HOCHSCHULE
HAMM-LIPPSTADT

Systemeinrichtung zum Arbeiten mit ROS2 Humble

Autor:	Benjamin Dilly
Matrikelnummer:	2200380
Hochschule:	Hochschule Hamm-Lippstadt Marker Alle 76-78 59063 Hamm
Studiengang:	Mechatronik
Betreuender Professor:	Prof. Dr.-Ing. Ulrich Schneider

Ort und Datum der Abgabe:	Lippstadt, 24. Juni 2024

Vorwort

Dieses Dokument ist Bestandteil der Bachelorarbeit: „Implementierung von FastSLAM 2.0 und Tests in Outdoor-Simulationsumgebungen“[1].

Die in diesem Dokument aufgeführten Informationen basieren auf einer Recherchekombination von: Web, Videos und eigenständigem Probieren bzw. Testen. Es kann in bestimmten Fällen durchaus auch andere Möglichkeiten geben, um das gleiche Ergebnis zu erzielen. Es ist jedoch zu sagen, dass die in diesem Dokument aufgeführten Methoden, sich im Rahmen meiner Bachelorarbeit als funktionierend und somit als „für mich korrekt“ herausgestellt haben.

[1] https://wiki.hshl.de/wiki/index.php/Implementierung_von_FastSLAM_2.0_und_Tests_in_Outdoor-Simulationsumgebungen

Inhalt

Vorwort 2
Inhalt I
Abbildungsverzeichnis II
1 Einrichtung des Ubuntu Betriebssystems 3
1.1 Probleme und Zugriff auf Daten der Partition über Windows 12
1.2 Boot-Manager-Anforderungen 12
1.3 Vor dem Einrichten 12
2 Einrichtung aller notwendigen Komponenten 13
2.1 Installation der ROS-Pakete 13
2.2 Installation und Einrichtung der Entwicklungsumgebung 13
2.3 Tools zum Erstellen eines ROS-Projekts 14
2.4 Erklärung der Visual Studio Code Entwicklungsumgebung 15
Funktion des .vscode Ordners 15
2.4.1 Funktion des install Ordners 16
2.4.2 Bedeutung des src Ordners 17
3 Bauen eines ROS-Paketes mit Colcon und Ausführen einer ROS-Node 19
4 ROS-Kurzeinführung 23
5 CMake-Einführung 26
5.1 Erstellen eines CMake Projektes 26
5.2 CMake If-Abfrage 26
5.3 CMake Variablen 26
5.4 CMake Pfade 28
5.5 Zusammenführen von CMakeLists.txt 28
5.6 CMake-Objekt vs. Variable 29
5.7 Praktisches Beispiel 29
5.7.1 Ausgangssituation 29
5.7.2 Master-CMakeLists-Header-Anpassung 30
5.7.3 ROS-Spezifische Paketeinbindung (find_packages) 30
5.7.4 Erstellen eines Targets 31
5.7.4.1 Bibliotheks-Target 31
5.7.4.2 Auszuführenden-Target 31
5.7.4.3 Target-Include-Verzeichnis 32
5.7.4.4 Target-Compile-Features 32
5.7.4.5 ROS-Spezifische Target-Abhängigkeiten 33
5.7.4.6 Binden einer Bibliothek zu einem Target 33

5.8 Ausgabenerstellung 34
5.8.1 Erzeugen einer Bibliotheksdatei aus Bibliothekstarget & einer Auszuführenden aus Auszuführenden-Target 34
5.8.2 Export und Installation von Ordnern für Auszuführende 35
5.8.3 ROS-Installationsortangabe 35
6 Debugging mit CMake und VSCode 36
6.1 Voraussetzung 36
6.1.1 c_cpp_properties.json 36
6.1.2 settings.json 38
6.2 CMake Debugging ausführen 39

Abbildungsverzeichnis

Abbildung 1 Eigene Darstellung der Rufus-Oberfläche 4
Abbildung 2 Eigene Bildergalerie des erweiterten Neustarts, für Bootreihenfolge 5
Abbildung 3 Eigener UEFI-Ausschnitt der Änderung der Bootreihenfolge 6
Abbildung 4 Eigene Darstellung der Festplattenpartitionen 7
Abbildung 5 Eigene Darstellung der Bearbeitung einer bestehenden Partition 7
Abbildung 6 Beispiel einer nicht zugeordneten freien Partition 7
Abbildung 7 Startfenster des Ubuntu-Installationsmanager. Eigener Ausschnitt 8
Abbildung 8 Sprachauswahloberfläche im Ubuntu-Installationsmanager. Eigener Ausschnitt 8
Abbildung 9 Auswahloberfläche für erweiterte Pakete und Treiber zur Ubuntu-Installation des Ubuntu-Installationsmanager. Eigener Ausschnitt 9
Abbildung 10 Auswahloberfläche für den Installationsort des Ubuntu-Systems des Ubuntu-Installationsmanager. Eigener Ausschnitt 9
Abbildung 11 Partitionsübersichtsfenster des Ubuntu-Installationsmanager. Eigener Ausschnitt 10
Abbildung 12 Auswahldarstellung des Installationslaufwerks D in der Partitionsübersicht. Eigener Ausschnitt 10
Abbildung 13 Größenreservierung und Formatierung der ausgewählten Installationspartition. Eigener Ausschnitt 11
Abbildung 14 Eigene Darstellung der reservierten und Partitionierten Installationspartition 11
Abbildung 15 Eigener Ausschnitt der ROS-Erweiterung in VSCode 14
Abbildung 16 Eigener Ausschnitt der CMake-Erweiterung in VSCode 14
Abbildung 17 Eigener Ausschnitt des ROS-Projetordners in VSCode 15
Abbildung 18 Eigener Ausschnitt des .vscode Ordners in VSCode 15
Abbildung 19 Eigener Ausschnitt des Installationsordners in VSCode 16
Abbildung 20 Eigener Ausschnitt des Quellenordners in VSCode 17
Abbildung 13 Eigene Darstellung der Nodes 23
Abbildung 14 Eigene Darstellung der Topic-Kommunikation 23
Abbildung 15 Eigene Darstellung der Service-Kommunikation.....24
Abbildung 16 Eigene Darstellung der Action-Kommunikation 25

1 Einrichtung des Ubuntu Betriebssystems

Da für ROS2 das Ubuntu-Betriebssystem benötigt wird, muss dieses in einer geeigneten Form auf einem ausgewählten System installiert werden. Hierfür könnte zum einen ein Raspberry Pi verwendet werden oder ein Linux-Rechner auf dem nur Ubuntu läuft. Es kann aber auch parallel zu Windows installiert werden. Hierbei ist die letzte Variante, der parallelen Installation neben Windows, die sinnvollste. Die ergibt sich aus den folgenden Gründen:

1. Es wird keine neue Hardware benötigt,
2. Es kann einfach zwischen beiden Systemen gewechselt werden, ohne das Gerät zu wechseln,
3. Es wird kein extra Geld ausgegeben, da Ubuntu kostenlos ist,
4. Es kann die Leistungsstärkere Hardware des bestehenden Rechners verwendet werden.

