Dokumentation TheOpenTransporter

Die OpenSource-JSON-Schnittstelle für FHEM

Dieses Dokument enthält die Dokumentation für TheOpenTransporter. Es wird detailliert auf die einzelnen Funktionen des gesamten Modules eingegangen.

Hochschule Harz: Marcus Viererbe Oliver Jathe
18.07.2012

1 Inhaltsverzeichnis

1 Grundlegendes

1.1 Motivation

TheOpenTransporter (kurz: TOT) ist als Teil eines Teamprojektes, des Studiengang Informatik, der Hochschule Harz entstanden. TOT wurde als Teilprojekt in Auftrag gegeben, damit ein weiterer Teil des Teams die Möglichkeit bekommt via HTTP auf FHEM zugreifen zu können. Die Aufgabenstellung lag darin, nicht nur das lesen, sondern auch das schreiben und modifizieren aller Daten zu ermöglichen. Es wurde festgelegt, dass der Datentransfer per JSON durchgeführt werden soll.

Aus verschiedenen Gründen konnte das Gesamtprojekt leider nicht fertiggestellt werden. TheOpenTransporter wurde aber voll funktionsfähig fertiggestellt und konnte daher der FHEM-Community zur Verfügung gestellt werden.

1.2 Autoren

Die Autoren sind beide Studenten des Studiengangs Informatik der Hochschule Harz im 7. Semester.

Marcus Viererbe

E-Mail: MarcusViererbe@theOpenTransporter.de

Wohnort: Magdeburg

Oliver Jathe

E-Mail: ojathe@TheOpenTransporter.de

Wohnort: Hannover

1.3 Modulstruktur

In dem FHEM-Modul TheOpenTransporter wird in der Define-Funktion ein passiver TCP-Socket aufgebaut, der ab diesem Zeitpunkt auf einen Verbindungsaufbau wartet. Sollte eine Verbindung angefordert werden, wird die Funktion „Read“ durch FHEM in TOT aufgerufen. In der Methode wird dann ein aktiver Socket angelegt. (Dies kann bei Bedarf auch ein SSL-Socket sein, sofern dies im Hauptmodul konfiguriert wurde.) Nach erfolgreichem anlegen des Client-Socket (aktiver Socket) kann dieser nun Anfragen entgegen nehmen. Es wird nicht nur ein Socket definiert, sondern ein Kind-Modul für diesen Socket mit dessen Port erzeugt. Sollten Daten eingegangen sein, wird der dafür genutzte Service festgestellt (GET, POST usw.). Bei einem bekannten Service (sind in einer Hashtable definiert) wird die dem Service zugehörige Verarbeitungsroutine aufgerufen. Bei einem unbekannten Service wird

ein Http-Fehler-Code zurückgegeben (501 – Not Implemented). Die Verbindung bleibt nun solange bestehen, bis der Client diese beendet oder der Timeout abgelaufen ist.

Der eigentliche Aufruf von Funktionen von TOT geschieht mittels Standard RFC-Konformität. Dies bedeutet, dass Funktions-Namen an die URL angehängt werden.

HTTP-Aufruf:

```
Http://<Server-IP>:<Port>/<Funktionsname>?<Parametername1>=<Parameter1>&<Pname2>=<P2>
```

CodeSnippet 34: Beispielaufruf einer Funktion

Je nach Art des Service werden die entsprechenden Routinen aufgerufen, welche die Funktion bestimmen und eine weitere Routine aufrufen, welche prüft, ob die angeforderte Funktion vorhanden ist. Sollte diese vorhanden sein, wird die entsprechende Funktion aufgerufen, andernfalls wird der HTTP-Fehlercode „404 – Not found“ zurückgegeben.

Ein weiteres Feature ist, die explizite Unterscheidung der einzelnen Content-Types, die von den Clients gesendet werden und extra Abwicklungsroutinen besitzen. Sollte eine der beiden Seiten einen Content-Typen nicht unterstützen, wird dieser auf „text/html“ eingestellt.

2 Implementierung der Grund-Modul-Funktionen

Folgend werden die Funktionen beschrieben, welche für den Grundaufbau des Modules notwendig sind. Gemeint sind damit Methoden, welche benötigt werden, um ein vollwertiges und funktionsfähiges FHEM-Modul darzustellen. Aber auch jene, welche die Funktionalität des HTTP-Servers stellen.

2.1 Initialize($)

Diese Funktion wird beim Laden des Moduls von FHEM aufgerufen. Sie stellt die Verbindung zu FHEM her. Es wird definiert, wie die von FHEM aufzurufenden Methoden im Modul benannt sind. Außerdem wird die Attribut-Liste mit den anzubietenden Attributen initialisiert.

Perl-Code:

```
$selectlist{$name} = $hash;         my ($hash) = @_;
                $hash->{ReadFn}  = "TheOpenTransporter_Read";
                $hash->{DefFn}   = "TheOpenTransporter_Define";
                $hash->{UndefFn} = "TheOpenTransporter_Undef";
                $hash->{AttrFn}  = "TheOpenTransporter_Attr";
                $hash->{NotifyFn}= "TheOpenTransporter_NotifyFn";
                $hash->{AttrList}= "loglevel:0,1,2,3,4,5,6 HTTPS SSL_KEY SSL_CERT
RequestHeaderLineLimit";
```

CodeSnippet 35: Initialisierung der Schnittstelle

2.2 Define($$)

Die Methode Define() wird bei einer Definition des Moduls in „fhem.pl“ aufgerufen. Zuallererst wird die Syntax der angegebenen Parameter überprüft. Sollten nicht genügend (weniger als 3) , oder zu viele (mehr als 10) Parameter vorhanden sein, wird eine Fehlermeldung ausgegeben und damit das Modul nicht definiert.

FHEM-Code:

```
define <name> TheOpenTransporter <Port> [Optional: IPV6] [Optional: HTTPS] [Optional:</path/server-
key.pem> </path/server-cert.pem>]
```

CodeSnippet 36: Define von TheOpenTransporter

Sollten die Parameter ordnungsgemäß angegeben worden sein, wird ein passiver Socket erzeugt, der auf dem angegebenen Port auf eingehende Verbindungen wartet. Dies kann neben einem normalen IPv4-Socket, je nach Parameter, auch ein IPv6- und/ oder ein SSL-Socket sein. Es muss das File-handle des passiven Sockets dem Wert „$hash->{FD}“ in der lokalen Hashtable zugewiesen werden, dies ermöglicht lese/schreib Vorgänge zu erkennen. Die „fhem.pl“ erkennt diese und ruft die Methode „ReadFn“ auf. Um überhaupt in die Liste der aufzurufenden Module zu

gelangen muss sich das Modul in die Liste „selectlist“ eintragen. Dies geschieht über den Namen des Moduls, zugewiesen wird die gesamte Hashtable.

Achtung:

Wichtig ist das zurückgeben eines „undef“ Rückgabewertes, da ansonsten FHEM dies als nicht ordnungsgemäß funktionierendes Modul erkennt und demzufolge dies nicht definiert wird.

2.3 Read($)

Read wird in der Main-Schleife von dem Prozess „fhem.pl“ ausgeführt. Dazu muss sich das Modul in einer Hashtable namens „selectlist" befinden. Bedingung ist, dass die lokale Hashtable des Moduls in der Define-Phase, in die „selectlist“ eingetragen wird.

Perl-Code:

```
$selectlist{$hash->{NAME}} = $hash;
```

CodeSnippet 37: Eintragen in die "selectedlist"

Im weiteren muss der eigenen Hashtable ein File-Descriptor (Handle) des Objektes mit den I/O Operationen (passiver Socket) übergeben werden.

Perl-Code:

```
$hash->{FD} = $hash->{SERVERSOCKET}->fileno();
```

CodeSnippet 38: Handle übergeben

Hierdurch kann bei Schreib/Lesevorgängen (Verbindungsanforderungen) die Methode „TOT_Read“ von FHEM aufgerufen werden.

