1 Kommunikationsprotokoll

Das **Inter Board Protocoll (IBP)** dient als Vereinbarung von Kommunikationsregeln Ã¼ber eine serielle Kommunikationsschnittstelle. Innerhalb dieses Projektes soll es die Kommunikation Ã¼ber eine serielle USB Schnittstelle erleichtern.

HauptsÃchlich benutzte Programmiersprachen war C++. C code wurde wenn vorhanden auf C++ geportet, in dem Funktionen und Daten in Klassen gekapselt wurden. FÃ¼r zusÃtzliche Tools wurde Python verwendet. (funktionieren mit Python2 und Python3)

Das Protokoll wurde zum Teil iterativ Entwickelt. Das soll heiÃŸen es wurde wiederholt ein funktionsfÃhiges Produkt mit Features erstellt und integriert. Beim Auftreten neuer oder verÃnderter Anforderungen oder von Fehlern wurde die Komponente Kommunikationsprotokoll aber auch teils komplett neu implementiert. Das bedeutet aber auch, dass bestimmte Features, die in einer frÃ¼heren Version vorhanden waren, in eine spÃteren Version jedoch vernachlÃssigt oder nicht mehr reimplementiert wurden. Im folgenden werden daher mehrere Versionen, ihre Probleme und die Reaktion auf diese Probleme aufgefÃ¼hrt, erklÃrt und begrÃ¼ndet.

Entwickelt wurden Komponenten auf beiden Benutzten Plattformen, Raspberry Pi 3 und Arduino Uno.

VersionsÃ¼bersicht :

Im folgenden ist eine Ãœbersicht Ã¼ber die durchlaufenen Versionen wÃhrend der Entwicklung zu Orientierung bereitgestellt. Dabei sind auch zeitlich Beginn und Ende des Entwicklunszeitraumes angegeben.

VNr.	Wesentliche Ã„nderungen	Begin	Ende
0.0	Erste Versuche		
0.1	Kontrollstrukturen und Kapselung des Prozesses		

1.0.1 Version [V0.0]

Anforderungen erfÃ¼llt : [A01][A02]

Diese Version is als erster minimalistischer Versuch gedacht. Sie wurde erstellt um erste Anforderungen zu testen und um eventuell Ã¼bersehene Basisanforderungen zu finden. Die Arbeiten an dieser Version waren hauptsÃchlich auf dem Raspberry Pi angesiedelt. Die FunktionalitÃt des Slaves wurde dabei zunÃchst simuliert. Daduch, dass die Benutzung der Seriellen Schnittstelle in einer Komponente bereits funktioniert hat, konnte ohne tatsÃchliche Integrationstests ausgekommen werden. Des weiteren wurde zunÃchst ein simpler Protokollablauf verwendet, wie folgt:

In Version 0 alle benÃ¶tigten Aufrufe wurden zunÃchst testmÃÃŸig hart codiert. Ein Unit-Test bestÃtigte die ordnungsgemÃÃŸe FunktionalitÃt. Keine weiteren Anforderungen kamen zum Vorschein.

1.0.2 Version [V0.1]

Anforderungen erfÃ¼llt :

In diese Version sollten mehr Ãœberlegungen einflieÃŸen, kontollierende Strukturen sollen erstellt werden und der Prozess der Protokollbenutzung gekapselt. Eine automatisiertere, benutzbare Softwarekomponente wird hier zunÃ¤chst fÃ¼r die Masterplattform Raspberry Pi angestrebt. Die Implementierung auf der Slaveplattform Arduino Uno wurde zunÃ¤chst vernachlÃ¤ssigt.

GemÃ¤Ã¿ [A08.1] wurden zunÃ¤chst die Eigenschaften des Raspberry Pi eingeschÃ¤tzt. Der Raspberry Pi nimmt dabei die Rolle des Masters bei der Kommunikation ein. Folgende Eigenschaften waren dabei besonders interessant :

- Raspberry Pi 3 wurde mit Betriebssystem ausgestattet (Rasbian) → Raspberry ermÃ¶glicht Multitasking und -threading
- FÃ¼r einen Controller besitzt der Raspberry vergebend viel Arbeitsspeicher (1GB) =¿ Das anlegen eines Buffers fÃ¼r Nachrichten ist nicht kritisch.
- Das Raspbian Betriebssystem hat ein Filesystem → Konfigurationsdateien direkt auf der Plattform sind mÃ¶glich.