Insbesondere der letztere Punkt ist für die Simulation am wichtigsten, damit diese eine Echtzeitfähigkeit erlangen. Es gibt hierbei zwei Arten der parallelen Nutzung des Ubuntu-Betriebssystems mit Windows auf einem Rechner:

1. Einrichtung mittels einer virtuellen Maschine, wie VirtualBox,
2. Oder Einrichtung eines Dual-Boot-Systems.

Hierbei wurde sich in dieser Arbeit für das Dual-Boot-System entschieden. Dies resultiert aus den Gründen:

1. Die Hardware wird nicht geteilt und steht dem jeweiligen System zu hundert Prozent zur Verfügung.
2. Der Prozessor ist weniger ausgelastet und somit sparsamer.
3. Das System läuft flüssiger und die Simulationen bekommen eine Echtzeitfähigkeit.
4. Es kann die systemeigene Konsole des Ubuntu-Systems verwendet werden.

Insbesondere die Leistungsgründe haben dazu geführt, dass das Dual-Boot-System verwendet wurde.

Einrichtung des Ubuntu Betriebssystems

Die Einrichtung wurde auf einem Intel® Core™ i7-7700HQ mit 2.80 GHz, 16 GB RAM und einer Nvidia® GTX1060 vorgenommen. Des Weiteren wurde ein Speicherplatz von 60 GB reserviert.

Für die Einrichtung des Dual-Boot-Systems werden folgende Komponenten benötigt:

1. Ein funktionstüchtiger Windows-Rechner mit installiertem Windows,
2. Eine oder mehrere Festplatten mit mindestens 60 GB freiem Speicherplatz,
3. Eine USB-Maus sowie eine integrierte Tastatur oder USB-Tastatur,
4. Einen Boot-fähigen USB-Stick mit mindestens 10 GB Speicherplatz,
5. Die Software „Rufus“ für USB-Datenträger
6. Und das ISO-Image der Ubuntu-Desktopversion 22.04.3 LTS.

Um Windows vorzubereiten, müssen folgende Schritte unternommen werden:

1. Schnellstart deaktivieren,
2. Einrichten der Bootreihenfolge
3. Und Erstellen einer ext-4 formatierten Partition auf einer der verfügbaren Festplatten.

Die Deaktivierung des Schnellstartes, ermöglicht es bei einem Neustart durch das Drücken der F12 Taste in den Boot-Manager zu gelangen und kann unter `Einstellungen > Netzbetrieb und Energiesparmodus > Zusätzliche Energieeinstellungen > Auswählen, was beim Drücken von Netzschaltern geschehen soll` deaktiviert werden.

Um die Bootreihenfolge einzustellen, muss zunächst der gewählte USB-Stick mittels Rufus und dem ISO-Image beschrieben werden.

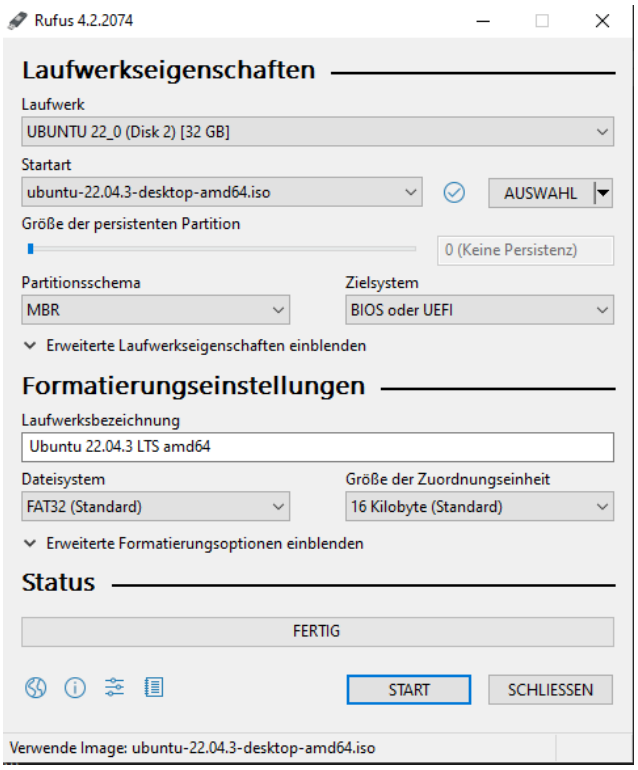


Abbildung 1 Eigene Darstellung der Rufus-Oberfläche

Um den USB-Stick zu beschreiben, muss zunächst Rufus geöffnet werden. Danach werden unter `Laufwerk` alle verfügbaren USB-Geräte aufgelistet. **Zu beachten ist, dass alle Daten auf dem ausgewählten Gerät gelöscht werden, wenn dieses beschrieben wird**. Nach der Auswahl des Sticks muss nun nach Klicken auf `AUSWAHL` das ISO-Image ausgewählt werden, welches nach erfolgreicher Auswahl unter `Startart` angezeigt wird. Nachdem das Image ausgewählt wurde, werden alle zuvor ausgegrauten Flächen aktiviert. Der Name des Bootlaufwerks, welches beim Ausführen des späteren Dual-Boots angezeigt wird, steht unter Laufwerksbezeichnung. Als letzter Schritt muss das Image auf den USB-Träger übertragen werden. Hierfür muss auf `START` geklickt werden, wobei die Standardeinstellungen übernommen werden.

Um jetzt die Bootreihenfolge einzustellen, muss der Stick eingesteckt bleiben. Durch die Eingabe von `uefi` in die Windows-Suche und der Auswahl von `Optionen für den erweiterten Start ändern` wird der erweiterte Start geöffnet. Die Bootreihenfolge kann dann durch `Jetzt neu starten` eingestellt werden.

Bevor dieser Schritt ausgeführt wird, sollte überprüft werden, ob das System das Booten von einem externen Datenträger unterstützt. Hierfür sollte einmal die Systeminformationen-App geöffnet werden. Nach dem Öffnen sollte unter `Systemübersicht` auf der rechten Seite unter `BIOS-Modus` UEFI stehen. Ist dies der Fall, kann fortgefahren werden.

