Developer Guide for CommView Decoder Plug-ins

How It Works

A custom decoder must be a 32-bit DLL written in Microsoft Visual C++. The best way to understand how custom decoding works and to write your own decoder is to look at the sample project that comes with this document. Below is a formal overview of the decoder architecture.

When CommView is launched, it scans for DLLs in the **Plugins** subfolder and calls the **Init** function for each DLL found. When the **Init** function is called, your DLL should register the decoder for one or several protocol-port pairs (there can be any number of registrations) using the **RegisterDecoder** and **RegisterDecodeAs** functions (these functions are called from **DHELPER.EXE**, as all the other functions mentioned in this document).

Each decoder for the same protocol-port pair overrides the previous one (even for internal CommView decoders). After **Init** returns, CommView can call the decoder at any time. Decoder functions are never called more than once at a time.

If your decoder uses external DLLs, please place them in the **Plugins** folder. CommView silently ignores any DLL without the **Init** function.

How to Register a Decoder

Decoder registration is performed by the **RegisterDecoder** function:

```
__declspec(dllimport) bool RegisterDecoder(proto_enum proto, int port, NewDecode_func *func);
```

proto	– protocol
port	– port number
func	– pointer to the decoder function

Possible proto values:

proto_enum_tcp	– TCP protocol
proto_enum_udp	– UDP protocol
proto_enum_etype	– Ethernet, ethertype is passed via the "port" argument
proto_enum_sap	– Ethernet 802.2, SAP is passed via the "port" argument
proto_enum_ipproto	– IP, protocol is passed via the "port" argument

Example:

```
RegisterDecoder(proto_enum_tcp, 1111, &Decode);
```

When CommView detects a packet from or to TCP port 1111, it calls the **Decode** function.

To use the CommView's **DecodeAs** function and configuration functions, you need to use another registration function:

```
__declspec(dllimport) bool RegisterDecodeAs(proto_enum proto, char *shortName, NewDecode_func *func,
Config_func *config, Test_func *test);
```

proto	– protocol
shortName	– short protocol name, will be displayed in DecodeAs list
func	– pointer to the decoder function
config	– pointer to the config function

test – pointer to the test function

Note: the **DecodeAs** function currently is implemented for proto_enum_tcp and proto_enum_udp protocols only.

Example:

```
RegisterDecodeAs(proto_enum_udp, "SpecProto", &udpDecode, &udpConfig, NULL);
```

CommView will add "SpecProto" to the **DecodeAs** list. Also, it will call the **udpConfig** function when the user selects "SpecProto" in the program's **Options => Plug-ins** tab and clicks **Configure**.

Decode Function

The function prototype is:

```
int NewDecode_func(OutputTree *otp, unsigned char *packet, int pack_size, __int64 protid, int curLevel, int &CountBits, int isSummary, NewCString& ProtName, int ByteOrder, int DataTypeByteOrder, int DataFieldByteOrder);
```

otp – the class that performs decoding tree construction
packet – pointer to the packet data (UDP or TCP data)
pack_size – size of the packet
protid – unique identifier (high 32 bits – window handle, low 32 bits – packet number in window)
curLevel – used to offset protocol decoding tree, you should always get curLevel of zero and you can ignore it
CountBits – used for bit structures, you should always get CountBits of zero and you can ignore it
isSummary – zero when full protocol decode tree is required and non-zero when just the summary is requested, in this case you should provide summary ASAP and return
ProtName – internal use, ignore this parameter
ByteOrder, DataTypeByteOrder, DataFieldByteOrder – byte order flags for internal use, ignore these parameters

All your decoder functions must return ERR_INCOMPLETE if a packet is damaged; otherwise ERR_OK must be returned.

Note: The packet pointer points to the first byte of the packet data that has been stripped of Ethernet, IP, TCP or UDP headers.

Config Function

CommView calls the **Config** function to allow plug-in configuration. The function prototype is:

```
__declspec(dllexport) void udpConfig(HWND window);
```

window – CommView window handle

It is recommended to use the Windows registry to save user plug-in configuration.

When the user modifies parameters of a plug-in, the plug-in may need to reprocess all packets as if they just have been received. This can be done by calling the **Reset** function. The function prototype is:

```
__declspec(dllimport) void Reset();
```

Test Function

CommView calls the **Test** function every time the user employs the **DecodeAs** command so that the plug-in can

allow or disallow it. The function prototype is:
```
__declspec(dllexport) bool Test_func(int port);
```

port – the port number that the user selecteded for **DecodeAs**

This function returns false if it prohibits **DecodeAs**, and true otherwise.