Daher Modellidee :

- Ein dedizierter Thread ist fÃ¼r das Senden und Empfangen, bzw. den Ablauf des Protokolls, zustÃ¤ndig.
- Benutzer geben ihre Befehle an den Slave als Pakete dem Thread. Die zugehÃ¶rige Softwarekomponente wird ¿Packet¡ genannt.
- Eine Softwarekomponente kapselt die FunktionalitÃ¤t des Threads, diese wird ¿Transceiver¡ genannt.
- Ein Benutzer kann mittels einer weiteren Softwarekomponente, ¿Inbox¡, auf Antworten warten.
- Informationen Ã¼ber die LÃ¤nge der Payload eines Request oder einer Response kann Ã¼ber eine Configurationsdatei bewerkstelligt werden. Das einlesen der Datei und die Bereitstellung der Information wird durch die Komponente ¿Rule¡ verwaltet.

Dadurch kÃ¶nnen Anforderungen bedient werden :

A01 Die Frage wird den Thread Ã¼bergeben, die Antwort landet in der ¿Inbox¡.

A02 ¿Packet¡en wird eine Nachrichtenart, ID, zugeschrieben.

A03 Der Thread kann selbst entscheiden in welcher Reihenfolge angekommene ¿Packet¡e verschickt werden. Damit ist Priorisierung umsetzbar.

A04 ¿Packet¡e fassen ein Maximum an Ãœbertragungsdaten ein.

A05.1 Eine Ã¼ber ¿Rule¡ eingelesene Konfigurationsdatei enthÃlt Informationen Ã¼ber die GrÃ¶ÃŸe der Payload fÃ¼r eine Nachrichtenart (ID).

A08 Raspberry Eigenschaften wurden ausgenutzt.

und erzeugt weitere Annehmlichkeiten :

- Benutzer kÃ¶nnen in ihren eigenen Threads arbeiten, d.h. verschiedenen Komponenten kÃ¶nnen potentiell Zugang
- TatsÃchliche Kommunikation lÃuft zentral (nicht jeder Benutzer baut eine eigene LÃ¶sung) → Kommunikation leichter zu verwalten.

Es entstehen dadurch auch neue Anforderungen:

F09 : Die Benutzung der Prokollsoftware muss auf der Zielplattform Pi von allen laufenden Threads der Software aus potentiell machbar sein.

F09.1 : Die Protokollsoftware auf der Zielplattform Pi ist thread-safe implementiert. (Vermeidung von Race-Conditions)

Implementierung :

Die Implementierung sieht fÃ¼r jede der besprochenen Komponenten eine Klasse vor. Die Klassen wurden zum grÃ¶ÃŸten Teil in der selben Reihenfolge implementiert, wie im Folgenden dargestellt.

Als Protokollablauf wurde der triviale aus V0.0 verwendet, um zu testen.

1.0.3 Version [V0.2]

Anforderungen erfÃ¼llt : [A01] [A02] [A03] [A04] [A05.1] [A05.2] [A06] [A08.1] [A09.1]

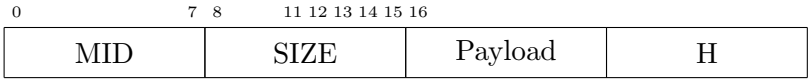
FÃ¼r diese Version wurde sich dem Protokollablauf angenommen.

[A01] wurde der Ablauf als recht unkompliziert Frage-Antwort-Schematisch angenommen, in dem in einem StÃ¼ck Daten hin und zurÃ¼ck gesendet werden.

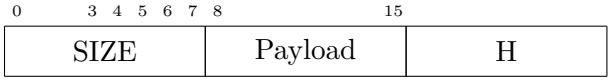
Legende:

MID	Identifikationsnummer der Anfrage
EID	Nummer des Fehlers bzgl. MID
SIZE	GrÃ¶ÃŸe einer dynamischen Payload, nicht existent bei statisch vereinbarter Ãœbertragung!
H	Hash
Payload	Nutzdaten

Request



Response



[A05.2] Optional kann ein Feld fÃ¼r die LÃnge der Payload mitzusenden um dynamische Frames zu ermÃ¶glichen. Die optionale Routine wÃ¼rde ausgelÃ¶st werden, wenn in der Konfigurationsdatei (Ã¼ber ¿Rule¡ eingelesen) ein bestimmter, zu groÃŸer Wert (255) als statische GrÃ¶ÃŸe stÃ¼nde $\rightarrow$ *size* $= 255 \Rightarrow$ *dynamisch*. Ein Benutzer kann also 255 als GrÃ¶ÃŸe definieren und ein ¿Packet¡ mit beliebiger GrÃ¶ÃŸe im Rahmen der MaximalgrÃ¶ÃŸe ([A04]) versenden.