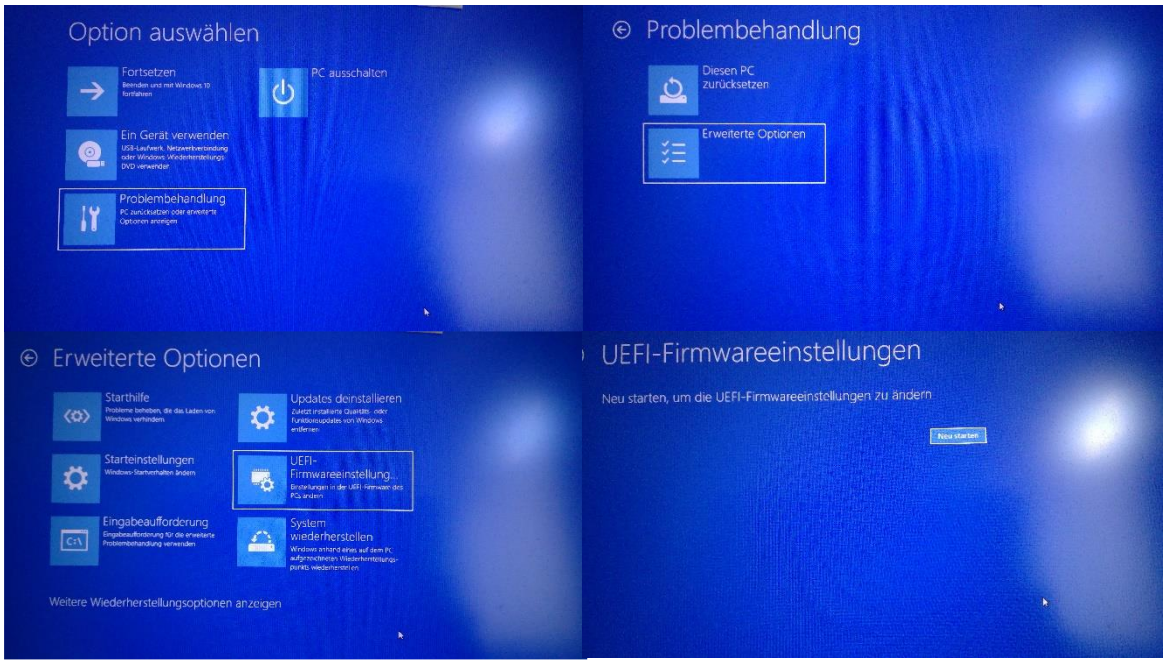


Abbildung 2 Eigene Bildergalerie des erweiterten Neustarts, für Bootreihenfolge

Das Vorgehen nach dem Ausführen des erweiterten Neustarts, wird in der obigen Bildergalerie von links oben nach rechts unten gezeigt.

Im Anschluss wird das UEFI gestartet. Dies kann je nach Version anders aussehen und sogar eine Mausunterstützung sowie eine ausgereiftere Nutzeroberfläche besitzen.

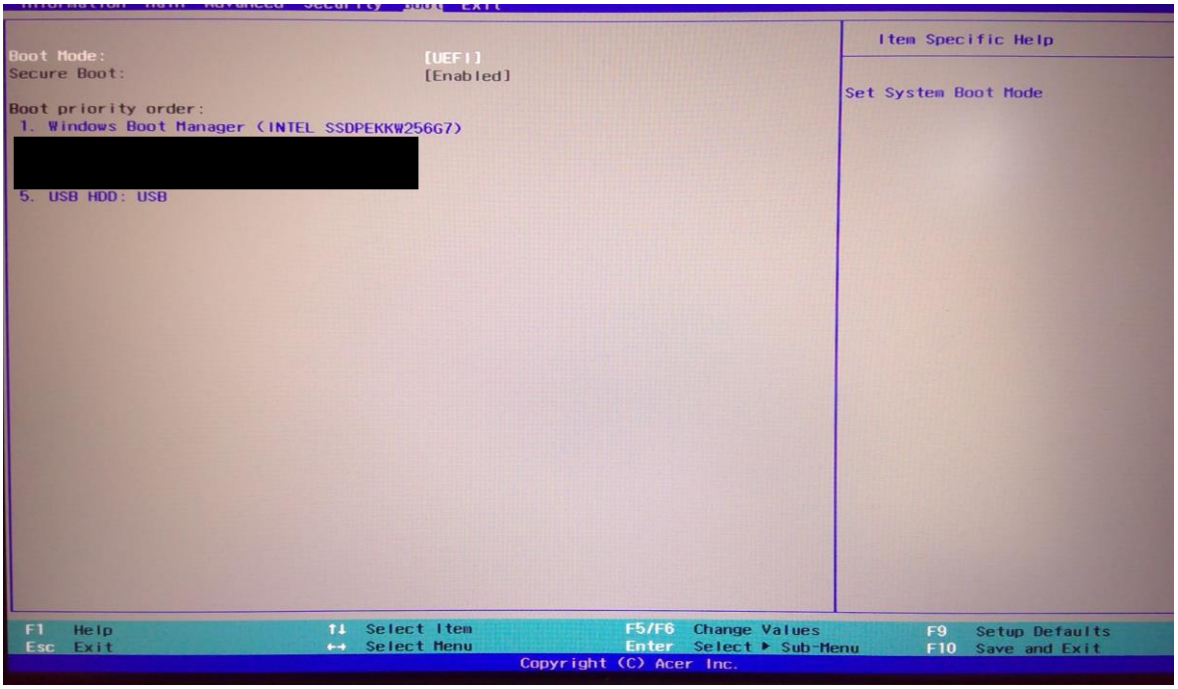


Abbildung 3 Eigener UEFI-Ausschnitt der Änderung der Bootreihenfolge

Um die Bootreihenfolge festzulegen, muss in das Boot-Menü gewechselt werden. Bei der hier dargestellten Version geschieht dies mittels der rechten Pfeiltaste. Wie zu sehen ist, wird der Boot-fähige USB-Stick hier an Stelle fünf angezeigt. Damit keine Verwirrungen entstehen, sollte immer nur ein solcher Stick eingesteckt sein.

Ist nicht klar, welches angezeigte Gerät der bootfähige Stick ist, so bietet es sich an, den Bootmanager vor dem Einstecken des Sticks bereits einmal aufzurufen, um diesen danach dann als neues Gerät in der Liste zu identifizieren.

Zum Ändern der Reihenfolge muss nun mit der nach unten zeigender Pfeiltaste zu diesem Punkt navigiert werden. Das Verschieben wird dann unter `Item Specific Help` in der rechten Bildhälfte erklärt. Damit nach dem Speichern der Einstellung durch die F10 Taste vom USB-Stick gestartet wird, muss der USB vor den Windows Boot Manager geschoben werden.

Wird dann die Einstellung verlassen, startet eine Ubuntu-Vorschau von dem USB.

Bevor dies gemacht wird, sollte zunächst eine Partition erstellt werden, also ein Speicherplatz reserviert werden, in dem das Ubuntu-Betriebssystem installiert werden kann. Hierfür muss zunächst in die Windows-Suche `Partition` eingegeben werden. Danach muss die App: `Festplattenpartitionen erstellen und formatieren` geöffnet werden.

