MultiAnalyzer

(Protokoll Analyse SDK)

Version 2026

(C) 2026 femvenner GmbH

Index of contents

1 History....3
2 Einleitung....4
2.1 Grundlegendes Konzept....4
2.1.1 Die Initialisierung....5
2.1.2 Das Auskoppeln der Daten....5
2.1.3 Das Einkoppeln der Daten....6
2.1.4 Das Ende der Analyse....6
3 DLL Funktionen....7
3.1 ThirdPartyComponent5V_Init()....7
3.1.1 Die Struktur „StructThirdPartyComponent5V_Init“....8
3.1.2 Die Struktur „StructThirdPartyComponent5V_RegData“....9
3.1.3 Beispiel für die Funktion „ThirdPartyComponent5V_Init()“....10
3.2 ThirdPartyComponent5V_Data()....11
3.2.1 Die Struktur „StructThirdPartyComponent5V_Data“....12
3.2.2 Die Struktur „StructThirdPartyComponent5V_Result“....14
3.2.3 Die Text Callback Funktionen....16
3.2.3.1 MSC Nachrichten Filtern....16
3.2.3.2 MSC Nachrichten-Formart....17
3.2.4 Beispiel für die Funktion „ThirdPartyComponent5V_Data()“....18
3.3 ThirdPartyComponent5V_Close()....19
3.4 Hilfreiche Makros und Funktionen....20
3.4.1 Lese Makros und Funktionen....20
3.4.2 Text Makros und Funktionen....23
3.4.3 Kombinierte Text und MSC-Einkoppel Makros....24

1 History

Date	Version	Author	Comment
2019-06-18	A23	GH/SZ	• Erste Version
2020-04-17	2019-12	GH	• Update der Version
2021-04-13	2020-12	GH	• Update der Version
2022-03-02	2021-12	GH	• Update der Version
2023-02-06	2022-12	GH	• Update der Version, Kapitel „3.4.4 Weitere Beispiele“
2025-02-06	2025-12	GH	• Update der Version
2026-11-12	2026-12	GH	• Update der Version

2 Einleitung

Mit dem MultiAnalyzer Protokoll Analyse SDK können DLLs erstellt werden, die es dem Benutzer erlauben, empfangene Daten des MultiAnalyzer auszukoppeln, zu analysieren und/oder zu entschlüsseln und wieder einzukoppeln. Zudem können eigene Filter-Option, um Nachrichten in der MSC zu unterdrücken, gesetzt werden. Um in der QoS diese DLL-eigenen Nachrichten gesondert darzustellen, können ihnen benutzerspezifische Nachrichten-Typen zugeordnet werden. Es können aber auch nur Daten exportiert werden, ohne in die Protokoll-Analyse einzugreifen, ein Beispiel wären IP- oder Sprach-Daten.

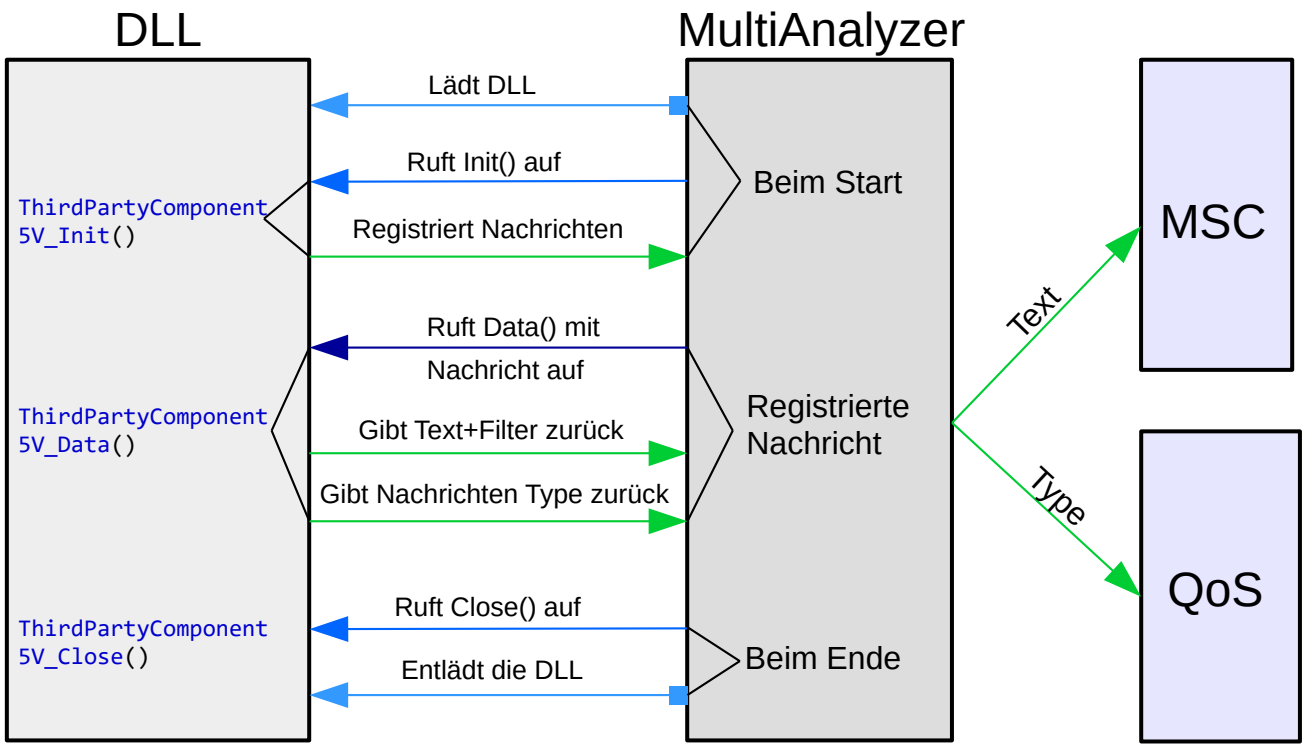
Die Programme MultiAnalyzerMsc und MultiAnalyzerQoS können gleichzeitig bis zu acht verschiedene DLLs laden. Dem Programm MultiAnalyzerProto kann eine DLL über die Kommando-Schnittstelle mitgeteilt werden.

Das SDK enthält für C/C++ Header-Dateien, Beispiel C-Dateien und ein Microsoft Visual Studio Projekt.

2.1 Grundlegendes Konzept

Die DLL exportiert drei Funktionen. Diese werden bei Bedarf vom MultiAnalyzer aufgerufen:

- Eine Funktion für die Initialisierung.
- Eine Funktion für das Aus/Ein-koppeln der Daten.
- Funktion die das Ende der Analyse ankündigt.



2.1.1 Die Initialisierung

Während der Aufnahme entsteht eine große Anzahl an Nachrichten. Um den Aufwand in der DLL selber klein zu halten, werden nur Daten an die DLL ausgekoppelt, die von Interesse sind. Die gewünschten Nachrichten/Daten müssen beim MultiAnalyzer registriert werden. Dieses geschieht nach dem Laden der DLL oder beim Start einer neuen Analyse-Instanz. Dazu ruft der MultiAnalyzer die Initialisierfunktion „`ThirdPartyComponent5V_Init()`“ auf. Die Rückgabe der Funktion enthält alle Daten über die DLL:

- Wurde die Initialisierung erfolgreich durchgeführt oder nicht
- Welche Protokolle werden unterstützt (TETRA-TMO, TETRA-DMO oder DMR)
- Der Name und die Versions-Information der DLL
- Welche Nachrichten werden gewünscht:
  - Die Protokoll-Schicht.
  - Die Empfangsrichtung (Downlink/Uplink).
  - Der Nachrichten Type (SDU,SDU Extension...).

2.1.2 Das Auskoppeln der Daten

Sobald eine aus der Initialisierung registrierte Nachricht empfangen wird, wird die DLL Funktion „`ThirdPartyComponent5V_Data()`“ mit den Nachrichten-Daten aufgerufen:

- Welcher Registrier-Type aus der Initialisierung hat zum Aufruf geführt.
- Richtung (Downlink/Uplink).
- Frame und Zeit Informationen.
- Kanal-Zustand (Unbenutzt, Kontroll-Kanal, Sprache [übertragend], Sprache [nicht übertragend], Kontroll-Kanal der Sprache, Daten-Kanal).
- Zell-Informationen (TETRA-TMO: Kanal-Nummer, ColorCode, MCC, MNC, LA).
- Wenn bekannt, die Herkunfts-Adresse.
- Wenn bekannt, die Ziel-Adresse.
- Gesetze Msc-Filter Einstellungen.
- Die Daten selber.

Die Funktion selber arbeitet blockierend. Es sollte darauf geachtet werden, dass keine zu langen Wartezeiten auftreten.