[A06] Eine Checksumme macht es dem Kommunikationspartner mÃ¶glich die Korrektheit der Ã¼bertragenen Daten festzustellen. Als Checksumme/Hash wurden dabei meistens alle gesendeten Bits effizient in einer bestimmten Schrittweite exklusiv verodert.

Hashing / Bildung von Checksummen Verwendete Hashfunktionen wurden selbst erstell, jedoch nach einem Schema: Der gesuchte Hash der BitlÃnge L_h ist praktisch immer als Potenz von 2 gewÃhlt : $\{L_h | i \in : L_h = 2^i\}$ (meist 2 , 4 oder gar 8)

und die zu hashenden Daten der BitlÃnge L_d sind immer ein Vielfaches von 2 : $\{L_d | i \in : L_d = 2i\}$ (tatsÃchlich meist von 8 da Daten meist byteweise behandelt werden)

Nun kann die LÃnge L_d in Schritten der LÃnge L_h durchlaufen und exklusiv verodert werden. Dabei kann ein Rest $L_d mod L_h \neq 0$ Ã¼brig bleiben. Dieser ist ein vielfaches von 2 aber kleiner L_h und kann deshalb mehrmals nebeneinander geschrieben werdeni bis er die BitlÃnge L_h wieder erreicht, um im Letzten Schritt exklusiv verodert zu werden.

Beispiel 1:

Wir Hashen 20 bit in einen 8bit hash. ZunÃchst werden desshalb in 8bit-Schritten die Daten exklusiv verodert, bis der Rest nicht mehr reicht.
$L_d = 20bit, L_h = 8bit, d = 01110011000011111010$ Schritt 1: $XOR(01110011, 00001111) = 01111100$ Schritt 2: Der Rest (1010) reicht nicht aus, desshalb schreiben wir ihn mehrmals hintereinander auf und exclusiv verodern ihn daraufhin : $XOR(01111100$(voriges Ergebnis)$, 10101010$(Rest 2 mal nebeneinander)$) = 11010110$

Beispiel 2:

$$L_d = 26bit; d = 11101001111100001010111110$$
$$L_h = 4bit$$
$$\text{Schritt } 1 : h = XOR(XOR(XOR(XOR(XOR(1110, 1001), 1111), 0000), 1010), 1111) = 1101$$
$$\text{Schritt } 2 : XOR(1101, 1010) = 0111$$
$$\rightarrow \text{h} = 0111;$$

1.0.4 Version 0.3

In dieser Version wurde sich der Entwicklung von Komponenten auf der Slaveplattform Arduino Uno angenommen.

[A08.2] muss hierbei einbezogen werden. Dabei waren folgende Eigenschaften der Plattform interessant :

- Arduino Uno fÃ¼hrt nur sequenziell Befehle aus, kein paraleller Ablauf von Code ist mÃ¶glich. Bestimmte Aufgaben kÃ¶nnen der Hardware Ã¼bergeben werde, was jedoch fÃ¼r das Protokoll uninteressant ist.
- Arduino Uno besitzt verhÃltnismÃÃŸig wenig Arbeitsspeicher (2Mb). Wenn man nun die unter [A04] angesprochene MaximalgrÃ¶ÃŸe einer Datensendung be-

denkt (255 bytes), wÃ¼rde das bereits $/frac18$ des Speichers einnehmen. Das ist inaktzeptabel.

Deshalb Modellidee :

- Eine Komponente/Klasse ¿IBC¡ auf dem Arduino Uno stellt alle FunktionalitÃten bereit.
- Benutzer machen alle Aufrufe zum Senden und Empfangen des variablen Teils der Daten, welche gesendet werden, selbst. Dabei fÃ¼gen sie berechnenden Code direkt hinzu.
- Der Ã¼brige Teil des Protokollablaufs wird den Benutzern durch Codegeneration abgenommen.
- Die Codegeneration hilft den Benutzern durch viel Kommentierung und Beispielcode.

Dadurch kÃ¶nnen alle nÃ¶tigen Befehle sequenziell in den Programmablauf direkt eingebettet werden. Das Speicherproblem wird auf den Stack verlegt, d.h. es findet kein Buffering von Nachrichten statt. Stattdessen kÃ¶nnen Daten direkt aus dem Bereits benutzten Speicher des Benutzers gesendet oder in diesen geschrieben werden.

