RSCP-Beispielapplikation

Anleitung

für Entwickler

Änderungshistorie

Datum	Änderung	Version	Bearbeiter
09.11.2015	Erstellung der Dokumentationen	1.0	DiKi
29.03.2016	Überarbeitung	1.1	HaDoele
25.04.2016	Link eingefügt	1.2	NiSchram

Rechtliche Bestimmungen

Weitere Informationen

Die Anleitung ist für den beidseitigen Druck (Duplexdruck) optimiert.

Bei Fragen hilft die E3/DC GmbH gerne weiter.

Weitere Informationen zum Produkt und zur E3/DC GmbH entnehmen Sie bitte der Firmenwebsite.

E3/DC GmbH
Karlstraße 5
D-49074 Osnabrück
Telefon: +49 541 760268-0
Fax: +49 541 760268-19
E-Mail: info@e3dc.com
Website: www.e3dc.com
Kundenportal: https://s10.e3dc.com (Anmeldung erforderlich)

Inhaltsverzeichnis

1 Allgemeines

Hinweis:

Bitte haben Sie dafür Verständnis, dass der Technische Support der E3/DC GmbH keine Unterstützung für diese Beispielapplikation leisten kann.

Diese Dokumentation richtet sich an Softwareentwickler und soll die Kommunikation mit Hilfe des RSCP-Protokolls und die AES-Verschlüsselung an einem Beispiel erläutern. Es wird nicht näher auf das RSCP-Protokoll an sich und dessen Aufbau eingegangen, sondern lediglich auf die Nutzung des Protokolls. Kenntnisse in der C/C++ Programmierung werden vorausgesetzt.

Bei den S10 Hauskraftwerken der E3/DC GmbH handelt es sich um Energiespeichersysteme, bestehend aus PV-Wechselrichter, Lithium-Ionen-Akkus und Batteriewandler. Alle Teilkomponenten sind vollständig in das System integriert.

Für die Kommunikation mit externen Komponenten wird die Ethernet-Schnittstelle verwendet. Dabei wird ein selbst entwickeltes Protokoll genutzt, das Remote-Storage-Control-Protokoll (RSCP), das auf TCP/IP aufsetzt jedoch nicht darauf limitiert ist. Für die externe Steuerung des Hauskraftwerks oder die Beschaffung von Hauskraftwerksdaten, wird von der Software des Hauskraftwerks ein TCP/IP-Server gestartet, auf den alle Teilnehmer im lokalen Ethernet Subnetz Zugriff haben. Damit kann die interne Steuerung des Hauskraftwerks (teilweise) ersetzt werden und den eigenen Bedürfnissen, z. B. im Lade-/Entladezyklus der Batterie, angepasst werden.

Um die Kommunikation vor unerlaubten Zugriffen und somit das gesamte Hauskraftwerk vor unerlaubter Kontrolle zu schützen, wird die gesamte Kommunikation mit dem AES-Verschlüsselungsverfahren chiffriert. Verbindet sich ein Teilnehmer zu der Serversoftware des Hauskraftwerks und sendet für die S10-Software unbekannte Daten, wie es z. B. bei einem falschen/unbekannten AES-Schlüssel der Fall wäre, so wird die TCP/IP-Verbindung direkt geschlossen und die Kommunikation damit abgebrochen.

Zur dieser Dokumentation sollte möglichst parallel die Beispielsoftware *RscpExample* eingesehen werden, da an vielen Stellen der Dokumentation ein Verweis auf diese Software gemacht wird.

Wichtig:

Da wir im Downloadbereich nur diese Anleitung als PDF zur Verfügung stellen, müssen Sie unter folgendem Link weitere Dateien, die zu dieser Anleitung gehören, herunterladen.

http://s10.e3dc.com/dokumentation/RscpExample.zip

2 Verschlüsselungsverfahren und Schlüsselerzeugung

2.1 Schlüsseleingabe und -erzeugung

Die Schlüsseleingabe (Passwort) wird direkt lokal am Display des Geräts im Menüpunkt „Einstellungen > RSCP Schlüssel“ eingegeben und kann bis zu 32 Zeichen werden. Das eingegebene Passwort wird direkt als AES-Schlüssel genutzt. Die fehlenden Zeichen, um den Schlüssel auf 32 Zeichen zu bringen, werden mit dem Wert 0xFF im Schlüsselpuffer aufgefüllt. Eine Eingabe von mehr als 32 Zeichen ist am Display nicht möglich und würde zusätzlich in der Software den Schlüssel auf 32 Zeichen kappen.