2.1.3 Das Einkoppeln der Daten

Das Einkoppeln von Daten in den MultiAnalyzer wird als Rückgabe-Wert des DLL Funktion „`ThirdPartyComponent5V_Data()`“ realisiert. Wenn die Funktion einen Wert von 0 zurück gibt, dann werden die Rückgaben ausgewertet. Alle anderen Werte führen dazu, dass die Rückgabe ignoriert wird.

Um analysierte Daten als Text für die MSC einzukoppeln, werden Callback Funkionen zur Verfügung gestellt. Diese werden beim DLL- Funktionsaufruf mit übergeben:

- Nachrichten Start (gibt an welcher Nachrichten-Name und in welcher Schicht die Nachricht dargestellt werden soll, es können multiple Nachrichten erzeugt werden).
- Nachrichten Text (Text der Nachrichten-Analyse)
- Nachrichten Ende (Beendet die Nachricht)

Das Einkoppeln von der Text-Analyse ist nicht zwingend nötig. Wenn Daten ausschließlich ausgekoppelt werden sollen (zum Beispiel Sprache/IP), werden die Funktionen nicht benutzt.

Als weiteres erwartet der MultiAnalyzer eine Rückgabe der DLL:

- Was soll genau mit den gegebenenfalls eingekoppelten Text passieren:
  - Text zusätzlich zur internen MultiAnalyzer Analyse anzeigen.
  - Text zusätzlich zur internen MultiAnalyzer Analyse anzeigen **und** interne MultiAnalyzer Filter-Einstellungen überschreiben.
  - Text anzeigen **und** die Darstellung der internen MultiAnalyzer Analyse unterdrücken.
  - Text anzeigen **und** die Darstellung der internen MultiAnalyzer Analyse für diese Nachricht **und** alle nachfolgenden unterdrücken.
  - Optional zusätzlichen Text anzeigen **und** einen veränderten, z.B. entschlüsselten E2E-SDS Inhalt, zur Analyse an den MultiAnalyzer zurückgeben.
- Eigene Filter-Einstellung zur Unterdrückung von Nachrichten in der MSC setzen.
- Eine Benutzer spezifische Nachrichten Type setzen für die gesonderte QoS Darstellung.

2.1.4 Das Ende der Analyse

Wenn der MultiAnalyzer die Analyse beendet (Aufnahme beendet, Datei vollständig gelesen), dann ruft er die DLL-Funktion „`ThirdPartyComponent5V_Close()`“ auf.

3 DLL Funktionen

3.1 ThirdPartyComponent5V_Init()

Die Deklaration lautet:

```
THIRD_PARTY_COMPONENT_5V int ThirdPartyComponent5V_Init( StructThirdPartyComponent5V_Init
*pstRegisterMessages );
```

Dabei ist:

- „`THIRD_PARTY_COMPONENT_5V`“ ein Makro, das den Import/Export der Funktion regelt.
- „`int`“ der Rückgabe-Wert der Funktion. Ist dieser ungleich **0**, so wird die DLL nicht benutzt und gleich wieder entladen.
- „`ThirdPartyComponent5V_Init`“ der Funktion-Name.
- „`StructThirdPartyComponent5V_Init *pstRegisterMessages`“ ein Zeiger auf eine Struktur, in dem die DLL-Funktion seine Rückgabe an den MultiAnalyzer schreibt. Also in der die zu registrierenden Nachrichten beschrieben werden.

Aufruf der Funktion:

Die Funktion wird nach dem Laden der DLL vom MultiAnalyzer aufgerufen, bzw. wenn eine neue Analyse Instanz geöffnet wird.

Rückgabe der Funktion:

Es wird ein Integer (`int`) zurückgeben. Ist dieser ***ungleich*** **0** ist die Registrierung fehlgeschlagen und der MultiAnalyzer entlädt die DLL wieder. Ist die Rückgabe jedoch gleich **0,** dann sind in der übergebenden Struktur „`StructThirdPartyComponent5V_Init *pstRegisterMessages`“ von der DLL die Nachrichten geschrieben, die beim MultiAnalyzer registriert werden sollen.

Anforderungen an die Implementierung:

Eingabe Überprüfungen:

- Überprüfung des Pointers auf „`pstRegisterMessages`“ auf nicht NULL.
- Überprüfung des Wertes „`StructThirdPartyComponent5V_Init::ulVersion`“ auf den Wert „`STRUCT_THIRDPARTYCOMPONENT5V_INIT__VERSION`“.

Ausgabe Überprüfungen:

- Den String „StructThirdPartyComponent5V_Init::strVersion“ nicht weiter als mit 128 Zeichen zu schreiben.
- Nicht mehr als 128 Nachrichten registrieren in „StructThirdPartyComponent5V_Init::stRegisterData“
- Der Wert „StructThirdPartyComponent5V_Init::ulRegisterCount“ ist auf die Anzahl der geschriebenen (registrierten Nachrichten) zu setzen.
- Für den Wert “StructThirdPartyComponent5V_RegData::eRegisterType” nur die “enum” benutzen die zum Protokoll „StructThirdPartyComponent5V_Init::ulProtokollSupport“ passen.

3.1.1 Die Struktur „StructThirdPartyComponent5V_Init“

Ein Pointer auf diese Struktur wird der Funktion „ThirdPartyComponent5V_Init()“ mit übergeben. Die Werte der Struktur sind mit 0 vorinitialisiert. Als einzige Ausnahme ist der Wert „StructThirdPartyComponent5V_Init::ulVersion“. Er gibt die Version der Struktur an. Der Wert ist auf „STRUCT_THIRDPARTYCOMPONENT5V_INIT__VERSION“ vom MultiAnalyzer gesetzt. Die DLL muss überprüfen, ob diese Versionsnummer kompatible zur eigenen kompilierten Version ist. Wenn das nicht der Fall ist, dann muss die Registrierung abgelehnt werden. Sprich die Funktion ThirdPartyComponent5V_Init()“ gibt einen Wert ***ungleich*** **0** zurück.

Alle anderen Werte der Struktur werden von der DLL gesetzt:

- „ulProtokollSupport“ zeigt dem MultiAnalyzer an, welcher Protokoll-Type unterstützt wird. Der Wert muss gesetzt werden. Die Werte sind im „enum EnumThirdPartyComponent5V_Protokoll“ definiert.
- „pDllIntance“ für interne Verwendung in der DLL, der Pointer wird in anderen Funktionen als Argument übergeben.
- „strVersion“ gibt einen NULL terminierten String zurück. Mit dem Namen und der Versions-Information der DLL. Der Wert ist optional und kann auch leer gelassen werden.
- „ulRegisterCount“ gibt an wie viele Einträge der nachfolgende Array „stRegisterData[128]“ gefüllt sind. Der Wert muss zwischen 1 und 128 liegen.
- „stRegisterData[128]“ ist Array von der Struktur „StructThirdPartyComponent5V_RegData“. Jeder Eintrag enthält den Datensatz einer zu registrierten Nachricht, siehe Beschreibung in Kapitel 3.1.2.

3.1.2 Die Struktur „StructThirdPartyComponent5V_RegData“

Diese Struktur wird beim Initialisieren benutzt. Sie wird von der Funktion `ThirdPartyComponent5V_Init()“` gefüllt. Ein Array von diesem Type ist in der Struktur „`StructThirdPartyComponent5V_Init`“ enthalten.

Je nach Nachrichten Type, der registriert werden soll, sind Werte der Struktur von Bedeutung und müssen gefüllt werden oder können leer gelassen werden.

- „`eRegisterType`“ dieser Wert definiert die Nachricht, die registriert werden soll. Die Werte sind in dem „`enum EnumThirdPartyComponent5V_RegType`“ definiert. Je nach eingesetzten Wert haben die Parameter „`ulType`“ und „`ulExtendedType[0,1]`“ dieser Struktur eine andere Bedeutung. Die Bedeutung ist im Kommentar hinter der Definition des „`enum`“ Wertes beschrieben.
- „`ulDirection`“ zeigt dem MultiAnalyzer an, für welche Empfangsrichtung die Registrierung gilt. Es ist ein Bitmap, verschiedene Werte können zusammensetzt werden:
  - 0x01 = Downlink
  - 0x02 = Uplink

  Es gibt Nachrichten, die die gleichen universellen Wert im Uplink und Downlink benutzen. Für diese Nachrichten kann dann der zusammengesetzte Wert 0x03 benutzt werden. Für Nachrichten, die je Richtung unterschiedlich sind, muss die Richtung explizit vorgeben sein.
- „`ulType`“ die Bedeutung dieses Wertes ist je nach „`eRegisterType`“ unterschiedlich:
  - Nicht benutzt
  - MLE/MM/CMCE/SNDCP - PDU type
  - SS (Supplementary Service) Type (z.B. DGNA)
  - MLE Protocol discriminator
  - PID (SDS Protocol Discriminator)
- „`ulExtendedType[0]`“ die Bedeutung dieses Wertes ist je nach „`eRegisterType`“ unterschiedlich:
  - Nicht benutzt
  - MM Sub PDU type (z.B. für U/D-OTAR)
  - SS PDU type (z.B. DGNA Assign)
- „`ulExtendedType[1]`“ wird zur Zeit nicht benutzt