Achtung:

Es ist wichtig, dass diese Methode in Initialize($) bekannt gemacht wird! (siehe 4.2.6.1), da FHEM ansonsten nicht auf diese Methode zugreifen kann.

Nun zu der eigentlichen Funktionsweise. Bei dem Aufruf von „TOT_Read“ muss im ersten Abschnitt eine Fallunterscheidung durchgeführt werden. Im ersten Fall registriert FHEM eine Verbindungsanforderung, hierbei muss der Server-Socket die neue Verbindung etablieren, falls gewünscht ist dies eine verschlüsselte SSL-Verbindung. Hierbei werden notwendige Eigenschaften für die Verbindung in eine neue Hashtable geschrieben und FHEM bekannt gemacht. Dabei ist zu beachten, dass für die Zeit der bestehenden Verbindung ein Helper-Modul in FHEM sichtbar ist. Das nun folgende Beispiel zeigt, wie ein SSL-Zertifikat erstellt werden kann.

Linux-Code:

```
installed with cpan -i IO::Socket::SSL or apt-get install libio-socket-ssl-perl
mkdir certs
cd certs
openssl req -new -x509 -nodes -out server-cert.pem -days 3650 -keyout server-key.pem
# nicht vergessen zertifikate in /usr/bin/certs zu packen
```

CodeSnippet 39: SSL-Zertifikat Erstellung

Im zweiten Fall ist das Modul kein Serversocket, sondern ein im ersten Fall angelegter Clone, wobei nun Daten zum verarbeiteten anliegen.
Nun wird die erste Request-Zeile gelesen, um die Art des Service und die Version des Service zu bestimmen. Dank einer Hashtable können Service-Anforderungen zu entsprechenden Service-Routinen zugeordnet werden. Bei einer Anfrage wird auf Existenz des entsprechenden Service überprüft und die darin enthaltene Routine aufgerufen.

2.4 Undef($$)

Die Methode Undef() wird von FHEM aufgerufen, um alle Ressourcen des Moduls zu bereinigen. Konkret bedeutet dies, wenn das Modul entfernt werden soll. Hierbei wird unterschieden, in welchem Modul wir uns befinden (Haupt- oder Clientmodul). In beiden Fällen werden die entsprechenden Sockets geschlossen und die eigene Referenz aus der FHEM „selectedlist“ entfernt.

2.5 Attr(@)

Wird von FHEM aufgerufen, sobald ein Benutzer ein Modul-Attribut ändern, bzw. entfernen möchte. Spezielle Attribute wie „HTTPS“, “SSL_Key“, “SSL_Zert“ wurden bereits in der Methode „Initialize()“ aufgenommen und können nun hier geändert werden. Dabei dient das Attribut „HTTPS“ zur Verwendung eines SSL-Tunnels. Die Attribute „SSL_Zert“ und „SSL_Key“ dienen dabei nur zur Authentifizierung und Verschlüsselung des SSL-Tunnels. Bei der Verwendung von SSL wird ein Perl-Paket „IO::Socket::SSL“ vorausgesetzt. Mit „eval“ wird hier ein „Try/Catch“ durchgeführt, sodass nicht das ganze Programm abstürzt, falls das Paket nicht Verfügbar sein sollte.

Perl-Code:

```
eval "require IO::Socket::SSL";
```

CodeSnippet 40: Einbinden des SSL-Pakets

Bei Verfügbarkeit des Paketes werden danach die Attribute „SSL_Zert“ und „SSL_Key“ durch den Aufruf der eigenen Methode gesetzt. Sollten bei dem Setzen

der Attribute keinerlei Fehler auftreten kann der SSL-Tunnel etabliert werden. Treten Fehler auf, wird eine entsprechende Fehlermeldung geliefert.

2.6 NotifyFn($$)

Wird von FHEM aufgerufen, wenn ein neues Modul oder Device geladen wird. Die Methode wird von uns genutzt, um nicht definierte, oder neu geladene Module/Devices in eine Liste einzutragen. Dazu wird als allererstes unterschieden, ob es das Hauptmodul, oder ein Client ist. Ist es ein Client-Modul, wird die Methode sofort wieder verlassen. Andernfalls wird überprüft, ob das Gerät bereits in der Liste enthalten ist. Ist dies nicht der Fall, wird es eingetragen.

2.7 ProzessOPTIONSResponse($$)

Liefert dem Client, bei einer HTTP-OPTIONS Anfrage, die vom Modul unterstützten HTTP-Services in der HTTP1.1-Defition laut RFC.

2.8 ProzessNotImplementedResponse($$)

Liefert dem Client HTTP-Fehlercode „501 – Not Implemented“, falls der angeforderte HTTP-Service vom Modul nicht unterstützt wird.

2.9 GetHTTPVersion($$)

Aufruf:

Perl-Code:
TheOpenTransporter_GetHTTPVersion($ModulHastable,HTTP/1.1);

CodeSnippet 41: Aufruf GetHTTPVersion()

Mit dieser Methode wird die Angeforderte HTTP-Version aus dem Request-Header ermittelt. Dazu muss die Modul-Hashtable und jener Teil vom Header, welcher die HTTP-Version enthält, übergeben werden. Handelt es sich nicht um eine der beiden Versionen 1.0 oder 1.1 wird die Verbindung mit einer Meldung, dass die angefragte Version nicht unterstützt wird, zurückgewiesen. Genauso wird die Verbindung zurückgewiesen, sollte ein anderes Protokoll als HTTP angefordert worden sein.

2.10 ProzessGETResponse($$)

Diese Methode verarbeitet eine HTTP-GET Anfrage eines Clients.

Zu Beginn wird die HTTP-Version ermittelt und die angeforderte Funktion und die gesendeten Daten extrahiert. Die Rohdaten werden mit „DecodeRawData($$$)“ (4.2.6.12) in das interne Datenformat umgewandelt, wobei diese eine Parameter-Hashtable liefert. Im nächsten Schritt werden die Headerlines der Anfrage

ausgelesen und zeilenweise in ein Array geschrieben. Leere Zeilen werden nicht beachtet. Bei diesem Vorgang wird gleichzeitig beachtet, dass maximal eine vorher definierte Anzahl von Headerlines eingelesen wird. Dadurch wird sichergestellt, dass der Dienst durch eine unendliche Anzahl von Headerlines in keine Endlosschleife verfallen kann. Nun wird eine zweite Hashtable (Funktions-Hashtable) angelegt, die die notwendigen Informationen über die Anfrage enthält, außerdem wird die Parameter-Hashtabel ebenfalls in diese als Referenz geschrieben. Dies ist nötig, da die Parameter-Daten klar von den Anfrage-Daten abgetrennt werden sollen. Nun wird „ProzessInfo($$$)“ (4.2.6.13) aufgerufen, wodurch die eigentliche Anfrage an die Schnittstelle abgearbeitet und Daten in die Funktions-Hashtable geschrieben wird. Anschließend wird aus den gelesenen Headerlines herausgesucht, welche Content-Types vom Client unterstützt werden. Sollte kein entsprechender Eintrag gefunden werden, wird per Default „text/html“ gewählt. Diesen Typen sollte jeder Browser mindestens unterstützen. Weiter wird die Art der Verbindung heraus gesucht, sollte keine übergeben worden sein, wird per Default „keep-alive“ gewählt. Das bedeutet, dass die Verbindung so lange aufrechterhalten wird, bis der Timeout abgelaufen, oder die Verbindung geschlossen wurde. Nun wird überprüft, ob der in der Antwort generierte Content-Type vom Client unterstützt wird. Sollte dies nicht der Fall sein, wird wiederum „text/html“ als Standard gesetzt. Der Response-Code wird derzeit bei einer erfolgreichen Antwort immer auf 200 gesetzt, mit einer einzigen Ausnahme: Die generierte Antwort ist leer. Dies lässt schließen, dass die angeforderte Funktion nicht vorhanden ist. Deshalb wird in diesem Fall „404 Not Found“ als Response-Code zurückgegeben.