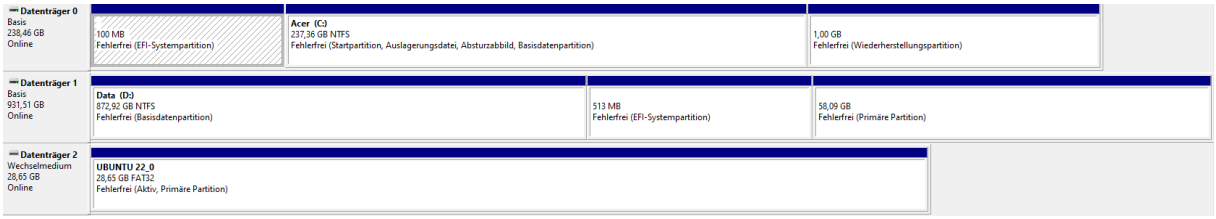


Abbildung 4 Eigene Darstellung der Festplattenpartitionen

Wie zusehen ist, ist Ubuntu hier mit knapp 60 GB reserviertem Speicherplatz auf der zweiten Festplatte installiert. Zudem kann der Boot-fähige USB unter dem Datenträger 3 gesehen werden, während Windows auf dem ersten Datenträger, also Datenträger 0 oder Laufwerk C installiert ist. Die jeweiligen Bootloader für beide Systeme sind in den EFI-Partitionen der jeweiligen Datenträger zu finden.

Um eine Partition für das Ubuntu-System zu erstellen, muss zunächst eine bestehende NTFS-Partition verkleinert werden. Dies geschieht durch das Klicken mit der rechten Maustaste auf die entsprechende Partition und das Auswählen von `Volume verkleinern.`

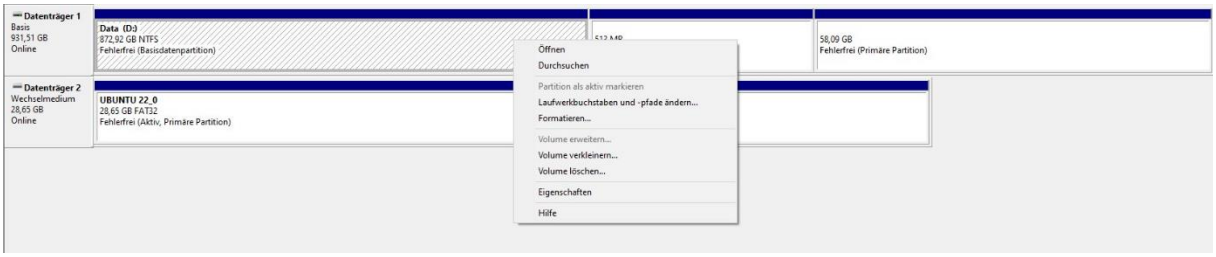


Abbildung 5 Eigene Darstellung der Bearbeitung einer bestehenden Partition

Um die 60 GB Speicher freizugeben, muss in die Kachel `zu verkleinernder Speicherplatz in MB` die Zahl 60.000 eingegeben werden. Nach der Verkleinerung wird der freie Speicher mit einem grauen Balken angezeigt und als „nicht zugeordnet“ deklariert.

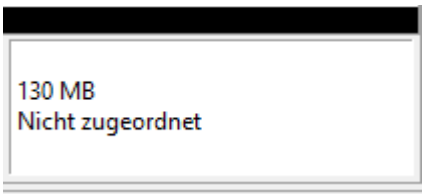


Abbildung 6 Beispiel einer nicht zugeordneten freien Partition

Wurde der freie Speicher auf dem ausgewählten Datenträger erstellt, kann das Booten vom USB-Stick mittels des UEFI und der geänderten Bootreihenfolge ausgeführt werden.

Beim folgenden Neustart des Computers, wird Ubuntu vom USB-Stick gestartet. Hierbei wird zunächst eine Bootoberfläche angezeigt, in der ausgewählt werden kann, ob Ubuntu getestet oder installiert werden soll, oder ob man doch Windows starten möchte. Die Auswahl „`try or install ubuntu`“ sollte hierbei standardmäßig ausgewählt sein. Sollte dies nicht der Fall sein, muss diese mittels der vertikalen Pfeiltasten auswählen und `ENTER` drücken.

Danach erscheint die folgende Oberfläche:

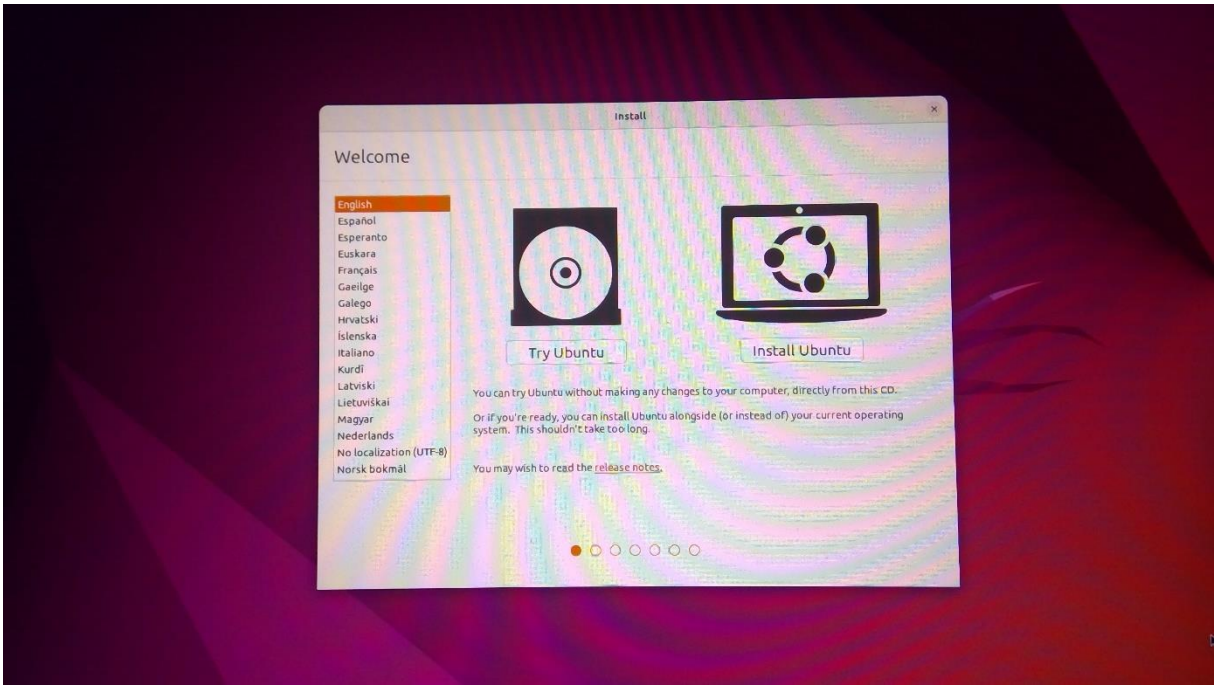


Abbildung 7 Startfenster des Ubuntu-Installationsmanager. Eigener Ausschnitt

Zum Installieren muss `Install Ubuntu` ausgewählt und danach die Sprache eingestellt werden.