3.1.3 Beispiel für die Funktion „ThirdPartyComponent5V_Init()“

```
THIRD_PARTY_COMPONENT_5V int ThirdPartyComponent5V_Init( StructThirdPartyComponent5V_Init *pstRegisterMessages )
{
    StructThirdPartyComponent5V_RegData *pstRegData = NULL;

    // Have result struct?
    if ( NULL == pstRegisterMessages )
    {   return -1;  }

    // Valid result struct?
    if ( STRUCT_THIRDPARTYCOMPONENT5V_INIT__VERSION != pstRegisterMessages->ulVersion )
    {   return -2;  }

    // Version/Company/Copyright string
    strcpy_s( pstRegisterMessages->strVersion, "ThirdPartyComponent5V, Version 0.0" );

    // Supported Protokoll, which type of protokoll data is supported
    pstRegisterMessages->ulProtokollSupport     = ThirdPartyComponent5V_TetraTmo;
    pstRegisterMessages→pDllInstance            = NULL;

    // Set data
    pstRegData = &pstRegisterMessages->stRegisterData[ pstRegisterMessages->ulRegisterCount ];
    pstRegData->eRegisterType     = ThirdPartyComponent5V_Mle_Pdu;
    pstRegData->ulDirection       = 0x01;     // DL
    pstRegData->ulType            = 2;        // PDU type: D-NWRK-BROADCAST
    pstRegisterMessages->ulRegisterCount++;

    // Set data
    pstRegData = &pstRegisterMessages->stRegisterData[ pstRegisterMessages->ulRegisterCount ];
    pstRegData->eRegisterType     = ThirdPartyComponent5V_Mm_Pdu;
    pstRegData->ulDirection       = 0x02;     // UL
    pstRegData->ulType            = 0;        // PDU type    : U-AUTHENTICATION
    pstRegData->ulExtendedType[0] = 1;        // PDU sub type: U-AUTHENTICATION RESPONSE
    pstRegisterMessages->ulRegisterCount++;

    // Set data
    pstRegData = &pstRegisterMessages->stRegisterData[ pstRegisterMessages->ulRegisterCount ];
    pstRegData->eRegisterType     = ThirdPartyComponent5V_Cmce_Pdu;
    pstRegData->ulDirection       = 0x03;     // DL/UL
    pstRegData->ulType            = 7;        // PDU type: U/D-SETUP
    pstRegisterMessages->ulRegisterCount++;

    // Set data
    pstRegData = &pstRegisterMessages->stRegisterData[ pstRegisterMessages->ulRegisterCount ];
    pstRegData->eRegisterType     = ThirdPartyComponent5V_Ss_Pdu;
    pstRegData->ulDirection       = 0x03;     // UL/DL
    pstRegData->ulType            = 22;       // DGNA
    pstRegData->ulExtendedType[0] = 7;        // SS-DGNA PDU type: ASSIGN
    pstRegisterMessages->ulRegisterCount++;

    // Set data
    pstRegData = &pstRegisterMessages->stRegisterData[ pstRegisterMessages->ulRegisterCount ];
    pstRegData->eRegisterType = ThirdPartyComponent5V_SdsType4_PID;
    pstRegData->ulDirection       = 0x03;     // DL+UL
    pstRegData->ulType            = 130;      // TL-SDS: Text message
    pstRegisterMessages->ulRegisterCount++;

    // Set data
    pstRegData = &pstRegisterMessages->stRegisterData[ pstRegisterMessages->ulRegisterCount ];
    pstRegData->eRegisterType     = ThirdPartyComponent5V_SdsType4_Content_TL;
    pstRegData->ulDirection       = 0x03;     // DL+UL
    pstRegisterMessages->ulRegisterCount++;

    // Nessages registred
    return 0;
}
```

3.2 ThirdPartyComponent5V_Data()

Die Deklaration lautet:

```
THIRD_PARTY_COMPONENT_5V int ThirdPartyComponent5V_Data( void *pDllInstance, const
StructThirdPartyComponent5V_Data *pstMessageData, StructThirdPartyComponent5V_Result
*pstAnalyseResult );
```

Dabei ist:

- „`THIRD_PARTY_COMPONENT_5V`“ ein Makro, das den Import/Export der Funktion regelt.
- „`int`“ der Rückgabe Wert der Funktion. Ist dieser ungleich **0**, so werden keine Daten wieder eingekoppelt.
- „`ThirdPartyComponent5V_Data`“ der Funktion-Name.
- „`pDllInstance`“ der Pointer aus „`StructThirdPartyComponent5V_Init::pDllInstance`“
- „`StructThirdPartyComponent5V_Data * pstMessageData`“ ein Zeiger auf eine Struktur die die Nachrichten Daten enthält.
- „`StructThirdPartyComponent5V_Result *pstAnalyseResult`“ ein Zeiger, in den die DLL-Funktion die Rückgabe schreibt (das Einkoppeln von Daten).

Aufruf der Funktion:

Die Funktion wird immer dann aufgerufen, wenn eine registrierte Nachricht empfangen wurde. Der Inhalt der Struktur „`StructThirdPartyComponent5V_Data * pstMessageData`“ enthält die Nachrichten Daten.

Rückgabe der Funktion:

Es wird ein Integer (`int`) zurückgeben. Ist dieser ***ungleich*** **0** wird ein geben-falls zurückgegebener Inhalt ignoriert. Ist die Rückgabe jedoch **0** dann werden die Daten in der übergebenden Struktur „`StructThirdPartyComponent5V_Result *pstAnalyseResult`“ die von der DLL gefüllt wurde plus den Text der über Call-back Funktionen eingekoppelt wurde vom MultiAnalyzer ausgewertet.

Anforderungen an die Implementierung:

Eingabe Überprüfungen:

- Überprüfung des Pointers auf „`pstMessageData`“ und „`pstAnalyseResult`“ auf nicht NULL.
- Überprüfung des Wertes „`StructThirdPartyComponent5V_Data::ulVersion`“ auf den Wert „`STRUCT_THIRDPARTYCOMPONENT5V_DATA__VERSION`“.
- Überprüfung des Wertes „`StructThirdPartyComponent5V_Result::ulVersion`“ auf den Wert „`STRUCT_THIRDPARTYCOMPONENT5V_RESULT__VERSION`“.

- Wenn Text eingekoppelt werden soll, prüfen der Call-Back Funktionen auf nicht NULL:
  - „StructThirdPartyComponent5V_Data::callback_PduOpen“
  - „StructThirdPartyComponent5V_Data::callback_PduClose“
  - „StructThirdPartyComponent5V_Data::callback_PduAddText“
  - (Zur Zeit immer NULL „StructThirdPartyComponent5V_Data::callback_ElementAddText“)

Ausgabe Überprüfungen:

- „ulAnalyseLength“ gibt an, wie viele **Bits** in „ucAnalyse“zurückgeben werden. Wird nur benutzt, wenn „eResult = ThirdPartyComponent5V_Result_AnalyseContent“ ist. Der Wert muss mindestens die Länge des PDU-Types der Nachricht haben und darf bis 4096 Oktetts (4096 * 8 Bits) gehen.
- Nicht mehr als 4096 **Bytes** für „StructThirdPartyComponent5V_Result::ucAnalyse“. Diese Daten werden **anstatt** der originalen Daten im MultiAnalyzer analysiert und ausgegeben.

3.2.1 Die Struktur „StructThirdPartyComponent5V_Data“

Ein Pointer auf diese Struktur wird der Funktion „ThirdPartyComponent5V_Data()“ mit übergeben. Der Inhalt der Struktur enthält die Nachricht selber, die vom MultiAnalyzer empfangen wurde, und weitere Information zum Kontext der Nachricht. Der Wert „StructThirdPartyComponent5V_Data::ulVersion“ gibt an welche Version die Struktur ist. Der Wert ist auf „STRUCT_THIRDPARTYCOMPONENT5V_DATA__VERSION“ vom MultiAnalyzer gesetzt. Die DLL muss überprüfen ob diese Versionsnummer kompatible zur eigenen kompilierten ist. Wenn das nicht der Fall ist dann muss die Analyse abgelehnt werden. Sprich die Funktion „ThirdPartyComponent5V_Data()“ gibt einen Wert *ungleich* **0** zurück.