Sofern die Antwort Werte enthält wird nun noch der Header der Antwort zusammen-gesetzt. Dazu wird zunächst überprüft, ob mit HTTP/1.0 oder HTTP/1.1 übertragen wird. Dies ist wichtig, weil die Header sich unterscheiden. An den Header wird schlussendlich noch die generierte Antwort angehängt und anschließend das Gesamtpaket abgeschickt.

Nun zu dem Unterschied zwischen HTTP1.0 und 1.1. Im ersten Fall wird laut Definition auf jeden Fall die Verbindung getrennt, wobei bei 1.1 diese Entscheidung dem Client überlassen wird. Geschlossen wird die Verbindung, indem das temporäre Modul (Clone) des Haupt-Moduls entfernt wird.

2.11 ProzessPOSTResponse($$)

Diese Methode verarbeitet eine HTTP-POST Anfrage eines Clients.

Zu Beginn wird die HTTP-Version ermittelt und die angeforderte Funktion und die gesendeten Daten extrahiert. Im Unterschied zu GET wird nun zuerst der Header ausgelesen, indem er Zeilenweise in ein Array geschrieben wird (Leere Zeilen werden nicht beachtet). Bei diesem Vorgang wird gleichzeitig beachtet, dass maximal eine vorher definierte Anzahl von Headerlines eingelesen wird. Dadurch wird sichergestellt, dass der Dienst durch eine unendliche Anzahl von Headerlines in keine Endlosschleife verfallen kann. In dem Header ist die Content-Length enthalten, welche angibt, wie viele Daten folgen. Diese wird extrahiert und anschließend entsprechend viele Daten gelesen.

Die Rohdaten werden mit „DecodeRawData($$$)“ (4.2.6.12) in das interne Datenformat umgewandelt. Nun wird eine zweite Hashtable angelegt. Dies ist nötig, da die Funktions-Daten klar von den umgewandelten Daten abgetrennt werden sollen. Allerdings wird die Parameter-Hashtable mit einem Zeiger in der Funktions-Hashtable hinterlegt. Nun wird „ProzessInfo($$$)“ (4.2.6.13) aufgerufen, wodurch die eigentliche Anfrage an die Schnittstelle abgearbeitet und in die Daten-Hashtable geschrieben wird. Anschließend wird aus den gelesenen Headerlines herausgesucht, welche Content-Types vom Client unterstützt werden. Sollte kein entsprechender Eintrag gefunden werden, wird per Default „text/html“ gewählt. Diesen Typen sollte jeder Browser mindestens unterstützen.

Weiter wird die Art der Verbindung heraus gesucht, sollte keine übergeben worden sein, wird per Default „keep-alive“ gewählt. Das bedeutet, dass die Verbindung so lange aufrechterhalten wird, bis der Timeout abgelaufen, oder die Verbindung geschlossen wurde.

Nun wird überprüft, ob der in der Antwort generierte Content-Type vom Client unterstützt wird. Sollte dies nicht der Fall sein, wird wiederum „text/html“ als Standard gesetzt.

Der Response-Code wird derzeit bei einer erfolgreichen Antwort immer auf 200 gesetzt, mit einer einzigen Ausnahme: Die generierte Antwort ist leer. Dies lässt schließen, dass die angeforderte Funktion nicht vorhanden ist. Deshalb wird in diesem Fall „404 Not Found“ als Response-Code zurückgegeben.

Sofern die Antwort Werte enthält wird nun noch der Header der Antwort zusammen-gesetzt. Dazu wird zunächst überprüft, ob mit HTTP/1.0 oder HTTP/1.1 übertragen

wird. Dies ist wichtig, weil die Header sich unterscheiden. An den Header wird schlussendlich noch die generierte Antwort angehängt und anschließend das Gesamtpaket abgeschickt.
Nun zu dem Unterschied zwischen HTTP1.0 und 1.1. Im ersten Fall wird laut Definition auf jeden Fall die Verbindung getrennt, wobei bei 1.1 diese Entscheidung dem Client überlassen wird. Geschlossen wird die Verbindung, indem das temporäre Device des Moduls entfernt wird.

2.12 DecodeRawData($$$)

Diese Methode unterscheidet verschiedene Content-Types und wandelt die übertragenden Roh-Daten in das interne Standardformat um. Bei einem unbekannten Content-Type wird „undef“ zurückgegeben. Verwendung findet die Methode beim Empfangen von Daten mit POST oder GET. Es wird zurzeit zwischen „application/x-www-form-urlencoded“ und „application/json“ unterschieden. Ersteres trifft ein, wenn die Daten über die URL-Kodiert übertragen werden. In diesem Fall wird der Teil der URL, hinter dem „?“ erwartet. Zuerst werden die Daten nach dem Zeichen „&“ gesplittet, wodurch Wertpaare vom Format „Bezeichner=Wert“ entstehen. Daher wird anschließend erneut nach dem Zeichen „=“ gesplittet. So sind nun Wert und Bezeichner voneinander getrennt. Die nun getrennten Paare werden in eine Hashtable geschrieben, wobei der Bezeichner als Schlüssel und der Wert als Wert hinterlegt werden. Die daraus entstandene Hashtable wird dann als Funktionswert zurückgegeben.
Soll ein JSON-Objekt für die interne Verarbeitung umgewandelt werden, wird vom Prinzip hier das selbe getan. Allerdings sind hier die Trennzeichen andere. Zu aller erst werden die äußeren Klammern entfernt. Anschließend werden die Wertepaare voneinander getrennt, in dem nach jedem '“,“' gesplittet wird. Anschließend werden Bezeichner und Wert mit '“:“'
voneinander gesplittet und das Ergebnis ebenfalls in eine Hashtable geschrieben.
So ist sichergestellt, dass die internen Daten jederzeit im gleichen Format vorliegen, egal ob JSON oder die URL zum übertragen genutzt wurden.

2.13 ProzessInfo($$$)

Die Methode „ProzessInfo“ ruft die passende Funktion, wenn vorhanden, auf. Dies bewerkstelligt sie, indem eine Hash-Table "FntTable" auf die Existenz der übergebenen Funktion durchsucht wird. Bei einem Treffer wird die entsprechende

Referenz auf die Funktion mit „CallFnt()“ aufgerufen, bei keinem Treffer, wird „undef“ zurückgegeben. Zusätzlich sei erwähnt, dass die Übergabeparameter unabhängig von Groß- und Kleinschreibung sind.

2.14 CallFnt(@)

Die Methode „CallFnt()“ ruft die ihr übergebende Funktion auf. Dazu wird per „shift“-Operation das erste Element des übergebenen Array aus dem Array gezogen (Array enthält danach dieses Element nicht mehr). Das nun herausgezogene Element beinhaltet den Funktionsnamen.

Achtung:

Es muss das „strict“ ausgeschaltet werden, damit eine Variable als Funktionsnamen dienen darf. Dies würde „strict“ sonst Verbieten. Anschließend muss „strict“ wieder angeschaltet werden.

Perl-Code:

```
sub CallFnt(@)
{
	my $function = shift;
	no strict "refs";
	return &{$function}(@_);
	use strict "refs";
}
```

CodeSnippet 42: Funktion CallFnt()

2.15 ReturnMessage($$)

Aufruf:

Perl-Code:

```
TheOpenTransporter_ReturnMessage($hash, "Test-Message", 1, 4);
```

CodeSnippet 43: Aufruf ReturnMessage()

Wird verwendet um einen String in ein JSON-Objekt zu verpacken. Gleichzeitig wird dem Objekt zugewiesen, ob es eine normale Meldung, oder eine Fehlermeldung ist. Noch dazu wird der String in das FHEM-Log geschrieben.

Es gibt vier Parameter, wobei der erste ein die lokale Hashtable ist, der zweite ein String ist, welcher die Meldung darstellt. Der dritte ist entweder mit irgendeinem Wert (Fehlermeldung) oder mit „undef“ (normale Meldung) zu belegen. Als letztes kann noch eine Zahl größer 0 für das gewünschte LogLevel angegeben werden.

