

Masterprojekt - Bericht

Sommersemester 2010

Praktikumsbericht

Autor / Matrikelnummer:

Markus Auer, BSc / S0910564035

Gruppe:

INIF

Projektbetreuer:

DI (FH) Raimund EDLINGER

Projektzeitraum:

1. März 2010 – 7. Juli 2010

Bericht-Abgabetermin:

3. September 2010

Inhaltsverzeichnis

1. Ziel des Praktikums	3
2. Erreichte Ziele	4
3. Verwendete Entwicklungs-Software	4
4. Aufbau der Software.....	4
5. Servo Motoren.....	5
6. IFM 3D Kamera	13
7. Abschließendes	14

1. Ziel des Praktikums

Ziel dieses Projektes ist die Entwicklung einer Software zur Interpretation von 3D Umgebungssensordaten (als SLAM-Algorithmus wird ICP-Scan Matching verwendet). Teilaufgaben hierbei sind die Exploration, d.h. das möglichst effiziente Erkunden einer unbekanntem Umgebung, Visualisierung und die Lokalisierung.

Die Visualisierung der generierten Maps steht ebenfalls im Vordergrund, wie die einwandfreie Kommunikation über WLAN (802.11.a). Deshalb sollten auch Überlegungen angestrebt werden, wie die Schnittstellen zwischen Roboter und Zentraleinheit definiert sind.

Für die Realisierung wird ein Time of Flight Sensor (IFM), Laptop (Zentraleinheit) und Mark10 als Testroboter zur Verfügung stehen, der für den Einsatz in der RoboCup Rescue League gedacht ist.

Als Fortsetzung dieser Aufgabenstellung ist im Rahmen weiterer Projekte im 3.Semester Master und mit anschließender Masterarbeit eine Erweiterung des entwickelten Konzeptes denkbar. Mit Hilfe der Modellierung von 3D-Umgebungen sollen im weiteren Sinne Bereiche der Roboteransteuerung übernommen werden (Semi-autonome Ansteuerung der Flipper).

2. Erreichte Ziele

Im Verlauf des Projekts wurde das Konzept des Rescue Robots im Bereich Hardware umstrukturiert. Die Ansteuerung der Servomotoren wurde bis Dato durch eine Compact Rio Karte von National Instruments abgewickelt. Für diesen Zweck zu überdimensioniert. Stattdessen kommt jetzt ein Board zum Einsatz auf Basis des .Net Microframework. Das EMX Modul von GHI Electronics bietet die richtigen Funktionen und genügend Leistung für diesen Einsatz.

Die neuen Ziele für dieses Projekt waren nun einen Treiber zur Ansteuerung der Servomotoren zu schreiben. Des weiteren sich ein Protokoll zur Kommunikation über TCP/IP zu überlegen und die Ansteuerung der Time of Flight Kamera.

Diese Ziele wurden während des Projekts erfüllt.

3. Verwendete Entwicklungs-Software

Die Programmierung des EMX Moduls kann sehr einfach mit Visual Studio 2008 durchgeführt werden. Als zusätzliches Update wird außerdem das Microsoft .Net Microframework benötigt sowie das SDK für das EMX Modul das über die Seite des Herstellers bezogen werden kann.

4. Aufbau der Software

Für die Ansteuerung der Servomotoren sowie der Time of Flight Kamera wurden zwei Klassen erstellt, die im einzelnen erklären werde.

5. Servo Motoren

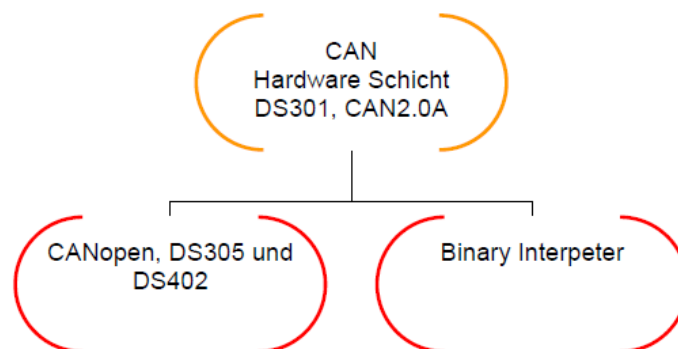
Für den Rescue Robot werden Servo Module der Firma Elmo Motion verwendet. Diese können über Can Bus angesteuert werden.