Im ersten Teil sind die kopierten Werte aus der Registrierung zur Identifikation enthalten. Damit kann die DLL feststellen, um welche Nachricht es sich überhaupt handelt:

- „eRegisterType“ dieser Wert definiert die Nachricht, die registriert wurde. Die Werte sind in dem „enum EnumThirdPartyComponent5V_RegType“ definiert. Je nach eingesetzten Wert haben die Parameter „ulType“ und „ulExtendedType[0,1]“ dieser Struktur eine andere Bedeutung. Die Bedeutung ist im Kommentar hinter der Definition des „enum“ Wertes beschrieben.
- „ulDirection“ zeigt der DLL an, für welche Empfangsrichtung die Nachricht empfangen wurde.

  Wenn in der Registrierung der zusammengesetzte Wert 0x03 für Down- und Uplink benutzt wurde, dann steht hier nun die tatsächlich aufgetretene Richtung, also entweder 0x01 oder 0x02.
- „ulType“ die Bedeutung dieses Wertes ist je nach „eRegisterType“ unterschiedlich. Es ist im normalen Fall der Wert, der in der Registrierung gesetzt wurde.

  Für eine Registrierung wie SDS ohne PID Angabe, die aufgetretene PID, bei gnerellen PDU-Typen der aufgetretene PDU-Type.
- „ulExtendedType[0]“ die Bedeutung dieses Wertes ist je nach „eRegisterType“ unterschiedlich.
- „ulExtendedType[1]“ wird zur Zeit gegebenenfalls für Rückgaben benutzt.

Im zweiten Teil der Struktur werden Metadaten zur Nachricht geteilt. Damit kann die DLL den tieferen Kontext der Nachricht ermitteln:

- „`ulFrame[4]`“ die TETRA/DMR Zeit, in dem die Nachricht empfangen wurde ([0]=Hyperframe, [1]=MultiFrame, [2]=Frame, [3]=Slot)
- „`ullTime`“ und „`ulMilliSeconds`“ die Uhrzeit. Im Formart „__time64_t“ und 0 bis 999 Millisekunden.
- „`eChannel`“ vom Type „`enum EnumThirdPartyComponent5V_Channel`“ in diesem wird der logische Kanal-Zustand mitgeteilt. Für TETRA sind folgende Werte definiert:
  - „`ThirdPartyComponent5V_Channel_Idle`“ = Unallocated
  - „`ThirdPartyComponent5V_Channel_Control`“ = CCCH (MCCH/SCCH)
  - „`ThirdPartyComponent5V_Channel_Traffic`“ = TCH (Gespräch sendend/hörend)
  - „`ThirdPartyComponent5V_Channel_Traffic_Control`“ = SACCH
  - „`ThirdPartyComponent5V_Channel_Traffic_Idle`“ = FACCH
  - „`ThirdPartyComponent5V_Channel_Data`“ = PDCH
- „`ulCellData[8]`“ enthält Zell-Informationen. Für TETRA sind folgende Werte definiert:
  - „`ulCellData[0]`“ = Kanal-Nummer (0-3999)
  - „`ulCellData[1]`“ = Color Code (0...63)
  - „`ulCellData[2]`“ = MCC (1...1023)
  - „`ulCellData[3]`“ = MNC (1...16383)
  - „`ulCellData[4]`“ = LA (1...16383)
- „`ulSpeechData[16]`“ wenn ein Sprachruf läuft die Rufdaten. (Zur Zeit noch nicht benutzt)
- „`eOrginatorAddressType`“ und „`ulOrginatorAddress`“ falls bekannt die Adresse von dem die Nachricht verschickt wurde. „`eOrginatorAddressType`“ gibt den Type an, für TETRA ist der Type:
  - „`ThirdPartyComponent5V_Address_Normal`“ = SSI
  - „`ThirdPartyComponent5V_Address_Temporary`“ = USSI
  - „`ThirdPartyComponent5V_Address_Short`“ = Eventlabel
- „`eTerminatorAddressType`“ und „`ulTerminatorAddress`“ falls bekannt die Adresse an dem die Nachricht verschickt wurde. „`eTerminatorAddressType`“ gibt den Type an, für TETRA ist der Type:
  - „`ThirdPartyComponent5V_Address_Normal`“ = SSI
  - „`ThirdPartyComponent5V_Address_Temporary`“ = USSI
  - „`ThirdPartyComponent5V_Address_Short`“ = Eventlabel
- „`ullFilterMask`“ die gesetzte Filter-Maske aus der MSC. Dieser Wert kann gebraucht werden um zu entscheiden ob Text ausgegeben werden soll oder ob dieser unterdrückt werden soll.

Im dritten Teil werden Callback Funktionen zur Verfügung gestellt. Diese müssen benutzt werden, wenn Text in der MSC ausgeben werden soll. Der Text wird als Nachricht ausgeben. Damit die MSC weiß, an welcher Stelle eine Nachricht anfängt und wieder aufhört, gibt eine ein „Öffne-Nachricht“ und eine „Schließe-Nachricht“ Funktion. Innerhalb der Nachricht wird dann der Text selber ausgegeben. Es ist der DLL erlaubt multiple MSC Nachrichten einzufügen.

- „`pInternalParameter`“ interner MultiAnalyzer Pointer, wird den Callback-Funktionen als Argument mit übergeben. Ansonsten hat er keine Verwendung innerhalb der DLL und darf nicht verändert werden.
- „`callback_PduOpen`“ öffnet eine MSC Text-Nachricht.
- „`callback_PduClose`“ schließt eine MSC Text-Nachricht.
- „`callback_ElementAddText`“ fügt den Text der Nachricht hinzu. Darf nur zwischen öffnen und schließen benutzt werden. Kann aber dort beliebig häufig aufgerufen werden.
- „`callback_ElementAddText`“ wird zur Zeit nicht benutzt.

Alle Namen, Schichtdaten und Texte sind in „`wchar_t`“ Formart (16Bit Unicode) und sind NULL-Terminiert zu übergeben. Die Gesamtgröße darf bis zu 524.287 Zeichen betragen. Siehe mehr dazu im Kapitel 3.2.3.

Im vierten Teil wird die Nachricht selber binär ausgeliefert. Diese Daten können dann von der DLL gelesen und ausgewertet werden:

- „`ulDataLength`“ gibt die Nachrichtenlänge in **Bits** an.
- „`ucData[4096]`“ enthält binär die Nachricht. Das erste Nachrichten Bit ist im MSB des ersten Bytes gespeichert. Alle anderen Bits nachfolgend bis zum LSB, um im nächsten Byte im MSB fortgesetzt zu werden.

3.2.2 Die Struktur „StructThirdPartyComponent5V_Result“

Ein Pointer auf diese Struktur wird der Funktion „`ThirdPartyComponent5V_Data()`“ mit übergeben. In dieser Struktur werden Daten zurückgeschrieben, die zum Einkoppeln in den MultiAnalyzer gedacht sind. Der Wert „`StructThirdPartyComponent5V_Result::ulVersion`“ gibt die Version der Struktur an. Der Wert ist auf „`STRUCT_THIRDPARTYCOMPONENT5V_RESULT__VERSION`“ vom MultiAnalyzer gesetzt. Die DLL muss überprüfen ob diese Versionsnummer kompatible zur eigenen kompilierten Version ist. Wenn das nicht der Fall ist, dann muss die Analyse abgelehnt werden. Sprich die Funktion `ThirdPartyComponent5V_Data()`“ gibt einen Wert *ungleich* **0** zurück.

Alle anderen Werte der Struktur werden von der DLL gesetzt:

- „`eResult`“ beschreibt was zu tun ist. Die möglichen Werte sind im „`enum EnumThirdPartyComponent5V_Result`“ definiert:
  - „`ThirdPartyComponent5V_Result_Nothing`“ bedeutet, dass nichts weiter gemacht wird. Wenn Text hinzugefügt wurde, dann wird dieser **zusätzlich** zur internen MultiAnalyzer angezeigt, vorbehaltlich der Filterung. Ein optionaler Wert „`ullFilterMask`“ wird zum internen Wert vom MultiAnalyzer **hinzugefügt** (or) und durch den Filter behandelt.