2.16 Log($$$)

Dient des setzen des Modulnamens als Präfix vor jedem Log Eintrag. Auch kann der LogLevel bestimmt werden.

2.17 ArrayToJSONArray(@)

Aufruf:

Perl-Code:

```
TheOpenTransporter_ArrayToJSONArray(@BeispielArraymitJSONObjekten);
```

CodeSnippet 44: Aufruf ArrayToJSONArray()

Mit dieser Methode werden mehrere einzelne JSON-Objekte zu einem JSON-Array zusammengefasst. Dazu muss ein Array mit JSON-Objekten übergeben werden. Dieses wird dann mit dem Bezeichner „Content“ und den Steuersymbolen „[„ und „]“ umschlossen.

2.18 DateTimeEqual(@)

Aufruf:

Perl-Code:

```
TheOpenTransporter_DateTimeEqual(@ArrayMitStartundEndDatum);
```

CodeSnippet 45: Aufruf DateTimeEqual()

Mit dieser Funktion kann überprüft werden, ob ein Datum X vor einem Datum Y liegt. Erwartet wird dafür ein Array mit einer geraden Anzahl von Argumenten. Die Anordnung der Attribute eines Datums lautet „YYYY MM DD hh mm“. Das Datum Y wird an das Datum X hinten angehängt (YYYY MM DD hh mm YYYY MM DD hh mm). Es muss dabei jedoch nicht jedes Argument des Datums angegeben werden. So kann das Datum ab jeder Stelle „abgeschnitten“ werden. Allerdings müssen beide Daten an der gleichen Stelle abgeschnitten werden.

Zur Funktionalität ist zu sagen, dass zu aller erst die Position des ersten Wertes des Enddatums ermittelt wird. Anschließend werden alle Wertepaare (Also jeweils Jahr, Monat etc.) nacheinander verglichen. Sobald sich zwei Werte eines Paares unterscheiden, muss das Datum X entweder vor oder hinter Y liegen. Wenn der Startwert größer ist, wird eine 1 geliefert. Ist er kleiner, wird eine -1 geliefert. Sollten alle Werte gleich sein, wird eine 0 geliefert.

3 Implementierung der Http-Schnittstellen

Hier werden nun die Funktionen, welche der Benutzer per HTTP-Anfrage aufrufen kann, beschrieben.

3.1 Device Funktionen

3.1.1 FntGetDevice($$)

Aufruf:

HTTP-Anfrage:

```
Http://<Server-IP>:<Port>/GetDevice?NAME=<dev_name>
```

CodeSnippet 46: Aufruf GetDevice()

Diese Funktion nimmt als Parameter „NAME“ den Namen eines Devices entgegen und liefert ein JSON-Objekt, welches den Inhalt dieses FHEM-Objektes enthält. Sollte das angeforderte Device nicht vorhanden sein, wird eine entsprechende Meldung ausgegeben und im Level 4 geloggt. Es werden alle Werte ausgelesen, welche in der Hashtable des Devices stehen. Zusätzlich werden dem JSON-Objekt die Attribute des Objektes, sowie die Sets und die gesetzten Attribute angehängt. Außerdem wird es mit einer „guid“ versehen, welche der „NR“ des Devices entspricht.

JSON-Objekt:

```
 {
"NAME": "FunkSchalter01",
"DEF": "1111 11",
"READINGS": {
"state": {
"TIME": "2012-02-09 22:49:20",
"VAL": "off"
}
},
"TYPE": "FS20",
"CODE": {
"1": "1111 11"
},
"IODev": {
"TCUL_TIME": "2012-02-10 03:40:52",
"DEF": "/dev/ttyACM0 1234",
"TCUL_MSGCNT": 3,
"NR": 7,
"RAWMSG": "T3A0D00A6003E",
"FHTID": "1234",
"MatchList": {
"E:CUL_TX": "^TX[A-F0-9]{10}",
"B:CUL_HOERMANN": "^R.........",
"A:CUL_RFR": "^[0-9A-F]{4}U.",
"6:CUL_WS": "^K.....",
"2:BS": "^81..(04|0c)..0101a001a5cf",
"C:ESA2000": "^S.............................$",
```

```
"4:FHT": "^81..(04|09|0d)..(0909a001|83098301|c409c401)..",
"5:KS300": "^810d04..4027a001",
"3:FS20": "^81..(04|0c)..0101a001",
"1:USF1000": "^81..(04|0c)..0101a001a5ceaa00....",
"8:HMS": "^810e04....(1|5|9).a001",
"D:CUL_IR": "^I.....",
"7:CUL_EM": "^E0.................$",
"9:CUL_FHTTK": "^T[A-F0-9]{8}"
},
"NAME": "TCUL",
"initString": "X21",
"PARTIAL": "",
"TYPE": "CUL",
"DeviceName": "/dev/ttyACM0",
"FD": 12,
"USBDev": null,
"RSSI": "-43",
"STATE": "Initialized",
"VERSION": "V 1.40 CUL868",
"Clients": ":FS20:FHT:FHT8V:KS300:USF1000:BS:HMS:
:CUL_EM:CUL_WS:CUL_FHTTK:CUL_RFR:CUL_HOERMANN: :ESA2000:CUL_IR:CUL_TX"
},
"XMIT": "1111",
"BTN": "11",
"NR": 8,
"ATTRIBUTES": "room comment alias IODev follow-on-for-timer:1,0 do_not_notify:1,0 ignore:0,1
dummy:1,0 showtime:1,0
model;fs20hgs,fs20hgs,fs20pira,fs20piri,fs20s20,fs20s8,fs20s4,fs20s4a,fs20s4m,fs20s4u,fs20s4ub,fs
20sd,fs20sn,fs20sr,fs20ss,fs20str,fs20tfk,fs20tfk,fs20tk,fs20uts,fs20ze,fs20as1,fs20as4,fs20di,fs20du,
fs20ls,fs20ms2,fs20rst,fs20sa,fs20sig,fs20st,fs20sv,fs20usr loglevel:0,1,2,3,4,5,6 Thermostrahler
room structexclude webCmd eventMap",
"STATE": "off",
"SETS": "dim06% dim100% dim12% dim18% dim25% dim31% dim37% dim43% dim50% dim56%
dim62% dim68% dim75% dim81% dim87% dim93% dimdown dimup dimupdown off off-for-timer on
on-100-for-timer-prev on-for-timer on-old-for-timer on-old-for-timer-prev on-till ramp-off-time ramp-on-
time reset sendstate timer toggle",
"guid": 8,
"EATTR": { }

}
```


CodeSnippet 47: Beispiel JSON-Objekt getDevice()

Die Werte werden komplett unverändert aus FHEM ausgelesen!

3.1.2 FntGetAllDevices($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/GetAllDevices?[OPT=showAll]

CodeSnippet 48: Aufruf GetAllDevices()

Mit dieser Funktion werden alle in FHEM enthaltenen Devices ausgegeben.

Die Funktionalität ist analog zu GetDevice, da eben diese Funktion dafür genutzt wird. Allerdings müssen keinerlei Parameter angegeben werden. Die einzelnen

JSON-Objekte werden zu einem JSON-Array zusammengefasst und an den Client gesendet.

Devices sind alle FHEM-Objekte, ausgenommen vom Typ:

- FileLog
- notify
- at
- structure
- FHEMWEB
- TheOpenTransporter
- weblink
- _internal_
- sequence

Um eine Möglichkeit zu bekommen wirklich alle Devices von FHEM anzeigen zu können, wurde ein optionaler Parameter „OPT“ hinzugefügt, welches mit „showAll“ zu belegen ist.

3.1.3 FntNewDevice($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/NewDevice?NAME=<dev_name>&TYPE=<dev_type>&DEF=<Parameter (beliebig viele)>

CodeSnippet 49: Aufruf NewDevice()

Diese Funktion definiert ein neues Device in FHEM. Es werden die Parameter „NAME“, „TYPE“ und DEF zwingend benötigt. „TYPE“ beinhaltet den Typen des Devices. In „DEF“ müssen alle für den Device benötigten Parameter angegeben werden, diese können beliebig viele sein. Mit diesen Informationen wird mittels der FHEM-Funktion „CommandDefine()“ das Device definiert.