Die Antriebe des Rescue Robots setzen sich zusammen aus den vier Flippern den beiden Hauptantrieben und den beiden Mittelantrieben. Für jeden Flipper sowie Antrieb werden einzelne Can Id's vergeben für die Ansteuerung. Es können außerdem sämtliche Antriebe sowie sämtliche Flipper mit einer Id angesprochen werden.

Die einzelnen Id's können aus der folgenden Liste entnommen werden.

Antrieb	Can Id
Flipper Hinten Links	0x31B
Flipper Hinten Rechts	0x319
Flipper Vorne Links	0x315
Flipper Vorne Rechts	0x317
Alle Flipper	0x37F
Antrieb Rechts	0x303
Antrieb Links	0x305
Antrieb Mitte Rechts	0x309
Antrieb Mitte Links	0x307
Alle Antriebe	0x37E

Um mit den Elmo Modulen zu kommunizieren kann man einerseits das CanOpen Protokoll verwenden. Oder man verwendet einen einfacheren Binary Interpreter.



Da im Emx Modul kein CanOpen Stack implementiert ist viel die Wahl auf den Binary Interpreter, da er einfacher zu programmieren ist und eine Implementierung eines rudimentären CanOpen Stacks zu viel Zeit in Anspruch genommen hätte.

Die Binary Interpreter sind sehr einfach Aufgebaute Nachrichten. Sie stellen im Grunde nichts anderes dar als eine einfach Can Message mit 11 Bit Identifier und 8 Datenbytes. Also eine normales Can Message Frame.

Die Elmo Module können mit verschiedenen Befehlen parametrieret werden.

Die Liste der einzelnen Kommandos die im Programm verwendet wurden können aus der folgenden Tabelle entnommen werden. Dies die grundlegenden Befehle, die gesamte Liste der Befehle kann der Elmo Dokumentation entnommen werden.

Kommando	Beschreibung
MO	Motor Enable/Disable [boolean] MO=0: Schaltet den Motor stromlos. (Default nach dem Einschalten der Steuerung) MO=1: Aktiviert die Ansteuerung des Servomotors. Der Servomotor bendet sich im Regelbetrieb.
BG	Begin Motion [dimensionless] Startet eine im vorhinein denierte Bewegung.
PR	Relative Position [cnt] Mit PR=1000 dreht sich der Servomotor 1000 Inkremente(counts) von der aktuellen Position.
PA	Absolut Position [cnt] Nach dem Einschalten der Steuerung oder mit dem Kommando PX wird der Zähler des Drehgebers auf den Wert Null gesetzt. Diese Position entspricht nun der Referenzposition für die Absolute Position, d.h. mit PA=1000 dreht sich der Motor 1000 Inkremente(counts) bis er den Zählerstand PX=1000 erreicht hat.
PX	Main Position [cnt] Liest die aktuelle Position (Zählerstand des Drehgebers) ein. Wenn der Motor ausgeschaltet ist (MO=0) kann man den Zählerstand des Drehgebers mit PX=n verändern.
ST	Stop Motion [dimensionless] Stoppt die momentan ausführende Bewegung mit der Verzögerung DC. Stop Motion hat keinen Einfluss auf Motion enable/disable.
SP	Speed for PTP(point to point) Mode [cnt/sec] Setzt die maximale Geschwindigkeit des Servomotors. Am Beginn der Bewegung beschleunigt der Motor mit dem Wert von AC, bis er die Sollgeschwindigkeit von SP in counts/sec erreicht hat. Wird der Motor mit ST wieder ausgeschaltet, verzögert er mit dem Wert von DC, bis er zum Stillstand gekommen ist.
JV	Jogging Velocity [cnt/sec] Setzt die Drehzahl des Motors.

Eine Can Message setzt sich nun aus folgenden Bytes zusammen.

Aufbau eines Telegramms.