- „`ThirdPartyComponent5V_Result_NothingButFilter`“: Wenn Text hinzugefügt wurde, dann wird dieser **zusätzlich** zur internen MultiAnalyzer Analyse angezeigt, vorbehaltlich der Filterung. Der Wert „`ullFilterMask`“ **ersetzt** den internen Wert vom MultiAnalyzer und wird durch den Filter behandelt.
- „`ThirdPartyComponent5V_Result_AnalyseContent`“: Wenn Text hinzugefügt wurde, dann wird dieser **zusätzlich** zur internen MultiAnalyzer Analyse angezeigt, vorbehaltlich der Filterung. Ein optionaler Wert „`ullFilterMask`“ wird zum internen Wert vom MultiAnalyzer **hinzugefügt** und durch den Filter behandelt.
  Die interne MultiAnalyzer Analyse nimmt aber als Daten für diese Nachricht **nicht** mehr die Originalen, **sondern** die Daten die in „`ulAnalyseLength`“ und „`ucAnalyse[4096]`“ zurückgebenden wurden. Zum Beispiel kann dieses eingesetzt werden, wenn von der DLL eine SDS End2End *entschlüsselt* wurde. Der entschlüsselte Teil wird zurückgeben um, zum Beispiel, den Inhalt von LIP oder Text Daten durch den MultiAnalzer analysiert darzustellen. Die Länge der Daten muss mindestens so lang sein, wie für die Identifizierung der Nachricht notwendig ist, also zum Beispiel der PDU-Type, PID, SS+SS PDU Type.
- „`ThirdPartyComponent5V_Result_Supress`“: Wenn Text hinzugefügt wurde, dann **ersetzt** dieser die interne MultiAnalyzer Analyse, vorbehaltlich der Filterung. Der Wert „`ullFilterMask`“ **ersetzt** den internen Wert vom MultiAnalyzer und wird durch den Filter behandelt.
- „`ThirdPartyComponent5V_Result_SupressFull`“: Wenn Text hinzugefügt wurde, dann **ersetzt** dieser die interne MultiAnalyzer Analyse, vorbehaltlich der Filterung. Es wird nicht nur der Text dieser Nachricht ersetzt **sondern** auch nachfolgende Schicht-Analysen. Der Wert „`ullFilterMask`“ **ersetzt** den internen Wert vom MultiAnalyzer und wird durch den Filter behandelt.

- „`ullFilterMask`“ fügt (optional, also ungleich 0) eigene Filter Eigenschaften hinzu, oder ersetzt interne, bzw. löscht damit diese. Die Filtereigenschaften sind Bitmaps und je nach Protokoll unterschiedlich. Wenn die gesetzte Filtereigenschaft zutrifft (*und*-Verknüpfung mit der Konfiguration aus der MSC) wird die Darstellung des gesamten Zweiges unterdrückt. Die Bitmap-Beschreibung sind im Handbuch des MultiAnalyzer zu finden, siehe Kapitel „5.5 MultiAnalyzerProto“.
- „`ulMsgType`“ ordnet optional ein Wert zwischen 1 und 128 der selbst analysierten Nachricht zu. Dieser gilt als QoS-Nachrichten Type. Dabei handelt es sich um die „Benutzer spezifische Nachricht“ zwischen 1 und 128. Diese kann zum Beispiel für die Last-Anzeige des MCCH/SCCH oder in den Benutzerspezifischen Anzeigen genutzt werden.
- „`ulAnalyseLength`“ und „`ucAnalyse[4096]`“ enthält binär die alternative Nachrichtenformationen. Also nur dann wenn der Wert „`eResult = ThirdPartyComponent5V_Result_AnalyseContent`“ ist. Das erste Bit ist im MSB des ersten Bytes gespeichert. Alle anderen Bits nachfolgend bis zum LSB, um im nächsten Byte im MSB fortgesetzt zu werden.

3.2.3 Die Text Callback Funktionen

Die Text Callback werden in der Struktur „`StructThirdPartyComponent5V_Data`“ der DLL Funktion „`ThirdPartyComponent5V_Data()`“ mitgeteilt. Sofern die DLL der MSC Textausgaben hinzufügen oder gar die Ausgabe des MultiAnalyzer ersetzen werden soll, müssen diese Funktionen benutzt werden.

3.2.3.1 MSC Nachrichten Filtern

Die Entscheidung, ob die DLL Text hinzufügen will oder nicht kann anhand der Nachricht, des Inhaltes, der Metadaten oder dem gesetzten MSC-Filteroptionen getroffen werden. Alle diese Daten befinden sich in der übergebenen Struktur „`StructThirdPartyComponent5V_Data`“.

Wenn Text hinzugefügt wurde, dann wird in der Struktur „`StructThirdPartyComponent5V_Result`“ der Wert „`eResult`“ gesetzt. Dieser gibt an, wie mit dem Text weiter zu verfahren ist. Zusätzlich kann mit dem Wert „`ullFilterMask`“ in der Struktur die internen MultiAnalyzer Filter Eigenschaften ergänzt oder ersetzt werden.

Zum Filtern des Textes stehen der DLL zwei mögliche Optionen zur Verfügung:

1. Keinen eigenen Text einfügen. Damit wird nur die interne MultiAnalyzer Analyse angezeigt. Die Entscheidung trifft die DLL nach eigenen Kriterien. Zum Beispiel werden die vom Benutzer eingestellten Filtereinstellungen aus der MSC im Wert „`StructThirdPartyComponent5V_Data::ullFilterMask`“ übergeben. Sie können zur Entscheidung in der DLL beitragen ob zusätzlicher Text eingefügt werden soll oder nicht. Die Bitmaps, aus den der Wert besteht, sind im Handbuch des MultiAnalyzer zu finden, siehe Kapitel „5.5 MultiAnalyzerProto“.
2. Die Filterung des DLL-Textes dem generellen Filter zu überlassen. Der zusätzliche DLL Text wird hierbei immer eingefügt, je nachdem ob das Filterkriterium erfüllt ist, wird der Text im Nachgang herausgefiltert.
   ***Achtung**: Es wird immer der gesamte Zweig (alle vorherigen Nachrichten) und nicht nur der Text selbst gefiltert.*
   Das Filterkriterium kann von der DLL beeinflusst werden. Dazu wird der Wert „`StructThirdPartyComponent5V_Result::ullFilterMask`“ gesetzt. Je nach dem Wert „`StructThirdPartyComponent5V_Result::eResult`“ wird damit das Filterkriterium **ergänzt** oder **ersetzt**. Das Filterkriterium ist erfüllt, *und damit wird der Text+Zweig gefiltert*, wenn das Filterkriterium „*und*-Verknüpft“ mit dem Benutzer eingestellten Filtereinstellungen aus der MSC wahr wird.

3.2.3.2 MSC Nachrichten-Format

Damit die MultiAnalyzerMsc eine Nachricht darstellen kann, muss sie zwei Dinge wissen. Einmal den Nachrichten-Name und als zweites die Schicht, in der die Nachricht dargestellt werden soll. Wenn zudem mehrere Nachrichten dargestellt werden sollen, dann muss die Msc auch wissen, wann die alte endet und wo die neue beginnt. Zwischen dem Start und dem Ende einer Nachricht steht dann der eigentliche Text.

Das bedeutet, dass zu aller erst die Nachricht beginnen muss. Diese wird mit der Callback-Funktion „`StructThirdPartyComponent5V_Data::callback_PduOpen`“ realisiert. Sie enthält als zu übergebende Argumente den Nachrichten-Namen, die Schicht-Darstellungs-Position und ein Verwaltungs-Pointer für den MultiAnalyzer:

1. Der Nachrichten Name, ein NULL terminierter String, im „`wchar_t`“ Formart (16Bit Unicode). Der Nachrichtenname kann freigewählt werden, er sollte nicht länger als 32 Zeichen lang sein.
2. Die Schicht in der die Nachricht dargestellt werden soll. Je nach Protokoll sind bestimmte Schichten vom MultiAnalyzer fest unveränderlich vorgegeben. Die Nachricht kann innerhalb einer Schicht dargestellt werden oder schichtübergreifend. Die gewünschte Darstellungsposition wird als zweites Argument als ein NULL terminierter String, im „`wchar_t`“ Formart (16Bit Unicode) übergeben:
   - Der Inhalt des Strings ist der exakte Schichtname (groß/klein Schreibung beachten), wenn die Nachricht innerhalb einer Schicht dargestellt werden soll.
     Beispiel: „`MLE`“
   - Der Inhalt des Strings sind die beiden exakten Schichtnamen (groß/klein Schreibung beachten) beginnend mit der höheren Schicht, getrennt durch ein Komma ',' mit der tieferen Schicht.
     Beispiel: „`MLE,LLC`“ oder „`TL-SDS/SNDCP/SS,MLE`“
3. Damit der MultiAnalyzer den Text zuordnen kann, muss noch ein interner Verwaltungs-Pointer aus der Struktur „`StructThirdPartyComponent5V_Data::pInternalParameter`“ mit übergeben werden.

Danach wird der eigentliche Text der Nachricht eingefügt. Dazu wird die Callback-Funktion „`StructThirdPartyComponent5V_Data::callback_PduAddText`“ benutzt. Diese Funktion kann multiple aufgerufen werden und hängt jeweils den neuen Text an. Sie enthält als übergebendes Argumente den Text und ein Verwaltungs-Pointer für den MultiAnalyzer:

1. Der Text selber ist ein in NULL terminierter String, im „`wchar_t`“ Formart (16Bit Unicode).
2. Damit der MultiAnalyzer den Text zuordnen kann, muss noch ein interner Reference Pointer aus der Struktur „`StructThirdPartyComponent5V_Data::pInternalParameter`“ mit übergeben werden.