Event Function

CommView calls the **Event** function to notify plug-ins about new packet window creation/destruction. The function prototype is:
```
__declspec(dllexport) void Event(HWND window, bool create);
```

window – CommView packet window handle
create - true when called upon the window creation, false when called upon the window destruction

Drop Function

CommView calls the **Drop** function to notify plug-ins about packet removal. Packets are removed when the main window contains too many packets and the circular buffer removed the oldest packets, or when the user clears the **Log Viewer** window. The function prototype is:
```
__declspec(dllexport) void Drop(__int64 ids[], int count);
```

ids – array of packet identifiers
count – number of packer identifiers

Help File Registration

It is possible to register plug-in help files to be added to the CommView **Help** menu. The function prototype is:
```
__declspec(dllimport) bool RegisterHelp(char *name, char *path);
```

name – help file title as it should be displayed in the menu
path – help file name and full path

Calling GUI functions

The only callbacks where the GUI functions are allowed are the **Init** function and **Config** function. You must not call GUI functions in any of the other callbacks; otherwise, your plugin might hang.

Writing a Decoder

All output from your decoder must be provided via the **OutputTree** class. Please note that you should not create **OutputTree** objects; instead use the pointer that is passed to every decoder function. You can use the following **OutputTree** methods:

```
int PutString(const int level, const int type, const int offset, const int size, char *const name, char *const value);
int PutStringComment(const int level, const int type, const int offset, const int size, char *const name, char *const value, char *const cmt);
```

```
int PutInt(const int level, const int type, const int offset, const int size, char *const name, const int num);
int PutIntComment(const int level, const int type, const int offset, const int size, char *const name, const int num, char *const cmt);
int PutUint64(const int level, const int type, const int offset, const int size, char *const name, const unsigned __int64 num);
int PutUint64Comment(const int level, const int type, const int offset, const int size, char *const name, const unsigned __int64 num, char *const cmt);
int PutULong(const int level, const int type, const int offset, const int size, char *const name, const unsigned long num);
int PutULongComment(const int level, const int type, const int offset, const int size, char *const name, const unsigned long num, char *const cmt);
int PutLong(const int level, const int type, const int offset, const int size, char *const name, const signed long num);
int PutLongComment(const int level, const int type, const int offset, const int size, char *const name, const signed long num, char *const cmt);
int PutFloat(const int level, const int type, const int offset, const int size, char *const name, float *const num);
int PutFloatComment(const int level, const int type, const int offset, const int size, char *const name, float *const num, char *const cmt);
int PutDouble(const int level, const int type, const int offset, const int size, char *const name, double *const num);
int PutDoubleComment(const int level, const int type, const int offset, const int size, char *const name, double *const num, char *const cmt);
int PutBinary(const int level, const int type, const int offset, const int size, char *const name, const unsigned int num, const int len);
int PutBinaryComment(const int level, const int type, const int offset, const int size, char *const name, const unsigned int num, const int len, char *const cmt);
int PutHexString(const int level, const int type, const int offset, const int size, char *const name, unsigned char *const str, const unsigned short len);
int PutHexStringComment(const int level, const int type, const int offset, const int size, char *const name, unsigned char *const str, const unsigned short len, char *const cmt);
int PutIPv4Addr(const int level, const int type, const int offset, const int size, char *const name, unsigned char *const ip);
int PutIPv6Addr(const int level, const int type, const int offset, const int size, char *const name, unsigned char *const ip);
int PutIPXnet(const int level, const int type, const int offset, const int size, char *const name, unsigned char *const ipx_net);
int PutIPXnode(const int level, const int type, const int offset, const int size, char *const name, unsigned char *const ipx_node);
int PutMAC(const int level, const int type, const int offset, const int size, char *const name, unsigned char *const MAC);
int PutUnkAddr(const int level, const int type, const int offset, const int size, char *const name, unsigned char *const addr, int addr_size);

void UpdateString(const int pos, char *const value);
void UpdateName(const int pos, char *const name);
void UpdateSize(const int pos, const int size);
void AddSize(const int pos, const int size);
void UpdateOffset(const int pos, const int offset);
```

The first group of functions provides output from the decoder. The common parameters are explained below:

level	– this parameter is used for tree constructing (a top node like UDP will have the value of 0, a sub-node will have the value of 1, sub-sub-node will have the value of 2, and so on)
type	– this parameter is ignored, use zero
offset	– offset of the current field from the packet pointer passed to the decode function; it should not include the length of Ethernet, IP, TCP, or UDP headers.
size	– size of the current field

name – this will be used as a field name

Please note that there are functions that end with Comment. They are used when you need to add a comment after the value (see checksum field below).

All functions return the position index. You can use it later to change the name/value/offset/size.