0x37F	0x4A	0x56	0x00	0x00	0xE8	0x03	0x00	0x00
Node ID 300h+127 300h+(Node ID)	ASCII Wert J	ASCII Wert V	Feldindex	Zuweisung oder nicht	Wert in HEX umgerechnet			

Wie man sehen kann besteht die Message aus der Id. Die ersten beiden Databytes stellen den Befehl dar der ausgeführt werden soll. Hier im Beispiel JV für Jogging Velocity. Diese beiden Bytes sind die einfache Übersetzung von Ascii in Hex. Das dritte Databyte stellt den Index dar falls vorhanden. Bei den Kommandos die im Programm verwendet werden bleibt dieses Feld immer auf 0x00. Das vierte Databyte setzt eine Zuweisung oder nicht. Die letzten Databytes ergeben den Wert des Befehls.

Bsp.:

MO=1

0x37F 0x4D 0x4F 0x00 0x00 0x01 0x00 0x00 0x00

JV=1000

0x37F 0x4A 0x56 0x00 0x00 0xE8 0x03 0x00 0x00

BG

0x37F 0x42 0x47 0x00 0x40 0x00 0x00 0x00 0x00

Wie man an diesem Beispiel gut erkennen kann ist der grundsätzliche Ablauf das man als erstes den Motor einschaltet. Dies geschieht mit dem Befehl MO=1. Als nächstes wird der Antrieb auf die entsprechende Betriebsart parametrisiert. In diesem Fall JV=1000, Jogging Velocity = 1000 cnt/sec. Das abschließende Kommando ist BG, Begin Motion, der das Kommando ausführt.

Am Beginn der Klasse Elmo.cs im Bereich *Can Variables* wird der Can Bus des EMX Boards initialisiert. Eine genauere Beschreibung kann aus der EMX Dokumentation entnommen werden. Im Bereich *Elmo Ids* wurden die entsprechenden Ids den Variablen zugeordnet. Im Bereich *Internal Methods* wurden die zuvor aufgelisteten Befehle implementiert.

```
private static void Elmo_Motion(ushort Id, byte motion):
```

Realisierung *Motor Enable/Disable* (MO)

`private static byte[] ToByteArray(Int32 b):`

Umrechnung eine 32Bit Integer in ein Byte Array

`private static void BG(ushort Id):`

Realisierung *Begin Motion* (BG)

`private static void PR(ushort Id, Int32 increments):`

Realisierung *Relative Position* (PR)

`private static void PA(ushort Id, Int32 increments):`

Realisierung *Absolut Position* (PA)

`private static void PX(ushort Id):`

Realisierung *Main Position* (PX)

`private static void ST(ushort Id):`

Realisierung *Stop Motion* (PX)

`private static void SetFlipperSpeed(ushort Id, Int32 speed):`

Realisierung *Speed for PTP(point to point) Mode* (SP)

`private static void SetDriveSpeed(ushort Id, Int32 speed):`

Realisierung *Jogging Velocity* (JV)

Im nächsten Schritt werden diese Methoden verwendet um die Bedienung der Flipper und des Antriebes zu erleichtern. Im Bereich *Drive Methods* und *Flipper Methods* sind diese Methoden implementiert.

`public static void DriveL(Int32 speed):`

Ansteuerung des linken Antriebs. Die Variable speed gibt dabei die Geschwindigkeit an.

`public static void DriveR(Int32 speed):`

Ansteuerung des rechten Antriebs. Die Variable speed gibt dabei die Geschwindigkeit an.

`public static void DriveMiddleL(Int32 speed):`

Ansteuerung des mittleren linken Antriebs. Die Variable speed gibt dabei die Geschwindigkeit an.

`public static void DriveMiddleR(Int32 speed):`

Ansteuerung des mittleren rechten Antriebs. Die Variable speed gibt dabei die Geschwindigkeit an.

`public static void DriveStop():`

Diese Methode stoppt sämtliche Antriebe, jedoch nicht die Flipper.

`public static void DriveStop(ushort Id):`

Mit dieser Methode kann ein einzelner Antrieb gestoppt werden.

Mit diesen Methoden wird die Bedienung der Antriebe sehr erleichtert da bereits der benötigte Ablauf implementiert ist. Im Bereich Flipper Methods sind die Methoden für die einzelnen Flipper implementiert.

`public static void FlipperFl(Int32 increments, bool isRelative):`

Ansteuerung des linken Flippers. Die Variable increments gibt dabei an wieviele incremente der Servomotor machen soll. Die Variable isRelative gibt an ob es sich dabei um eine Relative Positionsveränderung handelt oder nicht.