Wird kein Passwort am Gerät eingegeben (Auslieferungszustand), so wird von der Software ein eigens erzeugter Initialschlüssel genutzt der für Außenstehende nicht bekannt ist. Damit wird verhindert, dass initial ein unbefugter Zugriff zugelassen wird.

Der folgende Source-Code Ausschnitt aus dem *RscpExample* stellt den Initialisierungsvorgang des AES-Schlüssels vor.

```
// limit password length to AES_KEY_SIZE
int iPasswordLength = strlen(AES_PASSWORD);
if(iPasswordLength > AES_KEY_SIZE)
  iPasswordLength = AES_KEY_SIZE;


// copy up to 32 bytes of AES key password
uint8_t ucAesKey[AES_KEY_SIZE];
memset(ucAesKey, 0xff, AES_KEY_SIZE);
memcpy(ucAesKey, AES_PASSWORD, iPasswordLength);
```

Als Wert für AES_PASSWORD wird hierbei das Passwort als C-String eingesetzt und der AES_KEY_SIZE ist 32 Bytes.

2.2 AES-Verfahren

Das verwendete Verschlüsselungsverfahren für die gesamte Kommunikation mit dem S10-Hauskraftwerk ist das AES-Verschlüsslungsverfahren. Hierbei wird speziell das Cipher-Block-Chain-Verfahren (CBC) verwendet. Als Schlüssellänge wurde eine Länge von 256 Bits (32 Bytes) gewählt. Die Blocklänge für das AES-Verfahren wurde ebenfalls mit einer Länge von 256 Bits gewählt (AES256).
Somit wird die maximal mögliche Sicherheit beim Verschlüsselungsverfahren geboten.

In der Beispielsoftware *RscpExample* ist eine AES C++ Klasse mitgeliefert, die für freie Verwendung in kommerzieller Software freigeben ist (kein GPL), und das AES-Verfahren mit der gewählten Blocklänge und Schlüssellänge unterstützt. Im Folgenden wird dargestellt, wie diese Klasse für die Verwendung initialisiert werden kann:

```
AES aesEncrypter;
AES aesDecrypter;

// set encryptor and decryptor parameters
aesDecrypter.SetParameters(AES_KEY_SIZE * 8, AES_BLOCK_SIZE * 8);
aesEncrypter.SetParameters(AES_KEY_SIZE * 8, AES_BLOCK_SIZE * 8);
```

```
aesDecrypter.StartDecryption(ucAesKey);
aesEncrypter.StartEncryption(ucAesKey);
```

Hierbei werden als AES_KEY_SIZE und auch als AES_BLOCK_SIZE 32 Bytes verwendet. Als Parameter der StartDecryption- und StartEncryption-Methoden werden die zuvor erzeugte AES-Schlüssel verwendet.

Weiterhin wird für die AES-Verschlüsselung ein IV (Initial Value) Wert in der Größe der AES-Blöcke benötigt. Der IV wird beim Kommunikationsbeginn mit dem S10-Hauskraftwerk mit dem Wert 0xFF auf den gesamten 32 Bytes initialisiert und bei der Kommunikation durchgehend mit dem neu errechneten IV-Wert nach einer Entschlüsselung oder Verschlüsselung weiterverwendet. Der folgende Codeausschnitt stellt die Initialisierung des IV zum Kommunikationsbeginn dar.

```
uint8_t ucEncryptionIV[AES_BLOCK_SIZE];
uint8_t ucDecryptionIV[AES_BLOCK_SIZE];

memset(ucEncryptionIV, 0xff, AES_BLOCK_SIZE);
memset(ucDecryptionIV, 0xff, AES_BLOCK_SIZE);
```

Im weiteren Verlauf der Kommunikation muss der IV aus den verschlüsselten oder entschlüsselten Daten entnommen und für die weitere Verschlüsselung/ Entschlüsselung wieder verwendet werden.

```
// set continues encryption IV
aesEncrypter.SetIV(ucEncryptionIV, AES_BLOCK_SIZE);
// start encryption from encryptionBuffer to encryptionBuffer, blocks =
encryptionBuffer.size() / AES_BLOCK_SIZE
aesEncrypter.Encrypt(&encryptionBuffer[0], &encryptionBuffer[0],
encryptionBuffer.size() / AES_BLOCK_SIZE);
// save new IV for next encryption block
memcpy(ucEncryptionIV, &encryptionBuffer[0] + encryptionBuffer.size() -
AES_BLOCK_SIZE, AES_BLOCK_SIZE);
```

In diesem Codeausschnitt wird dargestellt, wie die Verschlüsselung aus dem Puffer *encryptionBuffer* in denselben Puffer *encryptionBuffer* durchgeführt wird. Dabei ist hier zu sehen, wie vor jedem Verschlüsselungsvorgang der IV neu im *aesEncryptor* gesetzt wird und nach dem Verschlüsselungsvorgang aus dem letzten verschlüsselten Block für den nächsten Verschlüsselungsvorgang kopiert wird.