Zum Abschließen der Nachricht wird die Callback-Funktion „`StructThirdPartyComponent5V_Data::callback_PduClose`“ benutzt. Sie enthält als übergebendes Argumente ein Verwaltungs-Pointer für den MultiAnalyzer („`StructThirdPartyComponent5V_Data::pInternalParameter`“).

3.2.4 Beispiel für die Funktion „ThirdPartyComponent5V_Data()“

Dieses Beispiel benutzt Makros aus dem Kapitel 3.4. Zum Beispiel um Daten zu lesen, oder Text einzukoppeln.

```
THIRD_PARTY_COMPONENT_5V int ThirdPartyComponent5V_Data( const StructThirdPartyComponent5V_Data *pstMessageData,
StructThirdPartyComponent5V_Result *pstAnalyseResult )
{
    __Mas_DEFINE_ulReadIndex; // Macro, see chapter 3.4

    // Have result struct?
    if ( NULL == pstMessageData )
    {   return -1;  }

    // Have result struct?
    if ( NULL == pstAnalyseResult)
    {   return -2;  }

    // Valid result struct?
    if ( STRUCT_THIRDPARTYCOMPONENT5V_DATA__VERSION != pstMessageData->ulVersion )
    {   return -2;  }

    // Valid result struct?
    if ( STRUCT_THIRDPARTYCOMPONENT5V_RESULT__VERSION != pstAnalyseResult->ulVersion )
    {   return -1;  }


    // Ensure nothing returned
    pstAnalyseResult->eResult           = ThirdPartyComponent5V_Result_Nothing;
    pstAnalyseResult->ullFilterMask     = 0;
    pstAnalyseResult->ulAnalyseLength   = 0;

    // Depending from message type write text for MSC:
    // For "__Mas_PduOpen_sprintf", "__Mas_PduText_sprintf", "__Mas_PduText_strcpy", "__Mas_Read8", see chapter 3.4
    switch ( pstMessageData->eRegisterType )
    {
        case ThirdPartyComponent5V_Mle_Pdu:
            __Mas_PduOpen_sprintf( L"MLE", L"MY-MLE-PDU" );
            __Mas_PduText_sprintf( L"PDU-Type: %u!\n", pstMessageData->ulType );
            __Mas_PduText_sprintf( L"Hello World!\n" );
            __Mas_PduCose();
        break;
        case ThirdPartyComponent5V_Mm_Pdu:
            __Mas_PduOpen_sprintf( L"MM/CMCE", L"MY-MM-PDU" );
            __Mas_PduText_sprintf( L"PDU-Type: %u!\n", pstMessageData->ulType );
            __Mas_PduCose();
        break;
        case ThirdPartyComponent5V_Cmce_Pdu:
            __Mas_PduOpen_sprintf( L"MM/CMCE", L"MY-CMCE-PDU" );
            __Mas_PduText_sprintf( L"PDU-Type: %u!\n", pstMessageData->ulType );
            __Mas_PduCose();
        break;

        case ThirdPartyComponent5V_Ss_Pdu:
            __Mas_PduOpen_sprintf( L"MM/CMCE", L"MY-CMCE-PDU" );
            __Mas_PduText_sprintf( L"SS: %u!\n", pstMessageData->ulType );
            __Mas_PduText_sprintf( L"PDU-Type: %u!\n", pstRegData->ulExtendedType[0] );
            __Mas_PduCose();
        break;
        case ThirdPartyComponent5V_SdsType4_PID:
        {
            unsigned char ucPID = __Mas_Read8(8); // Read PID
            if ( 130 == ucPID )
            {  // Message data contains PID, TL-Header and content
                __Mas_PduOpen_sprintf( L"TL-SDS/SNDCP/SS", L"MY-SDS-TEXT PDU" );
                __Mas_PduText_sprintf( L"PID: %u!\n", ucPID );
                __Mas_PduCose();
                // Replace MultiAnalyzer TL-SDS and TEXT message analyse:
                pstAnalyseResult->eResult = ThirdPartyComponent5V_Result_SupressFull;
            }
            else
            { // Do nothing, no external analyse
                return -10;
            }
        break;

        case ThirdPartyComponent5V_SdsType4_Content_TL:
            if ( 130 == pstMessageData->ulType )
            {   // Message data contains ONLY content, PID is given in "pstMessageData->ulType"
                __Mas_PduOpen_sprintf( L"TL-SDS/SNDCP/SS", L"MY-TL-SDS-TEXT PDU" );
                __Mas_PduText_sprintf( L"PID: %u!\n", pstMessageData->ulType );
                __Mas_PduCose();
                // Replace MultiAnalyzer TEXT message analyse and add additional filter:
```

```
                pstAnalyseResult->eResult = ThirdPartyComponent5V_Result_Supress;
                pstAnalyseResult->ullFilterMask = 0x20000; // TetraTmo_Supress_UserDefined_1
            }
            else
            { // Do nothing, no external analyse
                return -10;
            }
        break;
        default:
        return -11;
    } // switch ( pstMessageData->eRegisterType )
}
```

3.3 ThirdPartyComponent5V_Close()

Die Deklaration lautet:

```
THIRD_PARTY_COMPONENT_5V int ThirdPartyComponent5V_Close(void *pDllInstance);
```

Dabei ist:

- „`THIRD_PARTY_COMPONENT_5V`“ ein Makro, das den Import/Export der Funktion regelt.
- „`int`“ der Rückgabe Wert der Funktion.
- „`ThirdPartyComponent5V_Close`“ der Funktion-Name.
- „`pDllInstance`“ der Pointer aus „`StructThirdPartyComponent5V_Init::pDllInstance`“

Aufruf der Funktion:

Die Funktion wird vor dem Entladen der DLL vom MultiAnalyzer aufgerufen. Bzw. wenn eine Analyse Instanz geschlossen wird.

Rückgabe der Funktion:

Es wird ein Integer (`int`) zurückgeben. Der Wert wird ignoriert.

Anforderungen an die Implementierung:

Keine.

3.4 Hilfreiche Makros und Funktionen

Um die Nachrichten-Daten zu verarbeiten, also zu lesen, können Funktionen und Makros benutzt werden. Da die PDU-Daten häufig verschiedene Bit Längen aufweisen, muss bitgenau über Byte Grenzen gelesen werden.

Des weiteren sind für die MSC Ausgabe Text-Operationen notwendig. Also hinzufügen, Argumente ausgeben usw… Dieses kann zusammen gefasst und das Ergebnis den Callback-Funktionen übergeben werden.

3.4.1 Lese Makros und Funktionen

Es werden drei Funktionen definiert. Diese unterscheiden sich nur darin, wie viele Bits sie maximal lesen können. Dieses ermöglicht das Einsparen von unnötigen Operationen. Die Funktionen arbeiten so, dass unabhängig von der genauen Bit-Position innerhalb mehrere Bytes der Wert zusammen gefasst wird (or). Dann wird der zusammengefasste Wert an die richtige Bit-Position geschoben und Bits außerhalb des Wertes aus-maskiert.