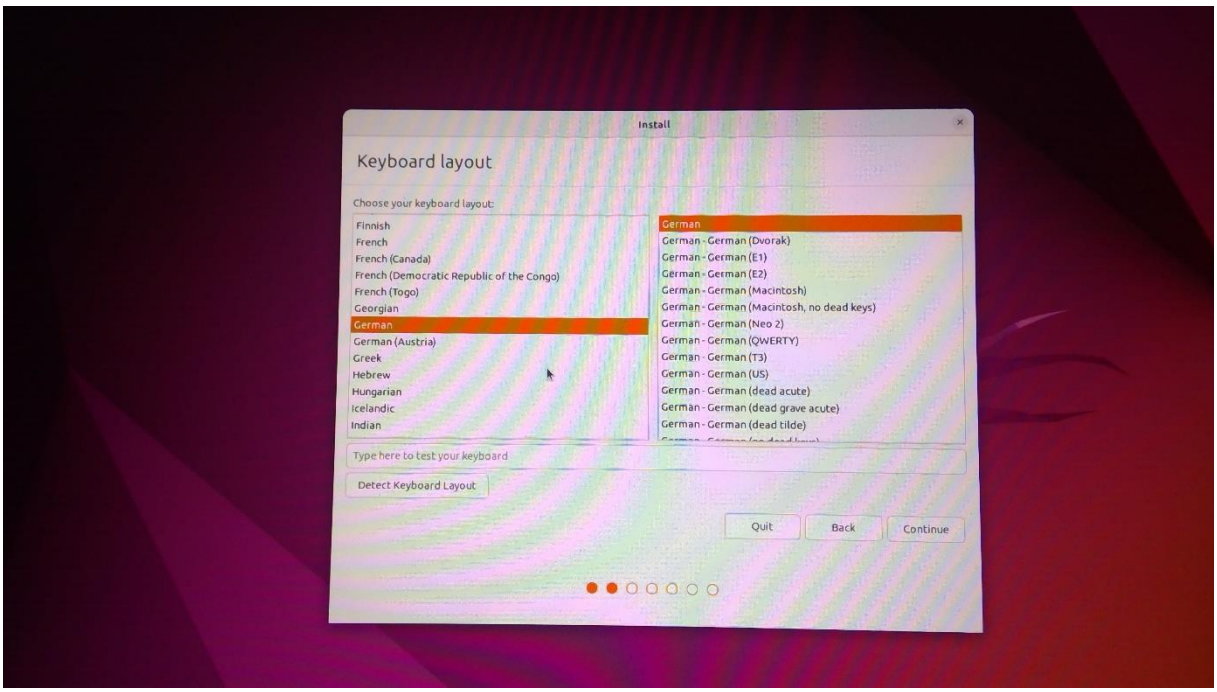


Abbildung 8 Sprachauswahloberfläche im Ubuntu-Installationsmanager. Eigener Ausschnitt

Im darauffolgenden Fenster sollten ebenso die Drittanbietersoftwarepakete installiert werden, damit später keine weiteren Pakete zugefügt werden müssen und keine Hardwareprobleme entstehen, wie dass bspw. USB-Treiber oder Grafikkartentreiber fehlen.

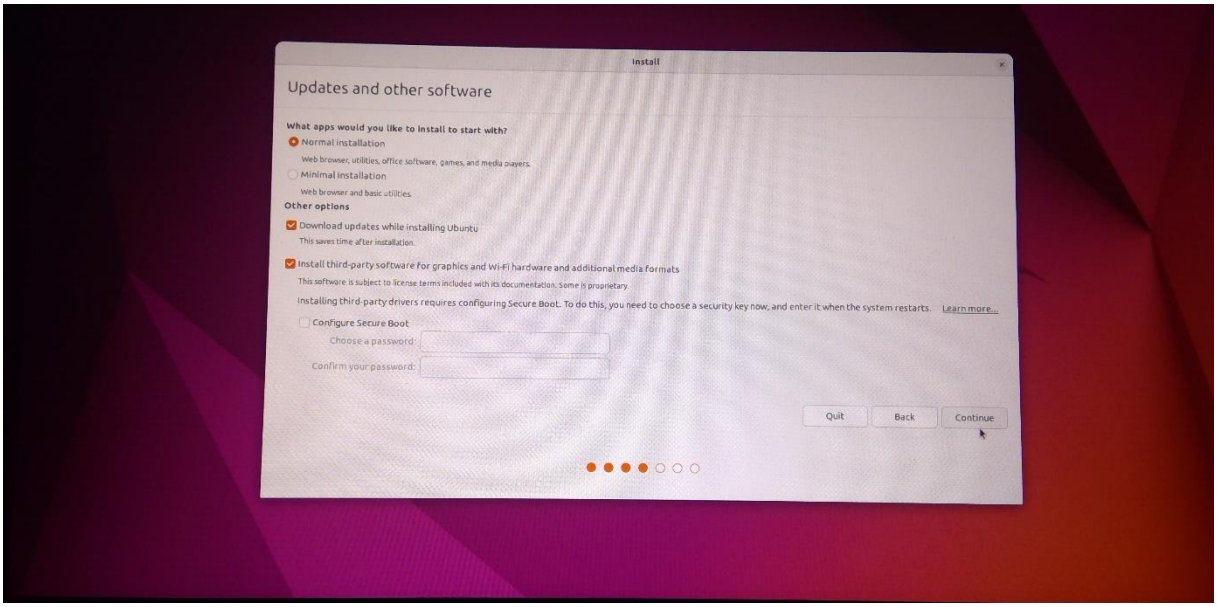


Abbildung 9 Auswahloberfläche für erweiterte Pakete und Treiber zur Ubuntu-Installation des Ubuntu-Installationsmanager. Eigener Ausschnitt

Hierbei muss noch Secure Boot eingerichtet werden.

Das nächste Fenster ermöglicht dann die Auswahl der Installation, also auf welchem Datenträger Ubuntu installiert werden soll. Da Ubuntu bereits installiert war, steht beim ersten Punkt etwas anders. Normalerweise würde dort stehen: Ubuntu neben Windows installieren, was bedeutet, das Ubuntu auf demselben Datenträger wie Windows installiert und der Windows-Bootloader vom Ubuntu-Bootloader überschrieben wird, wodurch dieser bei jedem Start des Rechners ausgeführt wird und man dann auswählen muss, ob Ubuntu oder Windows gestartet werden soll.

Da in dieser Arbeit Ubuntu auf dem zweiten Datenträger installiert wurde, wird der dritte Punkt ausgewählt. Dieser kann aber auch immer ausgewählt werden, wenn man bereits Partitionen vorbereitet hat.