Ein ähnliches Verfahren wird ebenfalls für die Entschlüsselung verwendet. Der folgende Codeausschnitt stellt diesen Vorgang dar.

```
// initialize encryption sequence IV value with value of previous block
aesDecrypter.SetIV(ucDecryptionIV, AES_BLOCK_SIZE);
// decrypt data from vecDynamicBuffer to temporary decryptionBuffer
aesDecrypter.Decrypt(&vecDynamicBuffer[0], &decryptionBuffer[0], iLength /
AES_BLOCK_SIZE);
…
// store the IV value from encrypted buffer for next block decryption
memcpy(ucDecryptionIV, &vecDynamicBuffer[0] + iProcessedBytes -
AES_BLOCK_SIZE, AES_BLOCK_SIZE);
```

Bei diesem Vorgang ist der IV von dem letzten verschlüsselten Block zu kopieren. Dabei muss bekannt sein, welcher Block als „letzter“ Block gilt. Dies wird in dem *RscpExample* durch ein temporäres Entschlüsseln und Prüfen der Daten realisiert. Solange das Frame nicht vollständig ist, müssen die verschlüsselten Daten im Puffer verbleiben, um im nächsten Schritt den Vorgang wiederholen zu können.

Es wäre auch denkbar, durchgehend nach jedem vollständigen (oder vielfachen) Block die Daten direkt zu entschlüsseln und den IV nach jedem dieser Vorgänge zu

sichern. Bei diesem Verfahren müssten dann die entschlüsselten Daten vorläufig in einem separaten statischen Puffer aufgestockt gesichert werden, bis ein vollständiger Frame erhalten wird. Die verschlüsselten Daten, die bereits entschlüsselt wurden, könnten jedoch dann direkt verworfen werden. Dies würde ggf. einen sichtbaren Performancevorteil gegenüber der ersten Methode bei Prozessoren mit geringer Leistung bedeuten bei eventuell höherer RAM-Speicher-Nutzung.

3 TCP/IP Kommunikation

Die TCP/IP Verbindung wird in dem *RscpExample* mit Hilfe von Unix-(POSIX)-Sockets realisiert. Dafür wurden vier sehr simple und auf das nötigste reduzierte Funktionen implementiert. Es ist ratsam bei einer Portierung des Codes, diese Methoden durch die eigenen, im Projekt bereits vorhandenen Methoden zu ersetzen.

Die Verwendung der Sockets ist jedoch nicht Gegenstand dieser Dokumentation und würde den Rahmen dieser sprengen.

Ein Beispiel für den Verbindungsaufbau mit einer Windows OS Maschine ist nicht in dem *RscpExample* enthalten. Es sollte jedoch einfach sein, dieses dahingegen zu realisieren, da die Funktionen die für die TCP Verbindung zuständig sind, von dem Rest des Beispiels entkoppelt in der Datei „SocketConnection.cpp“ untergebracht sind.

Folgende Funktionen müssen dafür angepasst werden:

```
int SocketConnect(const char *cpIpAddress, int iPort);
void SocketClose(int iSocket);
int SocketSendData(int iSocket, const unsigned char * ucBuffer, int iLength);
int SocketRecvData(int iSocket, unsigned char * ucBuffer, int iLength);
```

Die *SocketConnect*-Methode verbindet sich zu einem Server mit der als Parameter übergebenen IP-Adresse und den entsprechenden Port. Die *SocketClose*-Methode schließt diese Verbindung wieder.

Die beiden Methoden *SocketSendData* und *SocketRecvData* sind für die Übertragung der Daten zuständig.

4 Programmaufbau und -ablauf

In der ersten Phase des Programms wird die Verbindung über ein TCP/IP-Socket zum Server erstellt. Dabei wird eine IP im lokalen Subnetz angegeben und als Port wird die 5033 (S10-Software) oder 5034 (S10-Farm-Software) verwendet. Daraufhin werden die AES-Klassen für das Verschlüsseln und Entschlüsseln initialisiert. Diese Schritte wurden in den Kapiteln zuvor bereits ausführlicher erklärt und werden daher in diesem Punkt nicht weiter erläutert.