Sollten weniger Parameter übergeben werden, kommt eine entsprechende Fehlermeldung, zudem wird diese im LogLevel 4 abgelegt. Bei einem Fehler mit genügend Parametern wird die FHEM Fehlermeldung von „CommandDefine()“ geliefert. Bei einer erfolgreichen Durchführung wird „true“ geliefert.

Achtung:

Es ist wichtig, dass die Anzahl der Parameter bzw. die Art der Parameter aus der Dokumentation des entsprechenden FHEM-Devices entnommen wird, da diese Methode bewusst sehr Allgemein gehalten ist. So kann jede Komponente definiert werden, welche auch über FHEMWEB definiert werden kann.

3.1.4 FntDelDevice($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/DelDevice?NAME=<dev_name>

CodeSnippet 50: Aufruf DelDevice()

Diese Funktion löscht ein Device aus FHEM. Als Parameter muss der Name übergeben werden. Gelöscht wird das Device mit der FHEM-Methode „CommandDelete()“. Anschließend wird bei Erfolg „true“, sonst die Fehlermeldung der FHEM-Methode zurückgegeben.

3.1.5 FntSetDevice($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/SetDevice?NAME=<dev_name>&<attr_type1>=<attr_update1>&<attr_type2>=<attr_update2>&<attr_typeN>=<attr_updateN>

CodeSnippet 51: Aufruf SetDevice()

Diese Funktion gibt die Möglichkeit Attribute eines bestehenden Devices zu verändern.

Es können definierte Attribute eines Devices (welche kommt auf den Device-Typ an) gesetzt, verändert und gelöscht werden. Auch die in „DEF“ hinterlegten Werte können verändert werden. Beim Verändern von DEF ist aber zu beachten, dass Device spezifische Regeln eingehalten werden müssen. Diese sind der FHEM-Dokumentation zu entnehmen.

Es müssen mindestens 2 Parameter übergeben werden. Der Name des Devices, sowie den Namen und den Wert eines zu aktualisierenden Attributes. Es können beliebig viele zu aktualisierenden Attribute angegeben werden.

Es werden alle möglichen Attribute des Devices durchsucht, falls das zu aktualisierenden Attribut vorhanden ist wird dieses mit der FHEM-Methode „CommandAttr()“ bzw. „CommandSet()“ aktualisiert. Sollte für ein Attribut der Wert „undef“ übergeben worden sein, wird die Funktion „CommandDeleteAttr()“ aufgerufen und das Attribut wird gelöscht.

Sollten nicht genügend Parameter übergeben werden, wird eine Fehlermeldung geliefert und im Level 4 geloggt. Stimmt die Parameteranzahl, werden im weiteren Fehlerfall die bekannten FHEM-Fehlermeldungen ausgegeben.

Die folgende Anfrage würde das Attribut „room“ von „Testdevice“ entfernen.

HTTP-Anfrage:
Http://<Server-IP>:<Port>/SetDevice?NAME=Testdevice&room=undef

CodeSnippet 52: Löschen eines Attributes mit SetDevice()

Achtung:

Sollte eine Ansicht des Devices in FHEMWEB aktuell angezeigt werden, während diese Funktion aufgerufen wird, darf auf keinen Fall die Website per „refresh“ aktualisiert werden. Dadurch schreibt FHEMWEB die aktuell angezeigten Attribute neu in das Device und somit wird die vorherige Modifikation rückgängig gemacht.

3.1.6 FntGetNonDefinedDevices($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/GetNonDefinedDevices

CodeSnippet 53: Aufruf GetNonDefinedDevices()

Diese Methode hat keine Parameter. Sie liefert alle in FHEM aufgetauchten, nicht definierten Devices. Allerdings lediglich Geräte, welche sich selbst bei FHEM melden. So wird zum Beispiel niemals ein Funkschalter in der Liste auftauchen. Dieser wird seitens FHEM nicht erkannt, da er zunächst in FHEM definiert und anschließend in den Lernmodus versetzt werden muss. Durch einmaliges ein oder ausschalten wird dieser mit FHEM gepaart. Eine FHT (Heizungssteuerung) hingegen würde angezeigt werden.

Es werden der Name und die Adresse der Devices geliefert.

Es wird die Liste, welche in „NotifyFn()“ (siehe 4.2.6.6) erzeugt wird, genutzt, um ein JSON-Objekt zu erzeugen und zu liefern.

3.2 Struktur Funktionen

3.2.1 GetStructure($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/GetStructure?NAME=<structure_name>

CodeSnippet 54: Aufruf GetStructure()

Diese Funktion nimmt das Parameter „NAME“ entgegen und liefert ein JSON-Objekt, welches den Inhalt des zugehörigen FHEM-Struktur-Objektes enthält. Sollte die angeforderte Struktur nicht vorhanden sein, wird eine entsprechende Fehlermeldung geliefert. Es ist möglich jede beliebige FHEM-Struktur anzufordern.

3.2.2 GetAllStructures($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/GetAllStructures

CodeSnippet 55: Aufruf GetAllStructures

Arbeitet mit der Funktion „GetStructure()“, indem sie für alle vorhandenen Strukturen aufgerufen wird. Jedes JSON-Objekt jeder einzelnen Struktur wird dann in ein JSON-Array mit dem Bezeichner „Content“ abgelegt. Dieses wird dann zurückgegeben.

3.2.3 FntNewStructure($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/NewStructure?NAME=<name>&DEF=<Dev1...DevN>&ATTR=<structure_type>

CodeSnippet 56: Aufruf NewStructure()

Mit dieser Methode kann eine neue „structure“ definiert werden.
Zwingend muss der Name, ein Bezeichner und die Geräte angegeben werden. Der übergebene Name wird der Struktur zugewiesen. Mit „DEVICES“ werden alle Devices angegeben, welche der Struktur hinzugefügt werden. „ATTR“ kann ein beliebiger Bezeichner sein, also der Strukturtyp. Sollte eines Dieser Parameter nicht angegeben werden, wird eine entsprechende Fehlermeldung zurückgegeben. Sollte bei der Definition seitens FHEM etwas schief gehen, wird diese Fehlermeldung ausgegeben. Ein Beispiel Aufruf mit 4 Devices ist hier zu sehen:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/NewStructure?NAME=teststruct2&ATTR=room&DEVICES=dev1 dev2 dev3 dev4

CodeSnippet 57: Beispielaufruf NewStructure()

Hier wäre die Struktur vom Typ „room“. (Nähere Informationen zu der Verwendung von Strukturen sind der FHEM-Dokumentation zu entnehmen.) Bei erfolgreichem Vorgang wird eine „True“-Meldung geliefert.

3.2.4 FntDeleteStructure($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/DeleteStructure?NAME=<structure_name>

CodeSnippet 58: Aufruf DeleteStructure()

Mit dieser Methode kann eine komplette „structure“ gelöscht werden. Dazu wird die Methode FntDelDevice() genutzt. (siehe 4.2.7.4)

3.2.5 FntAddDeviceInStructure($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/AddDeviceInStructure?NAME=<stucture_name>&DEVICES=<dev1> <devN>

CodeSnippet 59: Aufruf AddDeviceInStructure()

Mit dieser Methode werden weitere Devices einer Struktur zugewiesen. Dazu müssen die Parameter „NAME“ und „DEVICES“ übergeben werden. Zum einen der Name und dazu die Namen der Devices, welche in die Struktur hinzugefügt werden sollen. Wenn die gewählte Struktur existiert, wird das Device in „state“ und „dev“ eingetragen, eine Erfolgsmeldung zurück gegeben und im Level 5 geloggt. Devices werden allerdings nur dann eingetragen, wenn es noch nicht in der Struktur existiert. Sollte keines der angegebenen Devices in die Struktur hinzugefügt werden können, oder die Struktur gar nicht existieren, wird eine entsprechende Fehlermeldung geliefert und im Level 4 geloggt.