Codegenerator Der Codegenerator besitzt folgende Anforderungen:

- Code soll nur in einer Datei generiert werden mÃ¼ssen.
- Generierungsparameter sind die Daten aus der selben Konfigurationsdatei wie fÃ¼r die masterseitige Komponente.
- Generator kann wiederholt auf derselben Datei ausgefÃ¼hrt werden.
- Generator preserviert von Benutzern geschriebenen Programmcode.

Der Codegenerator wurde in Python erstellt.

- 1 Mittels durch Code-Kommentare realisierter Tags sucht der Generator die Stelle zum generieren.

```
/* IBC_FRAME_GENERATION_TAG_BEGIN */
/* IBC_FRAME_GENERATION_TAG_END */
```

2 Er sucht daraufhin alle zu erhaltenen Stellen im Ursprungsdokument.

```
/* IBC_PRESERVE_RECV_BEGIN 252 vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv

/* IBC_PRESERVE_RECV_END 252 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
/* IBC_PRESERVE_SEND_BEGIN 252 vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv

/* IBC_PRESERVE_SEND_END 252 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

3 Ãœber die Konfigurationsdatei werden nun fÃ¼r jede konfigurierte Nachrichtenart entsprechende Code-Fragmente erstellt. Dabei werden Eingabeparameter aus der Konfigurationsdatei berÃ¼cksichtigt.

```
/* IBC_MESSAGE_BEGIN 253 255 2 */

/* IBC_MESSAGE_END 253 255 2 */
```

3.1 Dabei wird der zu kapselnde Ablaufcode auÃŸerhalb der Preservierung auf die neueste Version gebracht.

3.2 Falls unter 2tens zu erhaltener Code gefunden wurde wird dieser wieder eingefÃ¼gt und dadurch erhalten.

3.3 Falls unter 2tens kein Code gefunden wurde wird Beispielcode generiert unter einbezug der Konfiguration generiert und eingefÃ¼gt.

```
/* IBC_PRESERVE_RECV_BEGIN 252 vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv

byte buffr252[4];
recv(buffr252,4);

//DONT FORGET TO HASH
setDH(createDH(buffr252,4));

/* IBC_PRESERVE_RECV_END 252 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
/* IBC_PRESERVE_SEND_BEGIN 252 vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv

byte buffs252 [8] = {1,2,3,4,5,6,7,8};
for(int i = 0 ; i<8;i++) {send(0);}

//DONT FORGET TO HASH
setDH(createDH(buffs252, 8));

/* IBC_PRESERVE_SEND_END 252 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

3.4 Unter Beachtung der Konfiguration werden erklÃ¤rende und helfende Kommentare generiert.

```
/*Send exactly 8 bytes in the following                    */
/*If you have a dynamic size you have to send this size first
/*Also calculate their data hash along the way by
/*  xoring all bytes together once
/*  or use the provided function createDH(..)
/* Make the hash public to the IBC by setDH(Your DATAHASH HER
/* IBC_PRESERVE_SEND_BEGIN 252 vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
```

```
            byte buffs252 [8] = {1,2,3,4,5,6,7,8};
for(int i = 0 ; i<8;i++) {send(0);}

//DONT FORGET TO HASH
setDH(createDH(buffs252, 8));

/* IBC_PRESERVE_SEND_END 252 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

4 Der Generator schlÃgt fehl wenn die Ursprungsdatei und die Konfiguration divergieren. Das ist gewÃ$\frac{1}{4}$nschtes verhalten. Nicht konfigurierte Nachrichten sollten daraufhin aus dem Code entfernt und der Generator erneut benutzt werden.

Der Benutzer muss sich beim Entwickeln seiner Nachricht nun nur an folgende Regeln halten :

- Benutzercode wird nur in den angegebenen PRESERVE Tags eingefÃ$\frac{1}{4}$gt
- Die in der Konfigurationsdatei angegebene Anzahl an Bytes muss gesendet, bzw. empfangen werden. Generierte Kommentierung weist ausdrÃ$\frac{1}{4}$cklich darauf hin. Auch mÃ$\frac{1}{4}$ssen die Daten gehasht werden und ein Hashwert gesetzt. Dieser wird dann benutzt um den letztendlichen Hash zu erstellen, der Ã$\frac{1}{4}$ber die Leitung geht.

Output Hilfe :

