Contents

1 Teammitglieder

Betreuer:

Wei Lou
E-mail: wei.luo@itm.uni-stuttgart.de

Mitglieder:
Jakob Englert
Joachim Hecker
Sebstian
Hannes Pfitzner

2 Einleitung

Bei dieser Projektarbeit geht es um die Entwicklung einer Paketgreiffunktion für einen bereits funktionierenden Quadrokopter. Die zu entwickelnde Hardware lässt sich dabei in vier Kategorien einteilen: Ein Greifarm, der in der Lage ist, Pakete mit einer Größe von ca. 10 cm zu greifen und zu halten; eine Kamera, die das zu greifende Paket erkennt; diverse Sensorik und Aktorik für den Greifarm und anderweitige Kontrollfunktionen; ein System-on-a-Chip (SoC) für die Bildanalyse, Regelung und Steuerung des Greifarms und des Quadrokopters, wobei dieser selbst bereits mit einem weiteren SoC ausgestattet ist, mit welchem für die Steuerung nur kommuniziert werden muss. Es gibt eine Vielzahl verschiedener SoCs, wobei sich hier aufgrund weiter Verbreitung für den Raspberry Pi (RPi) entschieden wurde, welcher, salopp gesagt, als das Gehirn der neu entwickelten Funktionalität dient. Im Folgenden soll es nun um die grundlegende Softwarearchitektur und -infrastruktur gehen, welche die Aufgaben des Raspberry Pis auf diesem implementiert.

3 Die Greifermechanik

4 Sesorik, Aktorik und andere Hardware

5 Softwareimplementierung

5.1 Betriebssystem und Entwicklungsumgebung

5.2 Softwarearchitektur

5.2.1 Single Application vs. ROS

Wie es häufig in der Entwicklung der Fall ist, stehen auch hier für die Implementierung der Software auf dem RPi verschiedene Plattformen, Frameworks und Programmiersprachen zu Verfügung. Bei der Programmiersprache viel die Wahl recht schnell auf Python, da es einem im Vergleich zu C/C++, aber auch Java viel Overhead abnimmt, in der allgemeinen Komplexität geringer ist und vielerorts (wie auch in ROS, siehe unten) nativ unterstützt wird.

Bei der Softwarearchitektur wurde sich zunächst an einer normalen objektorientierten Struktur bedient: Es gibt eine main-Datei, welche vergleichsweise klein ist und nur die grobe Struktur bzw. den Gesamtablauf darstellt. Diese ruft dann weitere Softwaremodule auf, die die Unteraufgaben, wie beispielsweise das Empfangen und Verarbeiten von Bilddaten, implementiert. Ein Problem, das hier recht schnell auftritt ist, dass theoretisch zwei Prozesse gleichzeitig ablaufen müssen. Wenn ein Motormodul beispielsweise gerade darauf wartet, dass die Sollposition erreicht wurde, müsste in einem Programmierstrang, in dem ein Befehl nach dem anderen ausgeführt wird, das Kameramodul mit der Verarbeitung des Bildes auf das Motormodul warten, obwohl das Motormodul gerade gar nichts tut, außer auf den Motor in der echten Welt zu warten, bis er die richtige Position hat. Da das Kameramodul aber an das Motormodul eventuell kommunizieren muss, dass das Paket gar nicht mehr an der richtigen Stelle ist, weil der Quadrokopter aufgrund äußerer Umstände nicht ganz stillsteht, müssen beide Module gleichzeitig arbeiten und miteinander kommunizieren können. In der traditionellen Softwareentwicklung gibt es dafür sogenannte Threads. Diese sind separate Ausführungsstränge, die gleichzeitig ausgeführt werden und sich dann beispielsweise ein Datenobjekt im Speicher teilen und über dieses miteinander kommunizieren können. Das Problem bei dieser Sache: Multithreading (mehrere Threads) ist eine recht komplexe Angelegenheit, da sehr vielseitige Probleme auftreten können. Will zum Beispiel Prozess A auf die Variable Paketposition im Datenobjekt zugreifen und Prozess B will diese gleichzeitig beschreiben, wird das Programm abstürzen.

Bis zu einem gewissen Grad war es möglich die Funktionalitäten mit Multithreading zu implementieren. Allerdings wurde es im Laufe der Entwicklung immer komplizierter, da Bilddaten beispielsweise von zwei Ressourcen abhängen, die nicht immer verfügbar sind: zum einen der Speicherplatz im Datenobjekt, das andere Threads lesen und damit belegen wollen und zum anderen

von der Kamera selbst, die natürlich auch nicht willkürlich Bilddaten liefert. Deshalb basiert die endgültige Software auf ROS, das genau diese Probleme adressiert. ROS steht dabei für Robot Operating System und ist ein Framework, das auf dem Publisher-Subscriber-Modell basiert. Die Idee ist dabei, dass es verschiedene Themen (Topics) gibt, in die verschiedene dezentrale Skripte, die gleichzeitig laufen, Informationen zu einzelnen Themen veröffentlichen oder abonnieren können. So gibt es beispielsweise ein Kameraskript, das das Bild und die erkannte Position des Pakets veröffentlicht und ein Motorskript abonniert die Paketposition und kann daraus dann weitere Schlüsse ziehen. Ein anderes Skript könnte dann auf einem anderen Rechner (z.B. Laptop) im gleichen (WLAN-) Netzwerk das Kamerabild abbonnieren und mit sehr wenig Programmieraufwand anzeigen. Das elegante dabei ist, dass sämtliche vorher beschriebene Multithreading-Probleme dabei vom sogenannten ROS-Core, der zentrale Schaltstelle des ROS, übernommen werden und damit die einzelnen (selbstgeschriebenen) Skripte sehr entschlackt werden. Das macht es deutlich einfacher diese zu debuggen. ROS ist zudem sehr gut dokumentiert und für viele Probleme gibt es bereits vorgefertigte Skripte, die Dank des einfachen P/S-Modells auch einfach anzubinden sind.

5.3 Robot Operating System (ROS)

5.3.1 Mavros und Simulation

5.4 Dronensteuerung

6 Bilderkennung

7 Gesamtintegration

8 Quellenverzeichnis