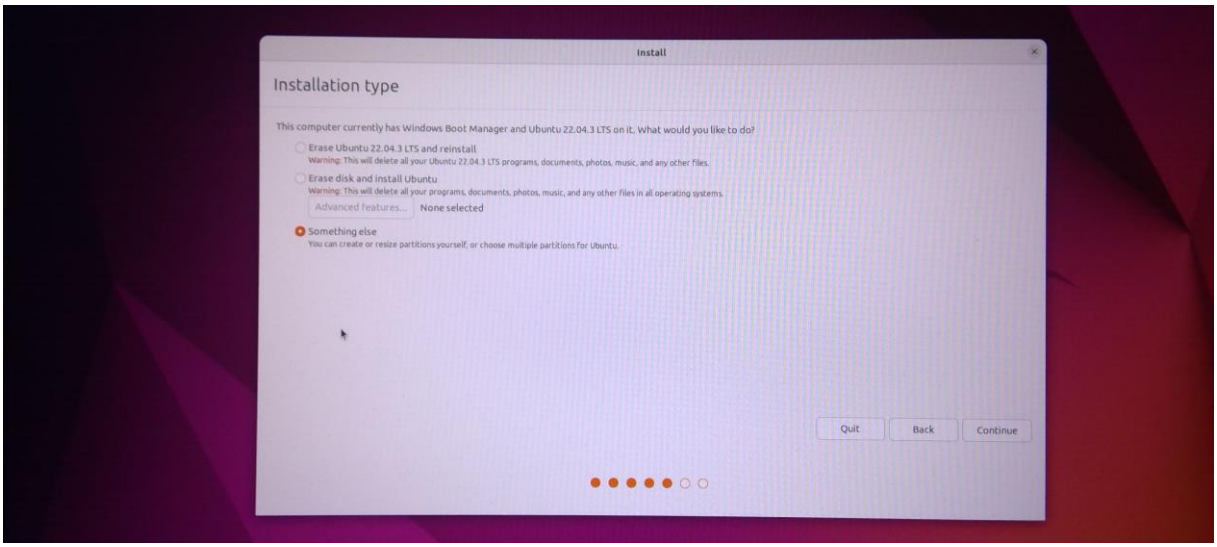


Abbildung 10 Auswahloberfläche für den Installationsort des Ubuntu-Systems des Ubuntu-Installationsmanager. Eigener Ausschnitt

Nach der Bestätigung der Aktion wird die Partitionsübersicht geöffnet.

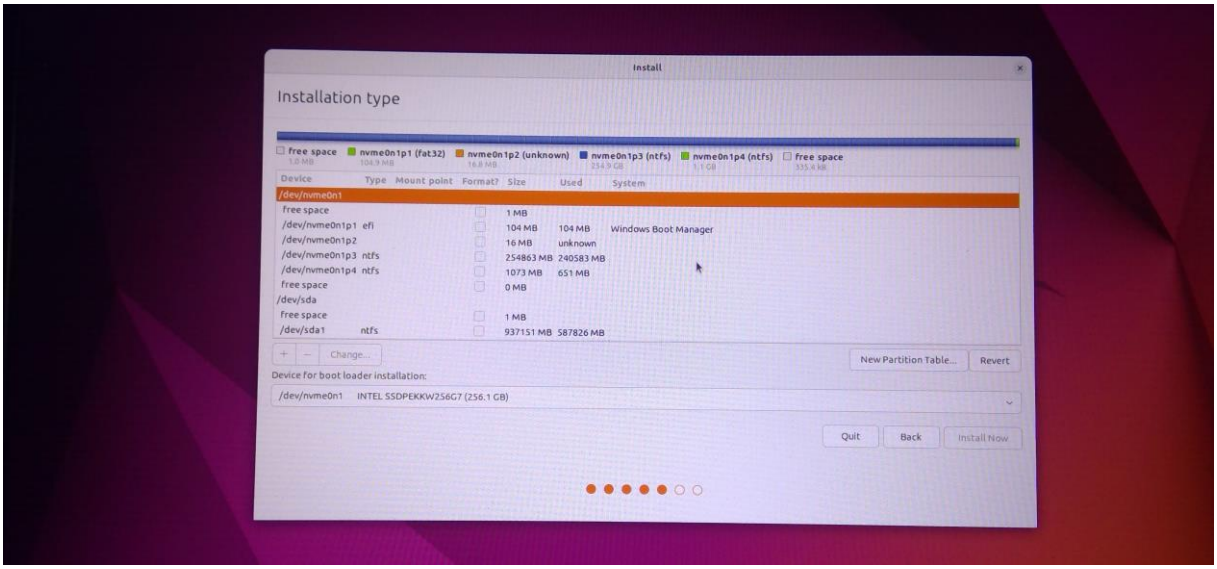


Abbildung 11 Partitionsübersichtsfenster des Ubuntu-Installationsmanager. Eigener Ausschnitt

Aktuell ist noch der erste Datenträger `/dev/nvme0n1` zur Bootloader-Installation ausgewählt. Wie zu sehen ist, entspricht dies dem Laufwerk `C` auf dem auch Windows liegt, was bedeutet, dass der Windows-Bootloader überschrieben werden würde.

Da in dieser Arbeit der Windows-Bootloader als Standard beibehalten werden soll, sodass immer Windows startet und der Ubuntu-Bootloader bei einem Windows-Neustart durch das Windows-Boot-Menü mittels der F12 Taste beim Neustart ausgeführt werden kann, wird hier der Ubuntu-Bootloader auf dem zweiten Datenträger `dev/sda` geschrieben, auf dem auch Ubuntu installiert wird. Dies entspräche dem Laufwerk `D`.

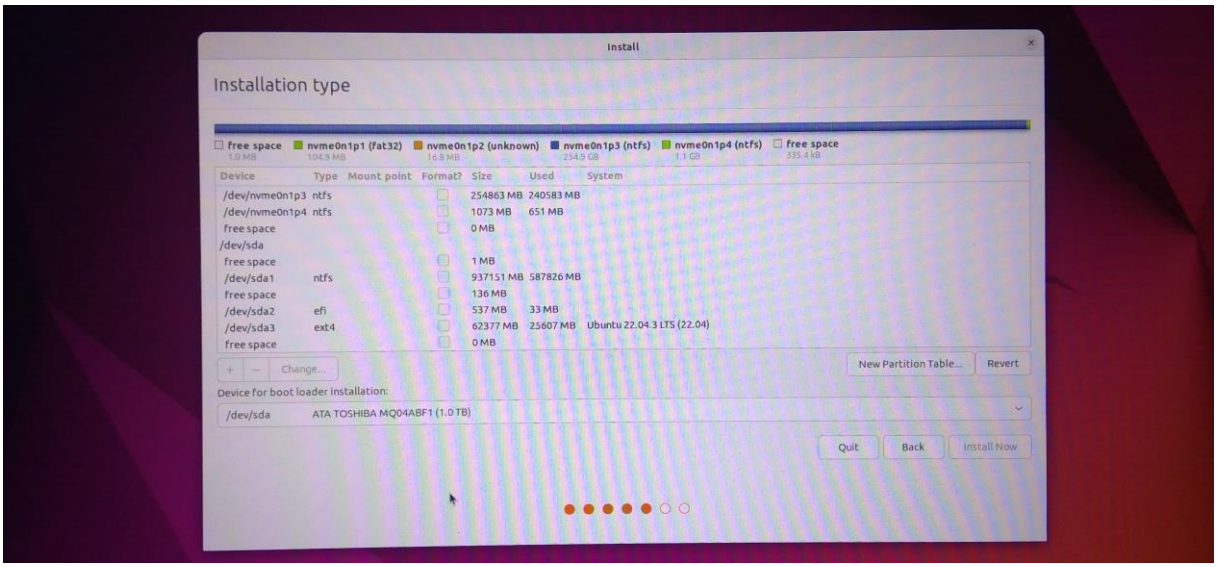


Abbildung 12 Auswahldarstellung des Installationslaufwerks D in der Partitionsübersicht. Eigener Ausschnitt

Um jetzt Ubuntu zu installieren, muss noch der zuvor freigegebene Platz unter `dev/sda` gefunden werden. Die 130 MB werden hier mit 136 MB angegeben, da noch 6 MB freier Speicher über den 130 MB existierten. Die Formatierung des freien Platzes wird dann mit einem Doppelklick gestartet.