It is possible to access lower layer data (Ethernet II, IP, TCP, UDP) via the **OutputTree** object. Structure definitions follow:

```
struct{
  bool  IsInitialized;
  unsigned short FrameControl;
  unsigned short Duration;
  unsigned char Mac1[6];
  unsigned char Mac2[6];
  unsigned char Mac3[6];
  unsigned short SeqControl;
  unsigned char Mac4[6];
} ieee80211;

struct{
  bool  IsInitialized;
  unsigned char dst_mac[6], src_mac[6];
  unsigned short proto;
} ethernet;

struct{
  bool  IsInitialized;
  unsigned char ver_len;
  unsigned char tos;
  unsigned short total_length;
  unsigned short id;
  unsigned short flags_offset;
  unsigned char ttl;
  unsigned char proto;
  unsigned short csum;
  unsigned char src_ip[4], dst_ip[4];
  unsigned char *after_ip_hdr;
} ip;

struct{
  bool  IsInitialized;
  unsigned short src_port, dst_port;
  unsigned short length;
  unsigned short csum;
} udp;

struct{
  bool  IsInitialized;
  unsigned short src_port, dst_port;
  unsigned int seq, ack;
  unsigned char off;
  unsigned char flags;
```

```
    unsigned short window;
    unsigned short csum;
    unsigned short urgent;
  } tcp;
```

Please be sure to check that **IsInitialized** is set to true before accessing any data from the structure.

Note: All short and int fields are converted from network to host order.

Another way to access lower layer data is to retrieve a pointer to the header with one of these functions:

```
unsigned char *GetIeee80211HdrPtr();
unsigned char *Get8023HdrPtr();
unsigned char *GetIpHdrPtr();
unsigned char *GetUdpHdrPtr();
unsigned char *GetTcpHdrPtr();
```

If the requested header is not present, then the function returns NULL.

Note: after_ip_hdr points to the first byte that follows the IP header. You can use this pointer to read any bytes of the packet that follow the IP header.

Sample Decoder (UDP)

All checks are removed for the sake of clarity

```
// UDP header definition
// Note: you need to distinguish between the udp and UDP_HDR structures
// UDP_HDR is used only in this sample
struct UDP_HDR {
        unsigned short src_p, dst_p;
        unsigned short length;
        unsigned short csum;
};

// Write header
otp->PutString(0, 0, 0, sizeof(UDP_HDR), "UDP", "");

// Write src port
otp->PutInt(1, 0, 0, 2, "Source port", ntohs(udp_hdr->src_p));

// Write dst port
otp->PutInt(1, 0, 2, 2, "Destination port", ntohs(udp_hdr->dst_p));

// Write length
otp->PutInt(1, 0, 4, 2, "Length", ntohs(udp_hdr->length));

// Write checksum
for (i = 0; i < (ntohs(udp_hdr->length) >> 1); i++) {
        tmp_s = static_cast<unsigned short>(~ntohsp(packet + (i << 1)));
        sum += tmp_s;
}

if ((ntohs(udp_hdr->length) & 0x01) == 0x01) {
        tmp_s = static_cast<unsigned short>(packet[ntohs(udp_hdr->length) - 1] << 8);
        tmp_s = static_cast<unsigned short>(~tmp_s);
```

```
        sum += tmp_s;
}

while ((sum & 0xFFFF0000) != 0) {
        sum = (sum & 0xFFFF) + (sum >> 16);
}

if (udp_hdr->csum == 0) {
        strcpy(tmp, "not computed");
} else if (sum == 0xFFFF) {
        strcpy(tmp, "correct");
} else {
        strcpy(tmp, "incorrect");
}

otp->PutIntComment(1, 0, 6, 2, "Checksum", ntohs(udp_hdr->csum), tmp);
```

Function List

PutString

```
int PutString(const int level, const int type, const int offset, const int size, char *const name, char *const value);
```

This function is used to put a string value (zero-terminated string) and to create empty nodes.

```
// Writes something like Password: dk345k
otp->PutString(1, 0, off, sz, "Password", pwd);

// Creates UDP node
otp->PutString(0, 0, 0, sizeof(UDP_HDR), "UDP", "");
```

PutInt, PutIntComment, PutUint64, PutULong, PutLong

```
int PutInt(const int level, const int type, const int offset, const int size, char *const name, const int num);
int PutIntComment(const int level, const int type, const int offset, const int size, char *const name, const int num, char *const cmt);
int PutUint64(const int level, const int type, const int offset, const int size, char *const name, const unsigned __int64 num);
int PutULong(const int level, const int type, const int offset, const int size, char *const name, const unsigned long num);
int PutULongComment(const int level, const int type, const int offset, const int size, char *const name, const unsigned long num, char *const cmt);
int PutLong(const int level, const int type, const int offset, const int size, char *const name, const signed long num);
int PutLongComment(const int level, const int type, const int offset, const int size, char *const name, const signed long num, char *const cmt);
```

These functions are used to write integer values of different size. Use **PutUint64** for 64-bit values, **PutULong** and **PutLong** for unsigned and signed longs, and **PutInt** for 8, 16 and 24 bit integers.