Nach der Initialisierungsphase wird die Hauptschleife *mainLoop* aufgerufen. Diese Hauptschleife wird solange durchlaufen, bis ein Fehler erkannt wird oder die Kommunikation aus irgendeinem Grund getrennt wird. Wird die Hauptschleife verlassen, so wird der noch offene Socket geschlossen und die Applikation startet von neuem mit dem Verbindungsaufbau. D. h., es handelt sich hierbei um eine Endlosapplikation, die nie beendet wird. Unter Linux kann diese Applikation nur durch das Eingreifen eines Benutzers z. B. mit der Tastenkombination <STRG + C> beendet werden.

4.1 Hauptschleife / Sendeschleife *mainLoop*

Die Hauptschleife *mainLoop* stellt die reguläre Kommunikation zwischen der *RscpExample*-Applikation und einem S10 Hauskraftwerk dar.

Im ersten Teil dieser Schleife wird mit der Funktion *createRequestExample* ein valides RSCP Frame mit den Anfrage-TAGs an das S10 Hauskraftwerk erzeugt. Die Behandlung dieser Methode wird im Kapitel „Erzeugen des RSCP-Frames – *createRequestExample*“ erläutert.

Ist das RSCP-Frame erzeugt und von der *createRequestExample*-Methode in den Container *SRscpFrameBuffer* eingebracht, so werden diese Daten verschlüsselt und an die S10-Software versendet. Dazu wird die Größe des Puffers zunächst auf ein vielfaches einer AES-Blockgröße (32 Byte) aufgerundet und die aufgerundeten Datenbytes mit 0 aufgefüllt. Die Daten müssen dazu in einen separaten, temporären Block kopiert werden, um nicht über das Ende des vorhandenen Puffers von *SRscpFrameBuffer* zu schreiben. Dies wird mit dem nachfolgenden Codeausschnitt durchgeführt.

```
// resize temporary encryption buffer to a multiple of AES_BLOCK_SIZE
std::vector<uint8_t> encryptionBuffer;
encryptionBuffer.resize(ROUNDUP(frameBuffer.dataLength,
AES_BLOCK_SIZE));
// zero padding for data above the desired length
memset(&encryptionBuffer[0] + frameBuffer.dataLength, 0, encryptionBuffer.size()
- frameBuffer.dataLength);
// copy desired data length
memcpy(&encryptionBuffer[0], frameBuffer.data, frameBuffer.dataLength);
```

Im nächsten Schritt werden diese Daten mit AES verschlüsselt. Dazu wird der aktuell gültige IV genutzt. Dies wird mit Hilfe dieses Codeausschnitts, welcher zuvor im Kapitel „AES-Verfahren“ erläutert wurde, realisiert.

```
// set continues encryption IV
aesEncrypter.SetIV(ucEncryptionIV, AES_BLOCK_SIZE);
// start encryption from encryptionBuffer to encryptionBuffer, blocks =
encryptionBuffer.size() / AES_BLOCK_SIZE
aesEncrypter.Encrypt(&encryptionBuffer[0], &encryptionBuffer[0],
encryptionBuffer.size() / AES_BLOCK_SIZE);
// save new IV for next encryption block
memcpy(ucEncryptionIV, &encryptionBuffer[0] + encryptionBuffer.size() -
AES_BLOCK_SIZE, AES_BLOCK_SIZE);
```

Nach diesem Vorgang werden die Daten nur noch über die Socket-Methode versendet und die Empfangsschleife *receiveLoop* betreten. Diese Schleife wird im nächsten Kapitel „Empfangsschleife *receiveLoop*“ näher erklärt. Ist die Empfangsschleife wieder verlassen, werden nur noch die gesendeten wieder frei gegeben, der Thread für eine Sekunden in den Schlaf geschickt und danach der ganze Vorgang wieder neu gestartet. Diese Schritte sind im nächsten Codeausschnitt dargestellt.

```
// send data on socket
int iResult = SocketSendData(iSocket, &encryptionBuffer[0],
encryptionBuffer.size());
if(iResult < 0) {
    printf("Socket send error %i. errno %i\n", iResult, errno);
    bStopExecution = true;
}
else {
```

```
    // go into receive loop and wait for response
    receiveLoop(bStopExecution);
    }
}
// free frame buffer memory
protocol.destroyFrameData(&frameBuffer);

// main loop sleep / cycle time before next request
sleep(1);
```

Die Zykluszeit von einer Sekunde ist frei gewählt und kann auch deutlich kleiner ausfallen, falls benötigt.

4.2 Empfangsschleife *receiveLoop*

Die Empfangsschleife wird nach jedem Sendevorgang eines RSCP-Frames betreten, da für jeden Sendevorgang eine Antwort im RSCP-Protokoll erwartet wird.