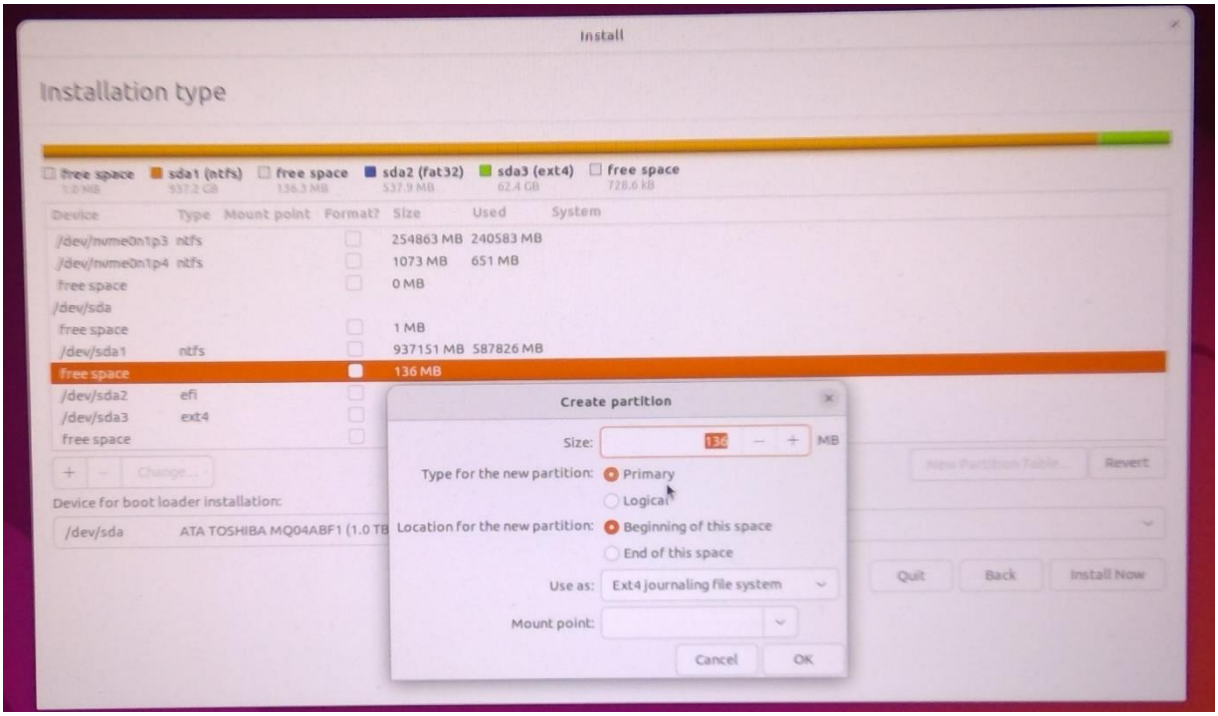


Abbildung 13 Größenreservierung und Formatierung der ausgewählten Installationspartition. Eigener Ausschnitt

Damit die Partition kompatibel mit Ubuntu ist, muss diese in einem `ext` Format formatiert werden. Hierfür kann der `ext4` Stil ausgewählt werden. Um die Formatierung abzuschließen, wird mit OK bestätigt. Nach der erfolgreichen Formatierung kann dann die Installation vorgenommen und den restlichen Anweisungen Folge geleistet werden.

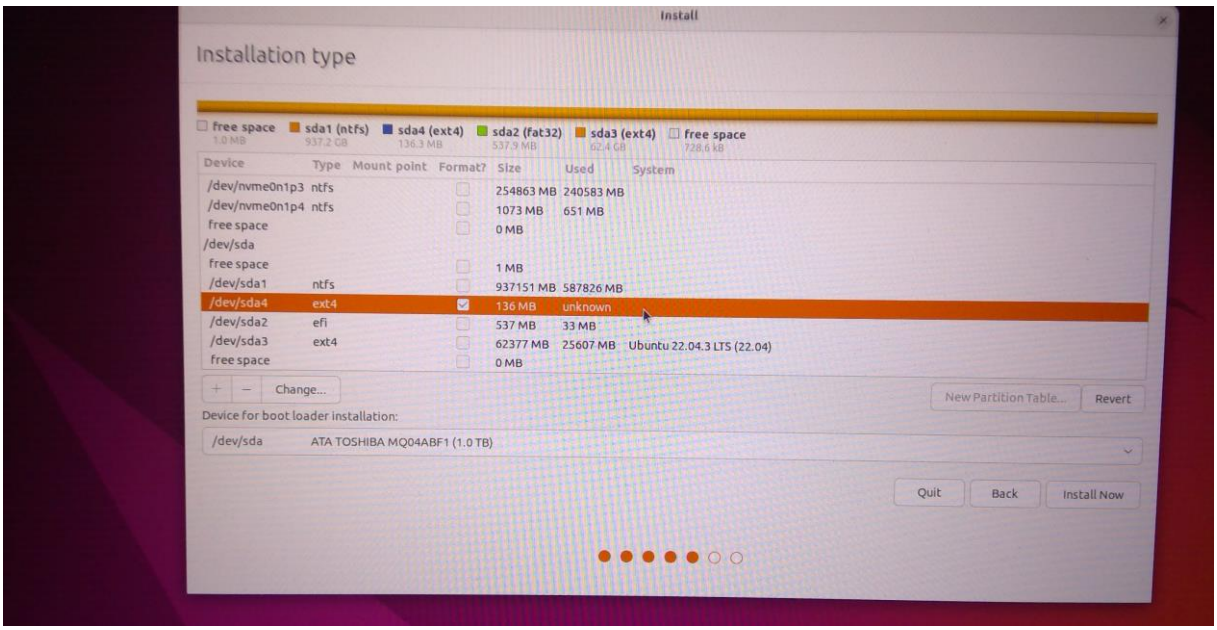


Abbildung 14 Eigene Darstellung der reservierten und Partitionierten Installationspartition

Nachdem alles installiert ist, muss ein Neustart durchgeführt werden. Hierbei muss der USB-Stick bei Aufforderung entnommen werden.