`public static void FlipperFr(Int32 increments, bool isRelative):`

Ansteuerung des rechten Flippers. Die Variable increments gibt dabei an wieviele incremente der Servomotor machen soll. Die Variable isRelative gibt an ob es sich dabei um eine Relative Positionsveränderung handelt oder nicht.

`public static void FlipperRl(Int32 increments, bool isRelative):`

Ansteuerung des hinteren linken Flippers. Die Variable increments gibt dabei an wieviele incremente der Servomotor machen soll. Die Variable isRelative gibt an ob es sich dabei um eine Relative Positionsveränderung handelt oder nicht.

`public static void FlipperRr(Int32 increments, bool isRelative):`

Ansteuerung des hinteren rechten Flippers. Die Variable increments gibt dabei an wieviele incremente der Servomotor machen soll. Die Variable isRelative gibt an ob es sich dabei um eine Realtive Positionsveränderung handelt oder nicht.

`public static void FlipperSpeed(Int32 speed):`

Diese Methode setzt die Umdrehungsgeschwindigkeit der gesamten Flipper.

`public static void FlipperSpeed(ushort Id, Int32 speed):`

Mit dieser Methode kann die Umdrehungsgeschwindigkeit eines einzelnen Flippers verändert werden.

`public static void FlipperStop():`

Mit dieser Methode werden sämtliche Flipper gestoppt.

`public static void FlipperStop(ushort Id):`

Mit dieser Methode kann ein einzelner Flipper gestoppt werden.

`public static void FlipperReference():`

Mit dieser Methode wird ein Referenzfahrt der Flipper gestartet.

`private static void RightButton_OnInterrupt(uint port, uint state, DateTime time):`

Endlage des Flippers für die Referenzfahrt

`private static void LeftButton_OnInterrupt(uint port, uint state, DateTime time):`

Endlage des Flippers für die Referenzfahrt

`private static void DownButton_OnInterrupt(uint port, uint state, DateTime time):`

Endlage des Flippers für die Referenzfahrt

`private static void UpButton_OnInterrupt(uint port, uint state, DateTime time):`

Endlage des Flippers für die Referenzfahrt

Die abschließenden zwei Methoden in dieser Klasse befinden sich im Bereich *Elmo StartStop*.

`public static void Start():`

Diese Methode muss einmal nach einem Power Reset ausgeführt werden. Hier werden sämtliche Module am Can Bus zurückgesetzt.

`public static void Stop():`

Bei dieser Methode werden sämtliche Antriebe (Flipper und Drive) gestoppt.

Diese Methode ist das Kernstück für die Steuerung des Rescue Robots. Die bereits beschriebenen Funktionen werden im Hauptprogramm verwendet um eine korrekte Ansteuerung über TCP/IP zu gewährleisten.

Das Hauptprogramm ist grundsätzlich so aufgebaut das in einer Endlosschleife der TCP/IP Socket abgefragt wird ob Daten im Buffer des EMX Boards vorhanden sind. Ist dies der Fall werden die Daten in einer separaten Klasse ausgewertet.

Code:

```
while (true)
{
    //Warten auf Client.Accept
    Socket clientSocket = listeningSocket.Accept();

    //Aufruf der Klasse ProcessClientRequest,...
    new ProcessClientRequest(clientSocket, false);
}
```

Sind nun Daten am Buffer vorhanden wird in die Methode

`public ProcessClientRequest(Socket clientSocket, Boolean asynchronously)`

in der Klasse *ProcessClientRequest* gesprungen.

Diese Methode hat auch noch die Möglichkeit, dass pro anstehenden Daten ein neuer Thread geöffnet werden. Diese Entscheidung wird mit der Variable *Boolean asynchronously* getroffen.

Code:

```
if (asynchronously)
{
    new Thread(ProcessRequest).Start();
}
```

```
else
{
    ProcessRequest();
}
```

In der Methode *ProcessRequest()* wird der ankommende Stream angearbeitet. Es wird zu Beginn das Elmo Startkommando gesendet.

Code:

```
//Start Kommando für die Elmo Module
Elmo.Start();
```

Anschließend werden die empfangenen Bytes in einen String konvertiert und die einzelnen Befehle getrennt.