```
pi@raspberry  ~/DT_WS1718_02_StarCar/uno $ ./Uno_ibcgeneratio

Help for this code generator.
This generator generates code in a correctly tagged code file
```

```
Arguments : [Rule] [Output]

    Rule    Is a file which specifies an IBC ruleset in form
    Output  Is a file where the output will go. It will be se

                /* IBC_FRAME_GENERATION_TAG_BEGIN */

                /* IBC_FRAME_GENERATION_TAG_END */

               this script will not touch any lines outside thes
               This script will also preserve custom code inside

                /* IBC BEGIN [MID] [RECV/SEND]

                /* IBC END [MID] [RECV/SEND]
```

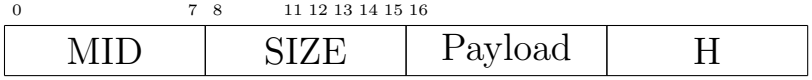
Benutzung :
Rule zielt auf die Konfigurationsdatei ab. Output auf
das File in dem generiert wird.

```
pi@raspberry  ~/DT_WS1718_02_StarCar/uno $ ./Uno_ibcgeneratio
```

Nachdem diese Version erstellt wurde wurde auf den
Zielplattformen standalone integriert und getestet,
d.h. keine anderen Komponenten auÃŸer die
Kommunikation liefen zu der Zeit auf den
Zielplattformen.

Version [V0.4]

Probleme : Der in [V0.2] entwickelte Ablauf weist
konzeptionelle Fehler auf.



Bei einer fehlerhaften Ãœbertragung von MID oder SIZE kann die LÃnge der Payload nicht ermittelt werden. Das bedeutet das der EmpfÃnger keine MÃ¶glichkeit hat zu erfahren wo sich das Feld fÃ¼r den Hash in der Ãœbertragun Ã¼berhaupt befindet. Schlimmer : Der EmpfÃnger kann den Beginn eines neuen Requests nicht mehr ermitteln und die Kommunikation wird bei folgenden Ãœbertragungen immer fehlerhaft. Potentiell kÃ¶nnte Ã¼ber den Hash die fehlerhafte Ãœbertragun erkannt werden, aber da der EmpfÃnger nicht ermitteln kann, wo sich dieser befindet droht Desynchronisation.

Ein dedizierter Header mit einem eigenen Hash und statischer GrÃ¶ÃŸe kann dieses Problem lÃ¶sen. Idealerweise EnthÃlt der Header die Felder *MID, SIZE* und *H*. Da das SIZE Feld jedoch optional ist und bei statischen Nachrichten nicht mitgeschickt werden soll wÃ¼rde der Header nicht mehr statische, sondern dynamische GrÃ¶ÃŸe haben. Deshalb wird der Header an dieser Stelle getrennt und jeder Teil selbst gehasht.

ZusÃtzlich wurde in dieser Version noch StatusÃ¼bertragung entwickelt. Das Statusfeld wird verwendet um den Status der Kommunikationspartner zu kommunizieren. Die Ã¼bliche DatenÃ¼bertragung kann dabei nebenher unbeeintrÃchtigt weiterlaufen. Das Statusfeld ist 4bit lang =¿ 16 verschiedene Stati mÃ¶glich. Die Bedeutung des Status kann je nach Art des Kommunikationspartners (Master oder Slave) variieren, d.h. der Status 8 beduetet beispielsweise auf dem Master etwas anderes als auf dem Slave. Das

Statusfeld hat im Kontrollfluss die wichtige Aufgabe, dem Master eine MÃ¶glichkeit zu geben, zwischen positiver oder negativer Antwort zu unterscheiden.

Masterseitig :

Bit 0 wird als STOP Befehl benutzt. Sein Versand bedeutet das Ende der Kommunikation. Bit 1 wird als REINIT Befehlt benutz. Sein Versand fordert eine Reinitialisierung der Kommunikation auf Slave-Seite. Bit 2 und 3 wird verwendet um eine erneute Sendung mit einer Nummer 0-3 zu kennzeichnen. Dies geschieht wenn eine Sendung im Vorhinein fehlgeschlagen hat. Eine Sendung kann bis zu 3 Mal wiederholt werden.