Die *receiveLoop*-Schleife wird solange durchlaufen, bis ein Receive-Timeout erkannt wird, ein Fehler auf der Socket-Schnittstelle auftritt oder mindestens 1 valides RSCP Frame erhalten wird.

Im ersten Schritt der Schleife wird ein im Speicher statischer aber in der Größe dynamischer Puffer in der Größe bei Bedarf vergrößert. Ist die Größe des Puffers abzüglich der im Puffer bereits vorhandenen Größe der Daten aus vorherigem Schleifendurchgang kleiner als 4096 Bytes, so wird der Speicher um 4096 Bytes vergrößert. Dies wird mit dem folgenden Codeausschnitt realisiert:

```
// check and expand buffer
if((vecDynamicBuffer.size() - iReceivedBytes) < 4096) {
  // check maximum size
  if(vecDynamicBuffer.size() > RSCP_MAX_FRAME_LENGTH) {
    // something went wrong and the size is more than possible by the RSCP protocol
    printf("Maximum buffer size exceeded %i\n", vecDynamicBuffer.size());
    bStopExecution = true;
    break;
  }
  // increase buffer size by 4096 bytes each time the remaining size is smaller than
    4096
  vecDynamicBuffer.resize(vecDynamicBuffer.size() + 4096);
}
```

Die Größe des Puffers ist dabei auf maximal 64 KB limitiert, was die maximale Übertragungsgröße des RSCP-Protokolls pro RSCP Frame darstellt.

Im nächsten Schritt werden die Daten von der TCP Schnittstelle gelesen. Diese werden hinter die Position von den Bereits vorhandenen Daten im Puffer, die noch nicht verarbeitet wurden, gelesen. Dies ist z. B. der Fall, wenn das RSCP Frame noch nicht vollständig erhalten wurde, sondern lediglich ein Teil davon. So werden die Daten weiter gelesen, um das Frame zu vervollständigen. Dies wird mit dem folgenden Codeausschnitt realisiert.

```
// receive data
int iResult = SocketRecvData(iSocket, &vecDynamicBuffer[0] + iReceivedBytes,
vecDynamicBuffer.size() - iReceivedBytes);
if(iResult < 0)
{
  // check errno for the error code to detect if this is a timeout or a socket error
  if ((errno == EAGAIN) || (errno == EWOULDBLOCK)) {
```

```
        // receive timed out -> continue with re-sending the initial block
        printf("Response receive timeout (retry)\n");
        break;
      }
    // socket error -> check errno for failure code if needed
    printf("Socket receive error. errno %i\n", errno);
    bStopExecution = true;
    break;
    }
    else if(iResult == 0)
    {
      // connection was closed regularly by peer
      // if this happens on startup each time the possible reason is
      // wrong AES password or wrong network subnet (adapt hosts.allow file required)
      printf("Connection closed by peer\n");
      bStopExecution = true;
      break;
    }
    // increment amount of received bytes
    iReceivedBytes += iResult;
```

Bei einem Fehler auf der Schnittstelle, einem Timeout oder einer regulären Schließung der Verbindung durch die Gegenseite, wird die Empfangsschleife umgehend unterbrochen und die Methode verlassen. Hierbei ist zu erwähnen, dass der Timeout in der Socket-Schnittstelle dieses Beispiels mit 3 Sekunden gewählt wurde. Tritt keiner dieser Fehler auf, so wird der Zähler für die bereits erhaltenen Daten inkrementiert und die Frame-Parse-Subschleife wird betreten.

Innerhalb der Frame-Parse-Subschleife der Empfangsschleife wird geprüft, ob die erhaltenen Daten bereits ein vollständiges RSCP Frame enthalten und in diesem Fall dann das RSCP-Frame verarbeitet. Diese Schleife läuft so oft durch, solange noch valide RSCP-Frames in dem Datenpuffer enthalten sind. Dies kann z.B. dann auftreten, wenn ein Timeout im vorherigen Sendevorgang aufgetreten ist und das RSCP-Frame zu früh erneut angefragt wird, so dass im nächsten Empfangsschritt bereits zwei Frames im Empfangspuffer befinden. Daher wird an dieser Stelle alles in einer Schleife verarbeitet, was möglich ist.