3.2.6 FntDeleteDeviceInStructure($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/DeleteDeviceInStructure?NAME=<structure_name>&DEVICES=<dev_name> <dev_nameN>

CodeSnippet 60: Aufruf DeleteDeviceInStructure()

Mit dieser Methode können ein oder mehrere Devices aus einer Struktur gelöscht werden.

Die Parameter, welche angegeben werden müssen lauten „NAME“ und „DEVICES“. In „DEVICES“ können die zu löschenden Devices angegeben werden. Es dürfen

aber maximal so viele Devices angegeben werden, dass mindestens eines in der Struktur verbleibt. Diese Eigenschaft wurde aus der Verwendung von FHEM übernommen. Sollten mehr angegeben werden, wird eine Entsprechende Fehlermeldung zurückgegeben. Wenn der Vorgang erfolgreich war (d.h. Es wurde mindestens ein Device gelöscht), wird eine Erfolgsmeldung geliefert und im Level 5 geloggt. Im Fehlerfall wird eine Fehlermeldung geliefert und im Level 4 geloggt. Weiterhin wird die Struktur insofern angepasst, dass der erste Device als „attr“ und alle anderen als „state“ eingetragen werden. Dies wurde so umgesetzt, da dies ebenfalls auf diese Art und Weise von FHEM gehandhabt wird. Weiter werden nach dem entfernen des Devices alle doppelten, am Ende und Anfang vorkommenden Leerzeichen entfernt.

3.3 Log Funktionen

3.3.1 FntGetAvailableLogFiles($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/GetAvailableLogFiles

CodeSnippet 61: Aufruf GetAvailableLogFiles()

Diese Funktion ist parameterlos und liefert eine Zusammenstellung von Fhem-Objekten, welche ein Logfile besitzen.

JSON-Objekt:
{"Content":[{"LOGFILE":"/var/log/fhem/fhem-%Y-%m.log","NAME":"global","TYPE":"_internal_","guid":1}, {"LOGFILE":"/var/log/fhem/fhem-%Y-%m.log","NAME":"Logfile","TYPE":"FileLog","guid":5}]}

CodeSnippet 62: JSON: Verfügbare Logfiles

Content ist ein JSON-Objekt, welches alle weiteren kapselt. Darin befinden sich die eigentlichen JSON-Objekte. Übergeben werden die Member „Logfile“ mit dem Pfad als Value, „Name“ mit dem Namen des FHEM-Objektes, „TYPE“ mit dem FHEM-Type und „GUID“ mit der FHEM-Devicenumber. Aus der obigen Antwort kann beispielsweise abgelesen werden, das die Objekte „global“ und „Logfile“ jeweils ein Logfile besitzen. Auch der zugehörige Pfad mit Datei können abgelesen werden. Für diese Objekte könnten nun mit GetLogFromDevice($$) die Logfiles angefordert werden.

3.3.2 FntGetLogFromDevice($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/GetLogFromDevice?NAME=<dev_name>

CodeSnippet 63: Aufruf GetLogFromDevice()

Diese Funktion ist mit FntGetLogFromDeviceInPeriod($$) zusammengefasst worden. Daher ist es möglich diese Funktion nur mit einem Parameter, also einem Device, oder weiteren Parametern, welche ein Start- und ein End-Datum angeben, aufzurufen. Das übergebene Device muss eine Log-Datei enthalten. (Derartige Objekte können mit der Funktion GetAvailableLogFiles($$) herausgefunden werden) Sofern ein bestimmter Zeitraum gewünscht ist, muss mindestens ein Start- und End-Jahr angegeben werden. Es ist aber auch möglich den Zeitraum Minutengenau zu bestimmen.

Da es bei dieser Methode einige Parameter verfügbar sind, werden sie folgend aufgelistet:

- NAME : zwingend anzugeben!
- SYEAR : Startjahr
- SMONTH : Startmonat
- SDAY : Starttag
- SHOUR : Startstunde
- SMINUTE : Startminute
- EYEAR : Endjahr
- EMONTH : Endmonat
- EDAY : Endtag
- EHOUR : Endstunde
- EMINUTE : Endminute

Diese Parameter können in vielen verschiedenen Kombinationen angewendet werden, welche in den folgenden Beispielen erläutert werden. Die einfachste Variante ist, nur dem Namen anzugeben, welcher immer angegeben werden muss.

HTTP-Anfrage:

```
Http://<Server-IP>:<Port>/GetLogFromDevice?NAME=<dev_name>
```

CodeSnippet 64: Aufruf GetLogFromDevice() mit Namen

In diesem Fall wird die, dem Device zugehörige, aktuelle Logdatei ausgeben.
Es ist aber auch möglich einen Zeitraum auf verschiedenste Arten anzugeben.

HTTP-Anfrage:

```
Http://<Server-IP>:<Port>/GetLogFromDevice?NAME=<dev_name>&SYEAR=2011&EYEAR=2012&EMONTH=10
```

CodeSnippet 65: Aufruf GetLogFromDevice() mit Zeitraum

Diese Variante gibt alle Logeinträge zwischen dem 01.01.2011 und dem 31.10.2012 aus.

HTTP-Anfrage:

```
Http://<Server-IP>:<Port>/GetLogFromDevice?NAME=<dev_name>&SYEAR=2011&EYEAR=2012&EDay=2
```

CodeSnippet 66: Aufruf GetLogFromDevice() mit Namen

Hier alle Logeinträge zwischen dem 01.01.2011 und dem 02.12.2012. Es ist ersichtlich, dass für alle End-Werte per Default der spätest mögliche gewählt wird. Also ist der Standardmonat immer „12“. Der Standardtag variiert allerdings zwischen

28-31! Bei den Anfangswerten wird immer der frühestmögliche Wert gewählt. Als Standardstartjahr ist 2000 definiert.

Zur Funktionsweise ist zu sagen, dass zunächst über das Startjahr und den Startmonat die erste Logdatei bestimmt wird. Dazu wird der Pfad dem FHEM-Objekt entnommen und die Platzhalter „%Y“ mit dem Startjahr und „%m“ mit dem Startmonat ersetzt.

Danach wird diese Datei geöffnet und die Datei wird Zeile für Zeile gelesen. Aus jeder Zeile wird, das Datum extrahiert und mit den Start- und End-Daten verglichen. Wenn das Datum der Logzeile in dem Bereich liegt, wird die Zeile in die Ausgabe aufgenommen. Sollte die Zeile kein Datum haben, wird diese ebenfalls hinzugefügt, sofern bereits Zeilen in die Ausgabe aufgenommen wurden.

Achtung:

Es sei zu erwähnen, dass die Logs als Text übertragen werden. Geplant war ursprünglich die Übertragung per JSON. Allerdings macht dies für Logfiles wenig Sinn, da sie anschließend wieder zurück gewandelt werden müssten. Dadurch würde auf beiden Seiten unnötige Rechenlast entstehen! Zumal die Logdateien schnell sehr Groß werden können.

Weiterhin ist bei dieser Methode das logging mit Bedacht zu nutzen. Diese Methode greift in Echtzeit auf das Log zu, in dem Zeile für Zeile gelesen wird. Sollte diese Funktion dabei etwas in das Log hineinschreiben, wird logischerweise eine Endlosschleife erzeugt!

Auch zu beachten ist, dass diese Funktion nur bei absoluten Dateinamen oder Dateinamen mit allgemeiner Syntax von Jahr (%Y) und/oder Monat (%m) Verwendung findet.

3.3.3 FntGetSVGFromDevice($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/GetSVGFromDevice?PARAMS=<dev_name>

CodeSnippet 67: Aufruf GetSVGFromDevice()

Ruft ohne jegliche Überprüfung die SVG_Render-Methode aus dem Modul „98_SVG.pm“ auf.

