
STORE 61, 62, 72, 78, 144, 159, 168
Store-Clause 61
String
  using quotes 17, 157
SUMMARY_COLUMNS 48
summary_tag 58
SUPPRESS_DETAIL 31, 32, 36, 37, 55, 59, 61, 63, 73, 143, 144, 168
SUPPRESS_IF_VERIFIED 60, 64, 159, 169

T

TABLE 11, 12, 22, 23, 30, 32, 35, 36, 50, 51, 52, 60, 62, 74, 135, 146, 147, 148, 150, 151, 152, 155, 158, 160, 169
Table Section 11
Tag Clause 59
Tag-Clause 59
TimeGap 112
timestamp 9
TRUNCATED_FIELD_OK 64

U

UltraEdit 9
UNTIL 64, 168, 169

V

variable 53, 54, 56, 61, 62, 65, 72, 80, 88, 93, 94, 96, 97, 98, 99, 105, 108, 111, 131, 132, 140, 144, 145, 159
VERIFY 12, 19, 20, 21, 24, 25, 26, 27, 28, 29, 31, 32, 35, 36, 37, 38, 60, 63, 68, 74, 78, 79, 140, 143, 152, 153, 154, 159, 160, 165, 169
Verify Methods 78
Verify-Clause 60
VerifyMethod 56