Dazu wird zunächst die Anzahl der empfangenen Datenbytes auf ein vielfaches der AES-Blockgröße (32 Bytes) **abgerundet**. Ist diese neue Anzahl der Datenbytes gleich 0, so wurden nicht genügend Daten erhalten und die Frame-Parse-Subschleife wird beendet und es wird erneut versucht auf der TCP-Schnittstelle zu lesen. Sind mindestens 32 Bytes an Daten vorhanden, so werden diese mit dem nachfolgenden Codeausschnitt entschlüsselt, und es wird versucht, die entschlüsselten Daten zu verarbeiten.

```
// round down to a multiple of AES_BLOCK_SIZE
int iLength = ROUNDDOWN(iReceivedBytes, AES_BLOCK_SIZE);
// if not even 32 bytes were received then the frame is still incomplete
if(iLength == 0) {
break;
}
// resize temporary decryption buffer
std::vector<uint8_t> decryptionBuffer;
decryptionBuffer.resize(iLength);
// initialize encryption sequence IV value with value of previous block
aesDecrypter.SetIV(ucDecryptionIV, AES_BLOCK_SIZE);
// decrypt data from vecDynamicBuffer to temporary decryptionBuffer
aesDecrypter.Decrypt(&vecDynamicBuffer[0], &decryptionBuffer[0], iLength /
AES_BLOCK_SIZE);
```

```
// data was received, check if we received all data
int iProcessedBytes = ProcessReceiveBuffer(&decryptionBuffer[0], iLength);
```

Die Funktion *processReceiveBuffer* wird dazu mit den entschlüsselten Daten aufgerufen. Diese Funktion wird im Kapitel „Verarbeiten des RSCP Frames – *processReceiveBuffer*“ näher erläutert. Der Rückgabewert dieser Funktion gibt entweder einen Fehler bei einem Wert kleiner Null wieder oder die Anzahl der verarbeiteten Datenbytes zurück. Gibt dieser Wert einen Fehler, so werden alle vorhergehenden Schleifen verlassen und die Verbindung geschlossen, da es sich in diesem Fall um keine RSCP-Daten handelt. Sind die Daten valide und wurden diese verarbeitet, so werden diese Daten aus dem dynamischen Puffer entfernt und die Anzahl der erhaltenen Datenbytes um diesen Wert aufgerundet auf ein vielfaches der AES-Blockgröße reduziert. Die Daten im dynamischen Puffer hinter den verarbeiteten Datenbytes werden dann nach vorne verschoben. Zusätzlich wird ein Zähler der verarbeiteten RSCP-Frames hoch gezählt und die Schleife neu gestartet um weitere mögliche RSCP-Frames im dynamischen Puffer zu verarbeiten. Dieser Vorgang wird wiederholt bis der *iProcessedBytes* Rückgabewert der *processReceiveBuffer* eine Null zurück gibt. Dies bedeutet, dass nicht genügend Datenbytes mehr im dynamischen Puffer enthalten sind, um ein vollständiges RSCP-Frame zu enthalten. Der Zähler der RSCP-Frames wird im späteren Verlauf für die Erkennung, ob mindestens ein RSCP-Frame erhalten wurde, in der Empfangsschleife genutzt, um diese regulär zu verlassen.

Diese Vorgänge werden in dem nachfolgendem Codeabschnitt dargestellt.

```
if(iProcessedBytes < 0) {
  // an error occured;
  printf("Error parsing RSCP frame: %i\n", iProcessedBytes);
  // stop execution as the data received is not RSCP data
  bStopExecution = true;
  break;

}
else if(iProcessedBytes > 0) {
  // round up the processed bytes as iProcessedBytes does not include the zero
  padding bytes
  iProcessedBytes = ROUNDUP(iProcessedBytes, AES_BLOCK_SIZE);
  // store the IV value from encrypted buffer for next block decryption
  memcpy(ucDecryptionIV, &vecDynamicBuffer[0] + iProcessedBytes -
  AES_BLOCK_SIZE, AES_BLOCK_SIZE);
  // move the encrypted data behind the current frame data (if any received) to the
 front
  memcpy(&vecDynamicBuffer[0], &vecDynamicBuffer[0] + iProcessedBytes,
  vecDynamicBuffer.size() - iProcessedBytes);
  // decrement the total received bytes by the amount of processed bytes
  iReceivedBytes -= iProcessedBytes;
  // increment a counter that a valid frame was received and
  // continue parsing process in case a 2nd valid frame is in the buffer as well
  iReceivedRscpFrames++;
}
else {
  // iProcessedBytes is 0
  // not enough data of the next frame received, go back to receive mode if
  iReceivedRscpFrames == 0
  // or transmit mode if iReceivedRscpFrames > 0
  break;
}
```

Nach diesen Vorgängen beginnt die Empfangsschleife wieder von vorn, falls keine validen RSCP-Frames gefunden wurden, oder sie wird beendet, wenn mindestens ein RSCP-Frame verarbeitet wurde.