3.4 Script Funktionen

3.4.1 FntGetAllScripts($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/GetAllScripts

CodeSnippet 68: Aufruf GetAllScripts()

Gibt alle auf dem Server hinterlegten Scripte („notify“ / „at“) in einem JSON-Objekt aus.

Auch hier werden die einzelnen Scipt-Objekte in einem JSON-Array gekapselt.

Folgende Member werden übergeben: „Volatile“, „Name“, „Readings“, „DEF“, „CHANGED“, „TYPE“, „REGEXP“, „LOGFILE“, „NR“, „STATE“, „guid“.

Bis auf GUID sind dies alles aus FHEM entnommene Attribute und deren Bedeutung kann in der FHEM-Dokumentation nachgelesen werden. GUID wurde hinzugefügt, damit „Sproutcoure“ einen eindeutigen Identifikator ansprechen kann. Dieser muss zwingend „guid“ heißen. In der Umsetzung hat „guid“ keinen anderen Wert als „NR“.

Sollten keine Scripte vorhanden sein, wird ein leeres JSON-Array geliefert.

3.4.2 FntGetScript($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/GetScript?NAME=<script_name>

CodeSnippet 69: Aufruf GetScript()

Diese Funktion arbeitet identisch zu „GetAllScripts()“, allerdings wird hier nur ein Script als JSON-Objekt geliefert, welches per Parameter (Der Name des Skriptes) angefordert wurde.

Allerdings ist hier eine Fehlerbehandlung implementiert. Sollte ein Script nicht vorhanden sein, wird eine entsprechende Fehlermeldung ausgegeben. Sollte es sich um kein „notify“ / „at“ handeln, wird ebenfalls eine Fehler-Meldung ausgegeben. Geloggt werden diese im Level 4.

3.4.3 FntAddScript($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/AddScript?NAME=<script_name>&REGEXP=<regexp>&COM=<commands>

CodeSnippet 70: Aufruf AddScript()

Mit dieser Funktion kann ein neues Script zu FHEM hinzugefügt werden. Erforderlich sind hierfür die Parameter „NAME“, „REGEXP“ und „COM“. Der Inhalt der Parameter entspricht jenen, welche dem „notifiy“ auch in FHEM übergeben werden müssen.
Sollten nicht alle Parameter angegeben werden, wird eine entsprechende Fehlermeldung geliefert, welche im LogLevel 4 abgelegt wird. Andernfalls wird das Script als notify in FHEM definiert. Sollte dabei etwas schiefgehen, wird die FHEM-Fehlermeldung zurückgegeben und auch dies im LogLevel 4 abgelegt. Wenn das Script erfolgreich definiert werden konnte, wird „true“ geliefert und im LogLevel 5 geloggt.

Achtung:

Bei dieser Funktion kann nur ein „notify“ angelegt werden. (siehe 4.4.6)

3.4.4 FntDeleteScript($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/DeleteScript?NAME=<script_name>

CodeSnippet 71: Aufruf DeleteScript()

Diese Methode erlaubt es einem Benutzer ein Script zu löschen. Es wird ein einziges Parameter benötigt. Der Name des Skriptes. Sollte das Script nicht existieren, wird eine FHEM-Fehlermeldung ausgegeben, da hier die FHEM eigene Delete-Methode benutzt wird. Somit wird auch das Logging von FHEM übernommen.

3.4.5 FntUpdateScript($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/UpdateScript?NAME=<script_name>&REGEXP=<regexp>&COM=<commands>

CodeSnippet 72: Aufruf UpdateScript()

Diese Methode ruft intern FntDeleteScript($$) (siehe 4.2.7.19) und anschließend „FntAddScript($$)“ (siehe 4.2.7.18) auf. Dadurch wird das alte Script erst gelöscht und anschließend neu erzeugt. Fehlerbehandlungen werden komplett von den

beiden Methoden übernommen auch die Parameter müssen dessen Anforderungen entsprechen.

3.5 Sonstige Funktionen

3.5.1 FntGetVersion($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/GetVersion

CodeSnippet 73: Aufruf GetVersion()

Liefert die aktuelle Version. Die erste Ziffer entspricht der Hauptversion, die zweite repräsentiert die Nebenversion und die letzte gibt Aufschluss über die Fehlerkorrektur.

3.5.2 FntSave($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/Save

CodeSnippet 73: Aufruf Save()

Diese Methode muss aufgerufen werden, wenn die gemachten Veränderungen am FHEM-System gespeichert werden sollen. Sie hat den selben Effekt, wie der Aufruf „save“ in FHEM, da diese Methode vom Modul aufgerufen wird. Wenn der Vorgang erfolgreich abgeschlossen werden konnte, wird „true“ zurückgegeben. Geloggt wird dies im Level 5. Sollte etwas schief gehen, wird die FHEM eigene Meldung als Fehler zurückgeben. Dies wird im LogLevel 4 abgelegt.

JSON-Response:
{"MESSAGE":"true","TYPE":"Response"}

CodeSnippet 74: JSON-Response bei erfolgreichem Speichervorgang

3.5.3 FntGetFhemFile($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/GetFhemFile?NAME=<File>

CodeSnippet 75: Aufruf GetFhemFile()

Mit dieser Methode können Files von FHEM vom Server heruntergeladen/angezeigt werden. Es ist möglich die „fhem.cfg“, oder alle Dateien, welche in dem /FHEM im „modpath“ (Bei Linux standardmäßig „/usr/share/fhem/“) abzurufen. Die Dateien werden im Format „plain/text“ übertragen. Sollte eine Datei nicht existieren wird eine entsprechende Fehlermeldung geliefert und im Level 4 geloggt.

JSON-Error-Response:
{"MESSAGE":"Can't open /usr/share/fhem/FHEM/sfdsf","TYPE":"Error"}

CodeSnippet 76: Fehlermeldung GetFhemFile

3.5.4 FntSetFhemCfg($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/SetFhemCfg?DATA=<Daten>

CodeSnippet 77: Aufruf SetFhemCfg()

Mit dieser Methode kann die Konfigurationsdatei von FHEM aktualisiert, bzw. verändert werden.

Achtung:

Sie kann ausschließlich mit HTTP-POST verwendet werden. Wenn sie mit GET aufgerufen wird, werden alle Sonderzeichen durch die URLencode ersetzt. (Bsp. Leerzeichen durch %20)

Es wird genau ein Parameter erwartet, nämlich die einzutragenden Daten als String mit dem Bezeichner „DATA“.

Es wird versucht die globale Konfigurationsdatei zu öffnen, welche im Attribut „configfile“ des Objektes „global“ abgelegt ist. Sollte dies nicht möglich sein, wird auch hier eine Fehlermeldung zurückgegeben und im Level 2 geloggt. Danach wird die Datei komplett mit dem übergebenen String überschrieben. Dies bedeutet, dass die vorherige Konfigurationsdatei nicht mehr vorhanden ist.

Achtung:

Es gibt keinerlei Überprüfung, ob der Inhalt wirklich eine FHEM Konfigurationsdatei darstellt. Weiterhin gibt es keinerlei Abfrage, ob die Aktion wirklich ausgeführt werden soll. Daher muss diese Funktion **unbedingt** mit Bedacht genutzt werden.

3.5.5 FntGetHelp($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/help

CodeSnippet 78: Aufruf help()

Listet eine Übersicht aller vorhandenen Funktionen. Zusätzlich werden die möglichen Parameter aufgelistet. Dies soll eine komfortablere Bedienung ermöglichen.

3.5.6 FntUpdateFhem($$)

Aufruf:

HTTP-Anfrage:
Http://<Server-IP>:<Port>/UpdateFhem?[OPT=backup Filename]

CodeSnippet 79: Aufruf help()

Diese Methode ruft im Wesentlichen die FHEM-Methode „UpdateFHEM“ auf. Es kann optional ein Parameter „OPT“ angegeben werden. (Genaue Funktionsweise der Methode bitte aus der FHEM-Dokumentation entnehmen.)