bit	Zweck
0	STOP
1	REINIT
2,3	Resend counter

Slaveseitig:

Bit 0 wird verwendet um einen internen Fehler am Slave zu propagieren, der zur Folge hat das dieser nicht korrekt oder gar nicht auf Befehle reagiert. Der Master kann dadurch auf den Zustand reagieren um zum Beispiel den Fehler zu loggen, den Sendevorgang fÃ$\frac{1}{4}$r eine bestimmte Zeit einzustellen, Systeme kontrolliert herunterfahren zu lassen, etc.

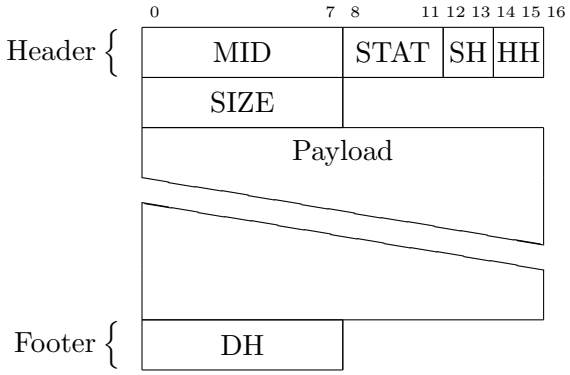
Bit 1, 2 und 3 werden zum Kommunizieren von Fehlern der 3 Checksummen des Protokolls wie folgt verwendet.

bit	Zweck
0	interner Slave Fehler
1	Headerhash falsch
2	Sizehash falsch
3	Datahash falsch

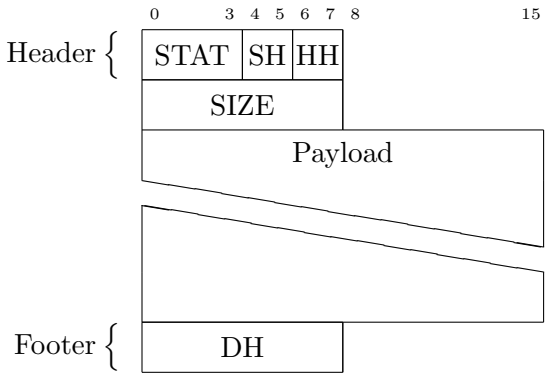
Legende:

MID	Identifikationsnummer der Anfrage
EID	Nummer des Fehlers bzgl. MID
SIZE	GrÃ¶ÃŸe einer dynamischen Payload, nicht existent bei statisch vereinbarter Ãœbertragun
STAT	Status der Ãœbertragung;Protokollinterne Fehlererkennung
HH	Hash des Headers
SH	Hash der dynamischen GrÃ¶ÃŸe (unwichtig bei statisch)
DH	Hash der Payload
Payload	Nutzdaten

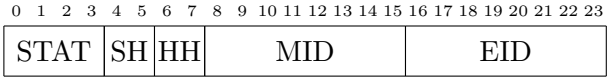
Request



Response



Negative Response



Auf dem Raspberry Pi wurde zu den Ã¼brigen Komponenten ein Fassade-Muster Implementiert. Das ist eine Komponente, welche den Benutzern eine klare Benutzungsschnittstelle liefert und dabei FunktionalitÃt, welche nicht angedacht ist vom Benutzer verwendet zu werden, versteckt. Die Komponente wurde ¿IBC¡ genannt.

1.0.5 Version [V1.0]

Diese erste volle Version wurde zum ersten mal in die Ã¼brige Projektumebung integriert. Alles schien zu funktionieren, Tests waren positiv, die Kollegen/Kolleginnen, welche die Komponenete in der Zukunft benutzen wÃ¼rden mÃ¼ssen, wurden in der Benutzung geschult.

Problem : Nach einiger Zeit traten Speicherzugriffsfehler (Segmentation Faults) auf. Die Fehler sind in ihrer Natur undefiniertes Verhalten, welches letztendlich zu einem Fehler fÃ¼hrt. Ein Backtrace mittels GNU-Debugger zeigte den Programmablauf zeigte die Aufrufhierarchie vor dem Fehler an. Der Ablauf zeigte den Aufruf von protokollspezifischem Code an, welcher Aufrufe der Standardbibliothek machte um darauf bei einem Zugriff abzustÃ¼rzen. Die Schuld am Absturz wurde also dem Protokollcode selbst gegeben.

1.0.6 Version [V1.1]

Anforderungen erfÃ¼llt : [A01][A02][A03][A04][A05.1][A0][A0][A0]

GrundsÃtzlich waren Versuche den Fehler zu beheben recht ratlos. Nach erheblichem Zeitaufwand bei der Fehlersuche wurde entschieden zu versuchen die Fehler zu vermeiden. So wurde ein GroÃŸteil des Codes Refactored um Aufrufe der C++ Standardbibliothek zu vermeiden. Dabei wurden viele bereits eingebaute Features nicht mehr neu Implementiert, da zu dem Zeitpunkt bereits klar war, dass sie nicht gebraucht werden wÃ¼rden. Zeit war dabei ein Thema, denn die Deadline zur ersten Ergebnisvorstellung stand bevor (1 Woche) Betroffen sind Features, welche aus den Anforderungen [A05.2] [A07.2] [A07.3] hervorgehen.

Verlorene Features:

A05.2 Die Bearbeitung optionaler PayloadlÃngen wurde bei der Reimplementierung des Protokollablaufs nicht wieder eingebaut, da sie offenbar innerhalb des Projektes nicht benutzt werden wÃ¼rde und die Reimplementierung erneut Zeit gekostet hÃtte.