4.3 Erzeugen des RSCP Frames – *createRequestExample*

Das Erzeugen des RSCP Frames mit der Methode *createRequestExample* wird je nach dem Zustand der Variable *iAuthenticated* unterschiedlich behandelt. Diese Variable wird vor dem Aufruf der Hauptschleife auf 0 initialisiert. Ist dieser Wert 0 innerhalb der *createRequestExample*-Methode so wird in das RSCP-Frame zunächst nur der *TAG_RSCP_REQ_AUTHENTICATION* eingebracht, mit den dazugehörigen Sub-TAGs für User und Passwort. Der folgende Codeausschnitt stellt dies einmal dar.

```
//-------------------------------------------------------------------------------------------------
----
// Create a request frame
//-------------------------------------------------------------------------------------------------
----
if(iAuthenticated == 0)
{
  printf("\nRequest authentication\n");
  // authentication request
  SRscpValue authenContainer;
  protocol.createContainerValue(&authenContainer,
  TAG_RSCP_REQ_AUTHENTICATION);
  protocol.appendValue(&authenContainer,
  TAG_RSCP_AUTHENTICATION_USER, E3DC_USER);
  protocol.appendValue(&authenContainer,
  TAG_RSCP_AUTHENTICATION_PASSWORD, E3DC_PASSWORD);
  // append sub-container to root container
  protocol.appendValue(&rootValue, authenContainer);
  // free memory of sub-container as it is now copied to rootValue
  protocol.destroyValueData(authenContainer);
}
```

Ist die Authentifizierung (Login) abgeschlossen und erfolgte eine Antwort auf diesen TAG, mit einem Authentifizierungslevel größer Null, so wird die Variable *iAuthenticated* auf 1 gesetzt. Das Verarbeiten der Empfangsbotschaften wird im Kapitel 8.4 „Verarbeiten des RSCP Frames – *processReceiveBuffer*“ näher erläutert.

Ist der Merker für eine erfolgreiche Authentifizierung auf 1 beim Betreten der *createRequestExample*-Methode, so werden die regulären Anfrage-TAGs in das RSCP-Frame gelegt. Dies wird mit dem nachfolgenden Codeausschnitt dargestellt.

```
else
{
  printf("\nRequest cyclic example data\n");
  // request power data information
  protocol.appendValue(&rootValue, TAG_EMS_REQ_POWER_PV);
  protocol.appendValue(&rootValue, TAG_EMS_REQ_POWER_BAT);
  protocol.appendValue(&rootValue, TAG_EMS_REQ_POWER_HOME);
  protocol.appendValue(&rootValue, TAG_EMS_REQ_POWER_GRID);
  protocol.appendValue(&rootValue, TAG_EMS_REQ_POWER_ADD);

  // request battery information
  SRscpValue batteryContainer;
  protocol.createContainerValue(&batteryContainer, TAG_BAT_REQ_DATA);
```

```
    protocol.appendValue(&batteryContainer, TAG_BAT_INDEX, (uint8_t)0);
    protocol.appendValue(&batteryContainer, TAG_BAT_REQ_RSOC);
    protocol.appendValue(&batteryContainer,
    TAG_BAT_REQ_MODULE_VOLTAGE);
    protocol.appendValue(&batteryContainer, TAG_BAT_REQ_CURRENT);
    protocol.appendValue(&batteryContainer, TAG_BAT_REQ_STATUS_CODE);
    protocol.appendValue(&batteryContainer, TAG_BAT_REQ_ERROR_CODE);
    // append sub-container to root container
    protocol.appendValue(&rootValue, batteryContainer);
    // free memory of sub-container as it is now copied to rootValue
    protocol.destroyValueData(batteryContainer);
  }
```

Hierbei wurden einige Beispiel TAGs verwendet, um den Füllvorgang des RSCP Frames exemplarisch darzustellen.

Am Ende der Methode werden die temporären Container die innerhalb dieser Methode angelegt wurden in ein gesamt Paket, das RSCP-Frame, zusammengeführt. Dies stellt der folgende Codeausschnitt dar.

```
// create buffer frame to send data to the S10
protocol.createFrameAsBuffer(frameBuffer, rootValue.data, rootValue.length, true);
// the root value object should be destroyed after the data is copied into the
frameBuffer and is not needed anymore
protocol.destroyValueData(rootValue);
```