4 Hinweise zur Verwendung

4.1 GET / POST Aufrufe der Funktionen

Es ist grundsätzlich möglich alle Funktionen mit beiden Varianten zu benutzen. Das bedeutet, die benötigten Parameter können sowohl mit GET, als auch mit POST übermittelt werden. Dies Resultiert vorwiegend aus der aufgetretenen Problematik aus Punkt 4.4.3. Konkret wird dies durch die Benutzung von Bezeichnern hervorgerufen, welche durch URL-Encoding vorgeschrieben werden und auch in JSON genutzt werden. Darüber ist es möglich die darin enthaltenen Werte eindeutig zuzuweisen. Es ist aber zu beachten, dass per GET **keine Sonderzeichen** unterstützt werden. So ist es zum Beispiel nicht Sinnvoll ein Script mittels „AddScript()“ zu erstellen, da z.B. alle Leerzeichen durch „%20“ ersetzt würden. Mittels POST und JSON ist dies aber durchaus möglich! Die JSON-Syntax und der JSON-Parser erlauben es innerhalb der Daten Sonderzeichen anzugeben. Aber auch hier ist zu beachten, dass nicht alle Sonderzeichen seitens FHEM erlaubt sind. Es sind z.B. keine Umlaute innerhalb von Namen erlaubt.

Achtung:

Es ist unbedingt zu beachten, dass die Bezeichner der Parameter Casesensitiv sind, es muss also unbedingt die geforderte Groß-/Kleinschreibung eingehalten werden!

4.2 Define von TheOpenTransporter in FHEM

An dieser Stelle wird nun beschrieben, wie „TheOpenTransporter“ in FHEM definiert werden kann. Es gibt viele Möglichkeiten, das Modul zu definieren.

Zunächst einmal die prinzipielle Syntax:

FHEM-Befehl:
define <name> TheOpenTransporter <Port> [Optional: IPV6] [Optional: HTTPS] [Optional: SSL_KEY </path/server-key.pem> SSL_CERT </path/server-cert.pem>]

CodeSnippet 80: FHEM-Define-Syntax TOT

Die einfachste Art und Weise ist es nur einen Port anzugeben.

FHEM-Befehl:
define JasonStatham TheOpenTransporter 1234

CodeSnippet 81: FHEM-Define TOT

Hiermit wird ein FHEM-Modul vom Typ TheOpenTransporter definiert, welches auf dem Port 1234 auf Anfragen wartet. Sollte ein Port angewählt werden, der schon vergeben ist, wird nichts definiert und es wird eine entsprechende Fehlermeldung geliefert.

FHEM-Befehl:

```
define JasonStatham TheOpenTransporter 1234 HTTPS
```

CodeSnippet 82: FHEM-Define TOT mit HTTPS

Erzeugt genau das selbe Modul, wie in dem Beispiel davor. Hier ist allerdings die Besonderheit, dass eine gesicherte Verbindung mit SSL aufgebaut wird. Es ist zu beachten, dass per Default der Standard Key und Zertifikat des Hosts verwendet wird, sofern diese gefunden werden können. Es ist aber auch möglich mit den Parametern „SSL_CERT“ und „SSL_KEY“ einen selbstgewählten Speicherort anzugeben.

FHEM-Befehl:

```
define JasonStatham TheOpenTransporter 1234 HTTPS SSL_CERT /Beispiel/Pfad/Zertifikat.pem SSL_KEY /Beispiel/Pfad/Key.pem
```

CodeSnippet 83: FHEM-Define TOT mit HTTPS u. eigenem Zertifikat & Key

Sollten kein Key und kein Zertifikat gefunden werden können, wird TOT ohne HTTPS-Unterstützung erstellt. In diesem Falle können die Pfade nachträglich abgeändert und die Unterstützung erneut angeschaltet werden, in dem das Attribut HTTPS auf 1 gesetzt wird.

Als letztes besteht die Möglichkeit TOT unter der Verwendung von IPv6 zu erstellen. Dafür muss lediglich das Parameter „IPV6“ übergeben werden.

FHEM-Befehl:

```
define JasonStatham TheOpenTransporter 1234 IPV6
```

CodeSnippet 84: FHEM-Define TOT mit HTTPS u. eigenem Zertifikat & Key

Natürlich ist es möglich alle optionalen Parameter in allen Varianten miteinander zu Kombinieren. Genauso ist die Reihenfolge der optionalen Parameter irrelevant – lediglich SSL_KEY und SSL_CERT müssen mit ihren Werten als Paare hintereinander angegeben werden.

4.3 Abhängigkeiten von TheOpenTransporter

Um das TheOpenTransporter benutzen zu können, muss sichergestellt werden, dass folgende vorausgesetzte Komponenten verfügbar sind.

- IO::Socket
- IO::File
- JSON
- IO::Socket::SSL (optional)
- IO::Socket::INET6 (optional)
- FHEM in der Version 5.2 oder höher

IO::Socket (https://metacpan.org/module/IO::Socket) ist eine Perl-Komponente, welche die Netzwerkkommunikation ermöglicht. Sie wird ebenfalls auch von FHEM (z.B. „fhem.pl“) genutzt und sollte damit auch vorhanden sein.

IO::File (https://metacpan.org/module/IO::File) ist eine Perl-Komponente, welche den Zugriff auf Dateien ermöglicht. Auch diese wird von FHEM selbst genutzt und sollte daher vorhanden sein (in der FileLog.pm).

IO::Socket::SSL (https://metacpan.org/module/IO::Socket::SSL) ist wie IO::Socket für die Netzwerkkommunikation zuständig, allerdings verschlüsselt. Auch diese Komponente wird bereits von FHEM genutzt und sollte somit verfügbar sein.

JSON (https://metacpan.org/module/JSON) ist eine Komponente, welche es ermöglicht Datenstrukturen in ein JSON-Objekt umzuwandeln. Hier ist zu beachten, dass diese Komponente nicht Standardmäßig von FHEM verwendet wird. Daher muss die Verfügbarkeit ggf. überprüft werden.

IO::Socket::INET6
(http://search.cpan.org/~shlomif/IO-Socket-INET6-2.69/lib/IO/Socket/INET6.pm)
Analog zu IO::Socket, allerdings für IPv6.

Dieses lässt erkennen, dass TheOpenTransporter sehr wenige Abhängigkeiten hat und somit extrem gut für den Einsatz auf „embedded Systems“ ist.

5 Fazit und Ausblick

TheOpenTransporter konnte erfolgreich fertig gestellt werden und erfüllt alle, für das ursprüngliche Projekt, vorgesehenen Anforderungen.
Es ist möglich alle Daten aus FHEM auszulesen, zu modifizieren oder zu erstellen. Somit kann FHEM in seinem vollen Umfang über das Modul genutzt werden. Es wurde dabei stets darauf geachtet die vorhandenen Perl-Schnittstellen von FHEM zu nutzen, um die FHEM-interne Logik nicht neu zu erfinden. Damit wurde das Risiko, Fehler einzubauen, minimiert.

Zudem wurden Funktionalitäten hinzugefügt, welche ursprünglich nicht vorgesehen waren: In der Hinsicht auf Sicherheit (HTTPS) und Authentizität und Datenkomplexität (JSON->Sonderzeichen) stellt TheOpenTransporter somit eine bislang nicht erreichte Möglichkeit für eine externe Kommunikation.

Generell wurde stets darauf geachtet das Modul sehr ausführlich zu kommentieren, damit die Community bei Bedarf Änderungen/ Weiterentwicklungen tätigen kann.
Die Autoren sind Studenten und haben daher nur eingeschränkt Zeit das Modul weiterzuentwickeln, sie haben aber gerne daran gearbeitet und versuchen, sofern es die Zeit zulässt, die ein oder andere Änderung/Verbesserung vorzunehmen.

An dieser Stelle möchten wir uns noch bei der Hochschule Harz und speziell bei Herrn Prof. Dr. F. Stolzenburg und Prof. Dr. O. Drögehorn, für die Möglichkeit zur Veröffentlichung, bedanken.