Die Funktionen brauchen als Parameter die Anzahl der Bits die gelesen wurden, der Speicherbereich in dem die Daten stehen, die maximal Anzahl der vorhanden Bits im Speicherbereich (um ein überlesen zu vermeiden), die aktuelle Lese-Position, und ein Rückgabe-Pointer ob erfolgreich gelesen werden konnte. Als direkte Rückgabe, im Erfolgsfall, wird der gelesene Wert zurückgegeben.

```
// Read max 1 to 8 bits of data and return result, increase pulReadBitPos
unsigned char __Mas_Read_Bits8_Function( const unsigned char ucBits, const unsigned char *pucSource, const unsigned int
ulMaxBitLength, unsigned int *pulReadBitPos, bool *pbSuccess )
{
    unsigned int        ulValue         = 0;
    unsigned int        ulBytePos       = ((*pulReadBitPos)  >> 3);
    const unsigned int  ulMaxBytePos    = (ulMaxBitLength+7) >> 3 ;
    unsigned char       ucDest          = 0;
    const unsigned char ucReverseBits   =  (8 - ucBits);
    const unsigned char ucBitPos        = ((8 - ((*pulReadBitPos) &  0x7)) + ucReverseBits);

    // Can read?
    if ( 8 < ucBits                                 )
    {   assert( 8             >= ucBits                   );  if (pbSuccess) { (*pbSuccess) = false; } return 0;   }
    if ( ulMaxBitLength < ((*pulReadBitPos)+ucBits) )
    {   assert( ulMaxBitLength >= ((*pulReadBitPos)+ucBits) ); if (pbSuccess) { (*pbSuccess) = false; } return 0;   }

    // Read data (16 bits, so every possible parts of 8 bits are included for 8 bit value)
                                    ulValue |=  ((unsigned int)pucSource[ulBytePos++] << 8);
    if ( ulMaxBytePos >= ulBytePos )    ulValue |=  ((unsigned int)pucSource[ulBytePos++]     );

    // Shift bits to LSB
    ucDest       = (unsigned char)(ulValue >> ucBitPos);
    // Mask only wanted bits
    ucDest      &= (0xFF >> ucReverseBits);

    // Increase bits
    (*pulReadBitPos)    += ucBits;

    if (pbSuccess) { (*pbSuccess) = true; }
    return ucDest;
}
```

```
// Read max 1 to 16 bits of data and return result, increase pulReadBitPos
unsigned short __Mas_Read_Bits16_Function( const unsigned char ucBits, const unsigned char *pucSource, const unsigned int
ulMaxBitLength, unsigned int *pulReadBitPos, bool *pbSuccess)
{
    unsigned int        ulValue         = 0;
    unsigned int        ulBytePos       = ((*pulReadBitPos)  >> 3);
    const unsigned int  ulMaxBytePos    = (ulMaxBitLength+7) >> 3 ;
    unsigned short      usDest          = 0;
    const unsigned char ucReverseBits   = (16 - ucBits);
    const unsigned char ucBitPos        = ((8 - ((*pulReadBitPos) &  0x7)) + ucReverseBits);

    // Can read?
    if ( 16 < ucBits                                )
    {   assert( 16            >= ucBits                   ); if (pbSuccess) { (*pbSuccess) = false; } return 0;   }
    if ( ulMaxBitLength < ((*pulReadBitPos)+ucBits) )
    {   assert( ulMaxBitLength >= ((*pulReadBitPos)+ucBits) ); if (pbSuccess) { (*pbSuccess) = false; } return 0;   }

    // Read data (24 bits, so every possible parts of 8 bits are included for 16 bit value)
                                    ulValue |=  ((unsigned int)pucSource[ulBytePos++] << 16);
    if ( ulMaxBytePos >= ulBytePos )    ulValue |=  ((unsigned int)pucSource[ulBytePos++] <<  8);
    if ( ulMaxBytePos >= ulBytePos )    ulValue |=  ((unsigned int)pucSource[ulBytePos++]      );

    // Shift bits to LSB
    usDest       = (unsigned short)(ulValue >> ucBitPos);
    // Mask only wanted bits
    usDest      &= (0xFFFF >> ucReverseBits);

    // Increase bits
    (*pulReadBitPos)    += ucBits;

    if (pbSuccess) { (*pbSuccess) = true; }
    return usDest;
}


// Read max 1 to 32 bits of data and return result, increase pulReadBitPos
unsigned int __Mas_Read_Bits32_Function( const unsigned char ucBits, const unsigned char *pucSource, const unsigned int
ulMaxBitLength, unsigned int *pulReadBitPos, bool *pbSuccess)
{
    unsigned long long  ullValue        = 0;
    unsigned int        ulDest          = 0;
    unsigned int        ulBytePos       = ((*pulReadBitPos)  >> 3);
    const unsigned int  ulMaxBytePos    = (ulMaxBitLength+7) >> 3 ;
    const unsigned char ucReverseBits   = (32 - ucBits);
    const unsigned char ucBitPos        = ((8 - ((*pulReadBitPos) &  0x7)) + ucReverseBits);

    // Can read?
    if ( 32 < ucBits                                )
    {   assert( 32            >= ucBits                   ); if (pbSuccess) { (*pbSuccess) = false; } return 0;   }
    if ( ulMaxBitLength < ((*pulReadBitPos)+ucBits) )
    {   assert( ulMaxBitLength >= ((*pulReadBitPos)+ucBits) ); if (pbSuccess) { (*pbSuccess) = false; } return 0;   }

    // Read data (40 bits, so every possible parts of 8 bits are included for 32 bit value)
                                    ullValue    |=  ((unsigned long long)pucSource[ulBytePos++] << 32);
    if ( ulMaxBytePos >= ulBytePos )    ullValue    |=  ((unsigned long long)pucSource[ulBytePos++] << 24);
    if ( ulMaxBytePos >= ulBytePos )    ullValue    |=  ((unsigned long long)pucSource[ulBytePos++] << 16);
    if ( ulMaxBytePos >= ulBytePos )    ullValue    |=  ((unsigned long long)pucSource[ulBytePos++] <<  8);
    if ( ulMaxBytePos >= ulBytePos )    ullValue    |=  ((unsigned long long)pucSource[ulBytePos++]      );

    // Shift bits to LSB
    ulDest       = (unsigned int)(ullValue >> ucBitPos);
    // Mask only wanted bits
    ulDest      &= (0xFFFFFFFF >> ucReverseBits);

    // Increase bits
    (*pulReadBitPos)    += ucBits;

    if (pbSuccess) { (*pbSuccess) = true; }
    return ulDest;
}
```

Für die Lese-Makros werden folgende Annahmen angenommen:

- Der Speicherbereich und die maximale Anzahl an Bits ändert sich nicht, sie sind immer diejenigen aus der Struktur „`StructThirdPartyComponent5V_Data *pstMessageData`“.
  Der Name für den Pointer wird in einem Makro eingesetzt:
  ```
  #define __MSGDATA_POINTER pstMessageData
  ```
- Es gibt ein fortlaufenden Leseprozess, sprich die Bit-Leseposition wird beim jeden Lesen hochgezählt. Daraus folgt, dass in der Daten-Analyse-Funktion zwei Variablen definiert werden müssen, eine die die Lese-Position enthält und eine die für die Erfolgsrückgabe der Funktion genutzt werden kann:
  - Die Variablen-Namen werden in Makros definiert:
    ```
    #define __MAS_READ_INDEX ulReadIndex
    #define __MAS_READ_SUCCESS bReadSuccess
    ```
  - Diese Variablen müssen in der Funktion definiert werden:
    ```
    #define __Mas_DEFINE_ulReadIndex unsigned int __MAS_READ_INDEX = 0; bool __MAS_READ_SUCCESS = false
    ```
- Im Fehler Fall, also beim Lesen, ist noch zu definieren was zu dann tun ist, hier ein `assert`(), es könnte aber auch ein „`throw`“ oder ein „`goto`“ sein:
  ```
  #define __MAS_READ_FAIL_ACTION assert( !"Read more bits than present" );
  ```

Nun können die Makros zum Lesen definiert werden:

Sie benutzen die Funktionen, definierten Makros von oben und statischen Variablen-Namen:

```
// Read wanted amount of bits, call "__MAS_READ_FAIL_ACTION" if fail
#define __Mas_Read8(  nBits ) __Mas_Read_Bits8_Function(  nBits, (__MSGDATA_POINTER)->ucData, (__MSGDATA_POINTER)→ulDataLength,\
    &(__MAS_READ_INDEX), &(__MAS_READ_SUCCESS) ); if ( !(__MAS_READ_SUCCESS) ) { __MAS_READ_FAIL_ACTION; }
#define __Mas_Read16( nBits ) __Mas_Read_Bits16_Function( nBits, (__MSGDATA_POINTER)->ucData, (__MSGDATA_POINTER)→ulDataLength,\
    &(__MAS_READ_INDEX), &(__MAS_READ_SUCCESS) ); if ( !(__MAS_READ_SUCCESS) ) { __MAS_READ_FAIL_ACTION; }
#define __Mas_Read32( nBits ) __Mas_Read_Bits32_Function( nBits, (__MSGDATA_POINTER)->ucData, (__MSGDATA_POINTER)→ulDataLength,\
    &(__MAS_READ_INDEX), &(__MAS_READ_SUCCESS) ); if ( !(__MAS_READ_SUCCESS) ) { __MAS_READ_FAIL_ACTION; }

// Read wanted amount of bits, user defined behaviour of "__MAS_READ_FAIL_ACTION"
#define __Mas_Watch8(  nBits ) __Mas_Read_Bits8_Function(  nBits, (__MSGDATA_POINTER)->ucData, (__MSGDATA_POINTER)→ulDataLength,\
    &(__MAS_READ_INDEX), &(__MAS_READ_SUCCESS) )
#define __Mas_Watch16( nBits ) __Mas_Read_Bits16_Function( nBits, (__MSGDATA_POINTER)->ucData, (__MSGDATA_POINTER)→ulDataLength,\
    &(__MAS_READ_INDEX), &(__MAS_READ_SUCCESS) )
#define __Mas_Watch32( nBits ) __Mas_Read_Bits32_Function( nBits, (__MSGDATA_POINTER)->ucData, (__MSGDATA_POINTER)→ulDataLength,\
    &(__MAS_READ_INDEX), &(__MAS_READ_SUCCESS) )
```

Die Anwendung in der Funktion:

```
 __Mas_DEFINE_ulReadIndex;                  // Define macros variable

unsigned char  ucBit   = __Mas_Read8(   1 ); // Read one bit
unsigned char  ucByte  = __Mas_Read8(   8 ); // Read eight bits
unsigned short usMany1 = __Mas_Read16( 14 ); // Read 14 bits
unsigned int   ulMany2 = __Mas_Read32( 24 ); // Read 24 bits
```

Es wurden 1+8+14+24=47 Bits gelesen.