Code:

```
//Konvertieren der Bytes in einen String
string message = new string(Encoding.UTF8.GetChars(buffer), 0, bytesRead);

//Unterteilen der String Message in die einzelnen Befehle. Dabei wurde ';'
als Trennzeichen beutzt.
string[] ipSplit = message.Split(';');
```

Die Befehle werden nun nacheinander abgearbeitet. Die einzelnen Informationen befinden sich in einem Stream. Dabei sind die Daten durch ; getrennt.

Translation X; Translation Y; Translation Z; Rotation X; Rotation Y; Rotation Z; Rotation um eigene Achse; Button

Die Abfrage der Buttons ist als Hexzahl im Buffer und muss konvertiert werden um die einzelnen Buttons abrufen zu können. Aus diesem Grund wurde auch die Methode

```
private string hex2binary(string hexvalue)
```

implementiert. Rotation und Translation werden als double Werte übergeben deshalb wurde auch hier eine Methode implementiert die die konvertierung durchführt.

```
public static double ToDouble(string In)
```

6. IFM 3D Kamera

Als zweites kleines Projekt wurde neben der Ansteuerung der Servomotoren auch ein Treiber geschrieben um mit einer 3D Kamera über Ethernet zu kommunizieren. Diese Kamera basiert auf dem Time of Flight Prinzip mit dem eine Terrainklassifikation ermöglicht wird.

Die Basis des Projekts ist eine Dll die von der Herstellerfirma Ifm zur Verfügung gestellt wird. Da diese Dll nicht mit .Net kompatibel ist muss ein Wrapper geschrieben werden der die beinhalteten Befehle ausführen kann.

Dies geschieht im allgemeinen in der Klasse *O3D2xx.c*. Der Prozess für das schreiben eines Wrappers ist meist eine schwierige und langwierige Aufgabe. Es gibt jedoch Möglichkeiten diesen Prozess etwas zu Beschleunigen.

Ein nützliches Tool für die Generierung ist das *Pinvoke Signature Toolkit*. Hier kann nach Programmstart im Bereich Translate Snippet ein C++ Code in einen äquivalenten C# Code mit den benötigten Dllimport Befehlen generiert werden.

Da im SDK für die Kamera Beispiele für die Programmierung enthalten waren konnte ganz einfach die vorhandene Header Datei in einen C# Code generiert werden.

Der Code musste noch geringfügig abgeändert werden um voll funktionsfähig zu sein. Eine Beschreibung der einzelnen Methoden kann aus der beiliegenden Ifm Dokumentation entnommen werden.

Dieser Klasse wurden außerdem noch einige Methoden hinzugefügt die die Bedienung erleichtern sollen.

public int GetDistance(**UInt32** remotePort):

Wird diese Methode ausgeführt wird das Distanz Image von der Kamera angefordert. Dieses wird in der Form eines float[] gespeichert.

public int GetAmplitude(**UInt32** remotePort):

Wird diese Methode ausgeführt wird das Amplitude Image von der Kamera angefordert. Dieses wird in der Form eines float[] gespeichert.

public int GetCartesian(**UInt32** remotePort):

Wird diese Methode ausgeführt werden die Kartesischen Koordinaten von der Kamera angefordert. Dieses werden in der Form von float[] gespeichert.

public int GetLine(**UInt32** remotePort):

Wird diese Methode ausgeführt wird das Linien Image von der Kamera angefordert. Dieses wird in der Form eines float[] gespeichert.

private void IsDataValid(**O3D2XXImageHeader** imageHeader):

Überprüfung ob das angeforderte Image korrekt ist.

Mit dieser Klasse ist die Kamera nun voll ansteuerbar. Im Zuge des Masterprojekts im 3 Semester kann diese Klasse nun eingesetzt werden um eine Terrain Klassifikation durchzuführen.

7. Abschließendes

Diese beiden Projekte wurden kurzfristig gestartet da die Hardware des Rescue Robots umstrukturiert wurde. Das Ursprüngliche Ziel war es die Terrain Klassifikation einzubinden. Diese Aufgaben können nun im 3 Semester durchgeführt werden.

Des weiteren sollte das Protokoll der TCP/IP Verbindung nochmals überdacht werden um die Auslastung am Netzwerk gering zu halten.