A07.2 Ebenso wie [A05.2] wurde die negative Antwort nicht mit reimplementiert.

A07.3 Die generelle Behandlung empfangener Fehler wurde nicht reimplementiert. Ãœbertragungsfehler traten zu wenig hÃufig auf.

Des weiteren wurde vermutet, das die Entwicklungsplattform und -umgebung des Kollegen zur Entwicklung der GUI, welche mit Qt entwickelt wurde, nicht mit der Standardbibliothek kompatibel war. Deshalb wurden alle Aufrufe der Standardbibliothek auf die sehr Umfangreiche QT-Bibliothek umgeschrieben. ZusÃtzlich kam zu der Zeit noch ein gewisser Druck mit der Deadline zur ersten Ergebnisvorstellung hinzu. Letztendlich wurde nach ca. 1,5 Wochen durch Zufall die falsche Benutzung eines Pointers festgestellt. Der Kollege hatte ein Objekt der Klasse ¿IBC¡ auf dem Heap angelegt und den Pointer dazu Ã¼berschrieben, was zu Fehlern im Protokollcode fÃ¼hrte, obwohl dieser korrekt war.

PersÃ¶nliche EinschÃtzung: Der Zeitverlust und der Verlust bereits ausgearbeiteter Features war fÃ¼r meinen eigenen geplanten Projektablauf ein Desaster. TatsÃchlich wollte ich noch an etwas anderes als dem Protokoll arbeiten (Raumerkennung zum Beispiel, siehe Zielvereinbarungen). Nun war der GroÃŸteil der angedachten Arbeitszeit schon dafÃ¼r verbraucht und das Endprodukt genÃ¼gte nicht mehr meinen eigenen Vorstellungen, da nun Features fehlten, deren Reimplementierung oder Reintegrierung als zu AufwÃndig, bzw. nicht mehr zweckgemÃÃŸ angesehen wurden (einige Features wÃ¼rden offenbar von den letztendlichen Benutzern gar nicht gebraucht werden, wie zum Beispiel [A05.2]). Dadurch das das Protokoll so von zentraler Wichtigkeit fÃ¼r einige Kollegen war, die Daten zwischen den Controllern versenden wollten, musste das Protokoll weiter Priorisiert entwickelt werden. Im Nachhinein betrachtet war auch das weiterarbeiten auf [V1.1] ein Fehler. Stattdessen hÃtte auf [V0.4] zurÃ¼ckgerudert werden mÃ¼ssen, um auch die nicht gebrauchten Features zu behalten. Das wurde jedoch unterlassen um die Ãœbersicht nach dem Fehlerchaos zu behalten (weniger aktive Features bedeutet weniger Programmiercode, der bei der Bearbeitung behindert).

1.0.7 Version [V1.2]

Diese nun durch den Fehler sehr runtergekommene Version[1.1] sollte nun etwas verbessert werden um wieder den wichtigsten Anforderungen zu genÃ¼gen. Auch wurde gehofft einige verlorene Features im Nachhinein wieder einbauen zu kÃ¶nnen.

Der Protokollablauf aus [V0.4] ist immernoch ungenÃ¼gend. Einfaches Beispiel: Wir nehmen an das MID falsch Ãœbertragen wird. Der EmpfÃnger erkennt die falsche Ãœbertragung anhand des Headerhashes. Der Protokollablauf sieht jedoch an dieser Stelle keinen Abbruch vor. Schlimmer : Der Master wird den Rest seiner Ãœbertragung senden, obwohl der Slave auf Grund der fehlerhaften Ãœbertragung wieder nicht weiÃŸ wo die Ãœbertragung endet. Bis dato wurde in diesem Fall einfach eine bestimmte Zeit gewartet, alle bis dahin empfangenen Daten verworfen und eine negative Nachricht zurÃ¼ckgesendet. Da dies jetzt keine option mehr darstellt (und Wartezeit niemals eine optimale LÃ¶sung darstellt) muss der Protokollablauf wieder geÃndert werden.