PutFloat

```
int PutFloat(const int level, const int type, const int offset, const int size, char *const name, float const num);
```

This function is used to put float values.

PutBinary

```
int PutBinary(const int level, const int type, const int offset, const int size, char *const name, const unsigned int num, const int len);
int PutBinaryComment(const int level, const int type, const int offset, const int size, char *const name, const unsigned int num, const int len, char *const cmt);
```

This function prints a binary number.

num	– number to print
len	– number of bits to print (no more than 32)

PutHexString

```
int PutHexString(const int level, const int type, const int offset, const int size, char *const name, unsigned char *const str, const unsigned short len);
int PutHexStringComment(const int level, const int type, const int offset, const int size, char *const name, unsigned char *const str, const unsigned short len, char *const cmt);
```

This function prints a hexadecimal dump of **len** bytes starting from **str**. The output will look like:
2B CD 1E 2B

PutIPv4Addr, PutIPv6Addr

```
int PutIPv4Addr(const int level, const int type, const int offset, const int size, char *const name, unsigned char *const ip);
int PutIPv6Addr(const int level, const int type, const int offset, const int size, char *const name, unsigned char *const ip);
```

These functions print IPv4 and IPv6 addresses in the standard notation. Length is not passed as it is fixed (32 and 128 bits respectively).

PutIPXnet, PutIPXnode

```
int PutIPXnet(const int level, const int type, const int offset, const int size, char *const name, unsigned char *const ipx_net);
 int PutIPXnode(const int level, const int type, const int offset, const int size, char *const name, unsigned char *const ipx_node);
```

These functions are used to write IPX net and node addresses respectively in the standard format.

PutMAC, PutUnkAddress

```
int PutMAC(const int level, const int type, const int offset, const int size, char *const name, unsigned char *const MAC);
int PutUnkAddr(const int level, const int type, const int offset, const int size, char *const name, unsigned char *const addr, int addr_size);
```

These functions print Ethernet MAC (6 bytes) and hardware address (**addr_size** bytes).

UpdateString

```
void UpdateString(const int pos, char *const value);
```

This function changes the node value to **value**.

UpdateName

```
void UpdateName(const int pos, char *const name);
```

This function changes the node name to **name**.

UpdateSize, AddSize

```
void UpdateSize(const int pos, const int size);
void AddSize(const int pos, const int size);
```

UpdateSize changes the node size to **size**, **AddSize** adds **size** to the node size.

UpdateOffset

```
void UpdateOffset(const int pos, const int offset);
```

This function changes the node offset to **offset**.

Template

To avoid compiling/linking problems you are provided with a Microsoft Visual C++ project with all the necessary headers and libraries.

Migrating From the Old API Version

If you have previously created a custom decoder for one of the older CommView versions, please follow this checklist to update it:

- Change linker settings: Replace fcd.lib with dhelper.lib; also remove fcd.lib and copy dhelper.lib.
- Replace decoder.h and output.h with the new versions and copy the new files: decode.h, NewCString.h, and external.h.
- Add `#include "external.h"` to you .cpp file(s)
- Decoder function prototypes have changed from:
  `int Decode_func(OutputTree *otp, unsigned char *packet, int pack_size, int param, __int64 id, unsigned short hour, unsigned short min, unsigned short sec, unsigned short msec)`
  to:
  `int NewDecode_func(OutputTree *otp, unsigned char *packet, int pack_size, __int64 protid, int curLevel, int &CountBits, int isSummary, NewCString& ProtName, int ByteOrder, int DataTypeByteOrder, int DataFieldByteOrder)`
  curLevel - used to offset protocol decoding tree. The value of curLevel that you get should always be zero and you can ignore it.
  CountBits - used for bit structures. The value of CountBits that you get should always be zero and you can ignore it.
  isSummary - zero when a full protocol decode tree is required and non-zero when only the summary is requested, in which case you should provide the summary ASAP and return.
  ProtName – for internal use, ignore this parameter.
  ByteOrder, **DataTypeByteOrder**, **DataFieldByteOrder** - byte order flags for internal use, ignore these parameters.
- Provide summary via one of the following methods for the OutputTree object. Note that only the first call sets the summary, all subsequent calls are ignored, so you should provide summary when you can and not call these functions if something is wrong with your protocol data, in which case the lower layer will provide the summary (TCP, UDP, etc.)
  ```
  void setSummary(char *protocol, char *str);
  void setSummary(NewCString& protocol, NewCString& str);
  ```
- Avoid using GUI functions in any callbacks besides **Init** and **Config**

After these changes, your decoder should compile and work as intended.