Es kann überprüft werden ob alle Bits gelesen wurden:

```
assert( (__MSGDATA_POINTER)->ulDataLength) == ulReadIndex );
```

3.4.2 Text Makros und Funktionen

Bei Text-Operationen im Speicher muss darauf geachtet werden, dass der Speicherbereich nicht überschrieben wird. Dazu können Makros benutzt werden. Die Makros kapseln die benötigten Sicherheitsüberprüfungen:

- Es wird davon ausgegangen das für den Schreib-Speicher kein Pointer benutzt wird, sondern das das String-Array bekannt ist.
  Um die Größe des String-Arrays festzustellen werden Makros definiert:
```
#define __Sec_sizeof( sDestString )             ((size_t)sizeof(sDestString)/(size_t)sizeof((sDestString)[0]))
#define __Sec_sizeofN( sDestString, nMaxLength ) __Sec_Min(__Sec_sizeof(sDestString), nMaxLength))
```
  Sofern nicht min und „max“ und „min“ bekannt sind werden diese unter eigenen Namen definiert:
```
#ifndef min
    #define __Sec_Min( a, b )           (((a) < (b)) ? (a) : (b))
#else  // min
    #define __Sec_Min                   min
#endif // min
#ifndef max
    #define __Sec_Max( a, b )           (((a) > (b)) ? (a) : (b))
#else  // min
    #define __Sec_Max                   max
#endif // min
```

- Als weiteres muss bei einem String die NULL-Terminierung sichergestellt werden.
  Um die Terminierung sicherzustellen wird ein Makro definiert:
```
#define __Sec_CloseN(sDestString, nMaxLength) sDestString[__Sec_Max(__Sec_sizeofN(sDestString,nMaxLength),1)-1] = 0;
```

- Um ein String zu löschen wird ein Makro definiert:
```
#define __Sec_Empty( sDestString )                      (sDestString)[0] = 0
```

- Um heraus zu finden wie viele Zeichen im String sind wird ein Makro definiert:
```
#define __Sec_Strnlen( sDestString, nMaxLength )\
     wcsnlen_s( sDestString, __Sec_sizeofN(sDestString, nMaxLength) )
```

- Um heraus zu finden wie viele Zeichen noch in den String passen wird ein Makro definiert:
```
#define __Sec_Remaining_strlen( sDestString, nMaxLength )\
(__Sec_sizeofN(sDestString, nMaxLength) - __Sec_Strnlen(sDestString, nMaxLength))
```

- Um eine formatierte Ausgabe zu einem String hinzuzufügen wird ein Makro definiert:
```
#define __Sec_Add_sprintf( sDestString, nMaxLength, sFormat, ... )\
     if ( (1 < __Sec_Remaining_strlen(sDestString, nMaxLength)) )\
     {   (void)swprintf_s( &sDestString[__Sec_Strnlen(sDestString, nMaxLength)], __Sec_Remaining_strlen(sDestString,\
                           nMaxLength), sFormat, __VA_ARGS__ );\
     }\
     __Sec_CloseN( sDestString, nMaxLength )
```

- Auch für das Kopieren der Strings werden Makros definiert:
```
#define __Sec_strncpy( sDestString, sSrcString, nMaxLength ) \
 (void)wcsncpy_s( sDestString, sSrcString , __Sec_sizeofN(sDestString, nMaxLength) );\
 __Sec_CloseN( sDestString, nMaxLength )

#define __Sec_strcpy( sDestString, sSrcString ) __Sec_strncpy( sDestString, sSrcString, __Sec_sizeof(sDestString) )
```

3.4.3 Kombinierte Text und MSC-Einkoppel Makros

Um Text einfach zu erstellen und einzukoppeln, können als erstes die Text-Makros benutzt werden. Das Ergebnis wird dann an die Callback-Funktionen zum Einkoppeln übergeben:

- Um Temporär Strings zu erstellen wird ein Speicherbereich definiert:
```
wchar_t g_strTextBuffer[4096] = { 0 };
```
- Die CallBack-Funktionen sind unveränderlich in der Struktur „`StructThirdPartyComponent5V_Data *pstMessageData`“ übergeben worden.
  Der Name für den Pointer wird in einem Makro eingesetzt:
```
#define __MSGDATA_POINTER                       pstMessageData
```
- Um eine Nachricht zu beginnen müssen die Darstellungs-Schicht in der MSC und der Nachrichten-Name generiert werden. Alle anderen Parameter („`g_strTextBuffer`“ und „`__MSGDATA_POINTER`“) bleiben gleich, darum müssen mindestens zwei Argumente übergeben werden:
  - Die MSC Darstellungs-Schicht ist ein statistischer String.
  - Da der Nachrichten unter Umständen dynamisch angepasst wird, z.B. *NAME*, *NAME* (**A**), *NAME* (**B**), wird für die Erzeugung die formatierte Ausgabe gewählt.
```
#define __Mas_PduOpen_sprintf( sLayer, sFormat, ... )\
if ( (__MSGDATA_POINTER) && (__MSGDATA_POINTER)->callback_PduOpen )\
{   __Sec_Empty( g_strTextBuffer );\
    __Sec_Add_sprintf( g_strTextBuffer, __Sec_sizeof(g_strTextBuffer), sFormat, __VA_ARGS__ );\
    (__MSGDATA_POINTER)->callback_PduOpen( g_strTextBuffer, sLayer, (__MSGDATA_POINTER)->pInternalParameter );\
} else assert( !"__Mas_PduOpen_sprintf: No callback!" )
```
  Beispiele:
```
__Mas_PduOpen_sprintf(L"TL-SDS/SNDCP/SS", L"MY-SDS-DATA" );
__Mas_PduOpen_sprintf(L"TL-SDS/SNDCP/SS", L"MY-SDS-DATA (%u)", ucPID );
```
- Um den Nachrichten Text einzukoppeln wird die formatierte Ausgabe gewählt. Alle anderen Parameter („`g_strTextBuffer`“ und „`__MSGDATA_POINTER`“) bleiben gleich, darum muss mindestens ein Argument übergeben werden:
```
#define __Mas_PduText_sprintf( sFormat, ... ) \
if ( (__MSGDATA_POINTER) && (__MSGDATA_POINTER)->callback_PduAddText )\
{   __Sec_Empty( g_strTextBuffer );\
    __Sec_Add_sprintf( g_strTextBuffer, __Sec_sizeof(g_strTextBuffer), sFormat, __VA_ARGS__ );\
    (__MSGDATA_POINTER)->callback_PduAddText( g_strTextBuffer, (__MSGDATA_POINTER)->pInternalParameter );\
} else assert( !"__Mas_PduText_sprintf: No callback!" )
```
  Beispiele:
```
__Mas_PduText_sprintf( L"Full IP data:\n" );
__Mas_PduText_sprintf( L"Dump data: %u bits\n", (__MSGDATA_POINTER)→ulDataLength );
```
  Um einfachen Text zufügen kann auch eine unformatierte Ausgabe benutzt werden:
```
#define __Mas_PduText_strcpy( sString ) \
if ( (__MSGDATA_POINTER) && (__MSGDATA_POINTER)->callback_PduAddText )\
{   (__MSGDATA_POINTER)->callback_PduAddText( sString, (__MSGDATA_POINTER)->pInternalParameter );\
} else assert( !"__Mas_PduText_strcpy: No callback!" )
```
  Beispiel:
```
___Mas_PduText_strcpy( L"Full IP data:\n" );
```
- Für das schließen der Nachricht werden keine Parameter benötigt:
```
#define __Mas_PduCose() \
if ( (__MSGDATA_POINTER) && (__MSGDATA_POINTER)->callback_PduClose )\
{    (__MSGDATA_POINTER)->callback_PduClose( (__MSGDATA_POINTER)->pInternalParameter ); }\
else assert( !"__Mas_PduCose: No callback!" )
```

3.4.4 Weitere Beispiele

Im SDK Verzeichnis befinden sich drei Projekte:

- „ThirdPartyComponent5V“: Zeigt auf wie das Einkoppeln und ersetzten von Text und neuen Nachrichten funktioniert
- “ThirdPartyComponent_IpExport”: Koppelt IP Daten aus. Schreibt es einem Format das Wireshark (https://www.wireshark.org/) lesen kann. Sofern IP-Header-Kompression über TETRA verwendet wird dann werden diese IP-Header Dekomprimiert gespeichert (max. ein Stream).
- “ThirdPartyComponent_SsiFilter”: Zeigt an wie Nachrichten gefiltert werden können. Es werden zwei Filter bis zu 100 SSIs ermöglicht. Der Filter kann als Black- oder White-List funktionieren.