4.4 Verarbeiten des RSCP Frames – *processReceiveBuffer*

Die *processReceiveBuffer*-Methode prüft den erhaltenen Puffer, der zuvor entschlüsselt wurde. Im ersten Schritt werden der Inhalt und die Länge des erhaltenen Puffers überprüft. Dazu wird die Funktion *parseFrame* des RSCP-Protokolls aufgerufen. Diese gibt bei einem Fehler einen Wert kleiner 0 und beim erfolgreichem Parsen einen Wert größer 0 zurück, entsprechend der verarbeiteten Bytes des Frames. Hiermit kann ebenfalls gleichzeitig geprüft werden, ob die erhaltene Daten im Puffer ausreichend in der Länge sind. Ist der Rückgabewert von *parseFrame* kleiner 0 und entspricht *RSCP::ERR_INVALID_FRAME_LENGTH*, so ist der Puffer valide, jedoch ist die Länge der übergebenen Daten zu kurz. In diesem Fall muss weiter auf der Schnittstelle gelesen werden. In allen anderen Fällen sind die Daten im Puffer keine validen RSCP-Daten. Der Fehlercode kann im RSCP-Protokoll eingesehen werden.

Der folgende Codeausschnitt stellt den Vorgang dar.

```
RscpProtocol protocol;
SRscpFrame frame;

int iResult = protocol.parseFrame(ucBuffer, iLength, &frame);
if(iResult < 0) {
  // check if frame length error occured
  // in that case the full frame length was not received yet
  // and the receive function must get more data
  if(iResult == RSCP::ERR_INVALID_FRAME_LENGTH) {
    return 0;
  }
  // otherwise a not recoverable error occured and the connection can be closed
  else {
    return iResult;
```

```
    }
  }
  int iProcessedBytes = iResult;
```

Im zweiten Schritt der *processReceiveBuffer*-Methode werden die erfolgreich verarbeiteten Daten des RSCP-Frames weiter verarbeitet. Der folgende Codeausschnitt stellt dies dar.

```
// process each SRscpValue struct seperately
for(unsigned int i; i < frame.data.size(); i++) {
handleResponseValue(&protocol, &frame.data[i]);
}

// destroy frame data and free memory
protocol.destroyFrameData(frame);

// returned processed amount of bytes
return iProcessedBytes;
```

Hierbei wird der *std::vector* mit den *SRscpValue*-Strukturen innerhalb der *SRscpFrame*-Struktur mit den darin enthaltenen Daten durch iteriert und die Unterfunktion *handleResponseValue* für jeden *SRscpValue* aufgerufen.

Die Unterfunktion *handleResponseValue* verarbeitet dann die Daten des *SRscpValue* entsprechend des Rückgabe-TAGs dazu. Es handelt sich dabei um einen simplen Switch-Block, der entsprechend des TAGs eine Aufgabe erledigt. Dies wird im folgenden Codeausschnitt dargestellt.

```
// check if any of the response has the error flag set and react accordingly
if(response->dataType == RSCP::eTypeError) {
  // handle error for example access denied errors
  uint32_t uiErrorCode = protocol->getValueAsUInt32(response);
  printf("Tag 0x%08X received error code %u.\n", response->tag, uiErrorCode);
  return -1;
}

// check the SRscpValue TAG to detect which response it is
switch(response->tag){
case TAG_RSCP_AUTHENTICATION: {
  // It is possible to check the response->dataType value to detect correct data type
  // and call the correct function. If data type is known,
  // the correct function can be called directly like in this case.
uint8_t ucAccessLevel = protocol->getValueAsUChar8(response);
if(ucAccessLevel > 0) {
  iAuthenticated = 1;
}
printf("RSCP authentitication level %i\n", ucAccessLevel);
break;
}
…
```

Hierbei ist zu erkennen, dass zunächst jeder TAG darauf überprüft wird, ob dieser einen Fehlercode zurückgegeben hat, weil z. B. kein Zugriff auf diesen TAG mit dem aktuellen Authentifizierungslevel gegeben ist, und entsprechend wird dann der Fehlercode ausgegeben. Ist die *dataType*-Variable der *SRscpValue*-Struktur nicht als *RSCP::eTypeError* markiert, so kann in diesem Feld der Datentyp des Rückgabewertes entnommen werden (siehe RscpTypes.h).

In dem oben zu sehenden Codeausschnitt wird der Rückgabewert des *TAG_RSCP_AUTHENTICATION* verarbeitet. So wird zum Beispiel die Variable

iAuthenticated auf den Wert 1 gesetzt, wenn der erhaltene Authentifizierungslevel größer 0 ist.

Nachdem alle *SRscpValue*-Werte behandelt wurden, wird nur noch der Speicher für die einzelnen Strukturen freigegeben und die *processReceiveBuffer*-Methode verlassen.