Erdacht wurde die im folgenden Sequenzdiagramm dargestellte LÃ¶sung :

Durch den Status-Handshake kann dieser Fehler direkt behandelt werden. Ein Problem stellte sich bei der genaueren Entwicklung des Status-Handshakes dar. Es ist ersichtlich das das letzte Ã¼bertragene Byte des Handshakes immer kritisch bleiben wird, im Falle, dass es falsch Ã¼bertragen wird. Ein Beispiel : Handshake :

- 1.Master sendet ID 5.
- 2.Slave sendet das die ID korrekt Empfangen worden ist. OK.
- 3. Master schlieÃŸt des Handshake ab.

Nun ist es fÃ¼r den Master unwichtig, ob Schritt 3 denn tatsÃchlich richtig empfangen worden ist. FÃ¼r den Fall der korrekten Ãœbertragung ist alles im Rahmen, jedoch im Fall der inkorrekten Ãœbertragung muss der Slave sich trotzdem auf die darauf kommende Flut an Daten einstellen. Hat der Slave in Schritt 2 nicht OK, sondern einen Fehler gemeldet, soll der Master den Ablauf neu beginnen. Dazu Signalisiert er dem Slave in Schritt 3 einen Ablaufneustart. Wieder kann der Slave in diesem Fall dieses Signal falsch erhalten und unter UmstÃnden nicht interpretieren. Die Krux der Ãœberlegungen an dieser Stelle ist vor allem, dass sie dazu verleiten im Kreis zu denken. Es gÃbe nahe an 100% sichere Methoden an dieser Stelle, die hauptsÃchlich auf rekursiven Sendeaufrufen beruhen, diese sind sind jedoch fÃ¼r den Rahmen des

Projektes nicht angemessen.(Auto ist kein Mars-Rover) Der Handshake wie beschrieben bietet eine Verbesserung der Situation, jedoch keine 100% LÃ¶sung. Das soll als angemessen angesehen werden.

Der Handshake und der neue Protokollablauf wurden implementiert, fÃ¼r die Behandlung der Fehler wurde jedoch keine Zeit gefunden.

Integation auf den Plattformen fand statt.

1.0.8 Version [V1.3] Final

Problem Werte wurden von Slave auf Master oft nicht richtig Ã¼bertragen. Untersuchungen des Problems fanden unerklÃ¤rlicherweise Sendeaufrufe von Statusbytes, die sich spÃ¤ter nicht im Ã¤bertragunslog fanden. Die in [V1.2] entwickelten Neuerungen wurden weitestgehend wieder zurÃ¼ckgebaut. Die bevorstehende VorfÃ¼hrung fÃ¼hrte zu dieser Zeitsparenden LÃ¶sung.

1.0.9 Fazit

Zwar durfte ich bei der Entwicklung es Protokolls viele auch teils aus dem Studium bekannte Problemstellungen selbst meistern und hatte viel SpaÃŸ und Lernerfolg dabei, jedoch lief der Aufwand dieses Teilprojektes stark aus dem Ruder. Ich hÃ¤tte mich gerne auch anderen Aufgaben gewidmet, aber ein nicht funktionsfÃ¤higes Kommunikationsprotokoll bedeutet auch das scheitern der Teilprojekte die von funktionierender Kommunikation abhÃ¤ngig sind, d.h. von Projekten von Kollegen, wofÃ¼r man letztendlich natÃ¼rlich nicht verantwortlich gemacht werden will, weshalb ich den Zeitverbrauch als gerechtfertigt ansehe. Ohne die tatsÃ¤chlichen Anforderungen an meine Komponente in Vorhinein zu kennen sind weite Teile der frÃ¼heren Versionen stark Ã¼berdimensioniert. Es wurden erst Features erstellt und danach festgestellt, ob sie gebraucht werden. Hier hÃ¤tte man viel Zeit sparen kÃ¶nnen, auch wenn die Features teils interessant zum Implementieren waren. Alles in allem darf ich etwas mehr Fachliche- und Projektkompetenz aus dem Fach Datenverarbeitung in der Technik mitnehmen als ich mit hineingenommen habe.