1.1 Probleme und Zugriff auf Daten der Partition über Windows

Sollte beim Wechseln ins Dual-Boot mittels der `F12` Taste Probleme auftreten, so muss in den Boot-Manager mittels `uefi` gewechselt werden und dieser ohne Änderungen wieder verlassen werden. Beim danach auftreten Neustart des Gerätes kann dann mittels der `F12` Taste ins Dual-Boot gewechselt und Ubuntu gestartet werden.

Sollte das Booten in Ubuntu nicht mehr möglich sein, so kann mittels der folgenden Software DiskInternals Linux Reader[2] auf die Dateien der Ubuntu-Partition über Windows zugegriffen werden.

Sollte ein GNU-Terminal nach dem Wechsel mit der F12 Taste erscheinen und dieses nicht mehr mittels dem Ausschalter des Computers verlassen werden können, so müssen die folgenden Befehle eingegeben werden, um in Windows zu booten:

```
insmod part_gpt;insmod chain;set root=(hd0,gpt1);chainloader /EFI/Microsoft/Boot/bootmgfw.efi;boot
```

Hierbei kann jedoch insbesondere die Auswahl des root je nach System abweichen, weshalb es sinnvoll ist die folgende Seite von GNU aufgesucht werden[3].

1.2 Boot-Manager-Anforderungen

Unter den Systeminformationen sollte das Haupt-Betriebssystem Windows den BIOS-Modus UEFI unterstützen.

1.3 Vor dem Einrichten

Vor dem Einrichten des Dual-Boot sollte unbedingt eine Sicherung des bestehenden Systems durchgeführt werden.

[2] https://www.diskinternals.com/linux-reader/

[3]ttps://www.gnu.org/software/grub/manual/grub/html_node/Chain_002dloading.html#Chain_002dloading

2 Einrichtung aller notwendigen Komponenten

Nachdem das Betriebssystem eingerichtet wurde, kann nun mit der Einrichtung aller für die Entwicklung notwendiger Komponenten begonnen werden. Diese und deren Installation werden in den folgenden Unterabschnitten erklärt.

2.1 Installation der ROS-Pakete

Um ROS2 einzurichten, muss zunächst Ubuntu gestartet werden. Anschließend muss das Systemterminal geöffnet werden. Daraufhin können die Anweisungen von der ROS2 Humble Website[4] befolgt werden. Nachdem der Einrichtungsteil abgeschlossen wurde, muss bei der Auswahl des ROS2 Pakets `sudo apt install ros-humble-desktop` ausgewählt und in das Terminal eingegeben werden. Hierbei werden alle wichtigen Inhalte von ROS installiert, welche auch wichtig für die spätere Simulation sind.

Nachdem alles eingerichtet ist, muss zuletzt ROS zu der Systemkonsolenumgebung hinzugefügt werden. Dies ist notwendig, damit ein ROS-Befehl jederzeit ausgeführt werden kann, ohne im Vorhinein erneut die ROS-Auszuführende zum Terminal hinzufügen zu müssen. Dies verhindert unnötige Fehler, die entstehen, wenn dieser „Source-Schritt“ vergessen wird.

Hierfür muss in das Systemterminal folgendes eingegeben werden: `gedit ~/.bashrc`. Dieser Befehl öffnet die Textdatei, welche beim Start des Terminals ausgeführt wird. Hierin sind alle Aktionen aufgeführt, welche das Terminal ausführen soll. Hierzu gehört bspw. auch das Anlegen von Pfaden zu auszuführenden Dateien wie Programme und ähnliches. Damit das System die ROS-Auszuführende finden kann und sich somit auch die spätere Entwicklungsumgebung richtig initialisiert, muss an das Ende der Datei der Befehl: `source /opt/ros/humble/setup.bash` eingefügt werden.

Um die Änderungen zu übernehmen, muss nach dem Speichern und Schließen der Datei der Befehl: `source ~/.bashrc` im Terminal ausgeführt werden.

2.2 Installation und Einrichtung der Entwicklungsumgebung

Nachdem ROS vollends eingerichtet wurde, benötigt es noch eine Entwicklungsumgebung. Hierfür muss Visual Studio Code aus dem App-Store installiert werden. Da die Entwicklung mittel C++ geschehen soll, müssen die notwendigen C++ Erweiterungen von Microsoft heruntergeladen werden. Des Weiteren bedarf es noch die ROS-Erweiterung von Microsoft für Visual Studio Code.

[4] https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debians.html

Einrichtung aller notwendigen Komponenten

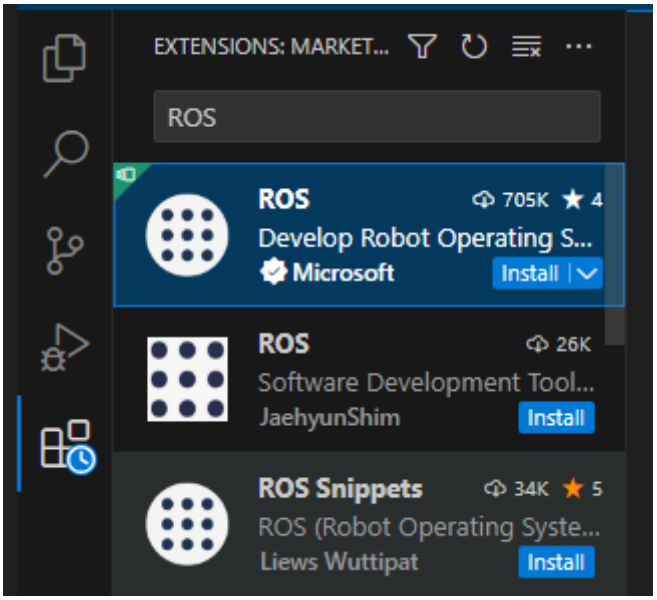


Abbildung 15 Eigener Ausschnitt der ROS-Erweiterung in VSCode

Als letzte Erweiterung muss noch CMake hinzugefügt werden.

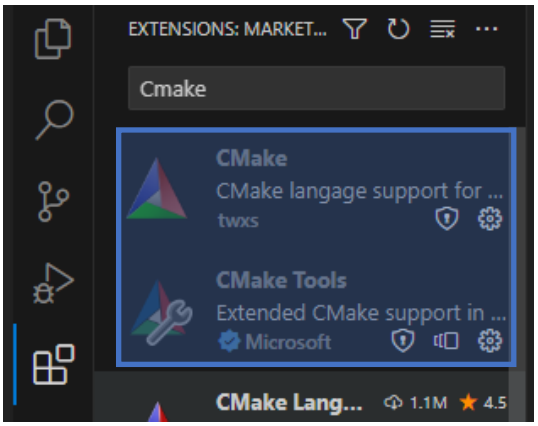


Abbildung 16 Eigener Ausschnitt der CMake-Erweiterung in VSCode

Nachdem dies durchgeführt wurde, ist Visual Studio vollends für die Verwendung von C++ und ROS vorbereitet. Sicherheitshalber kann auch das Python Paket hinzugefügt werden.

2.3 Tools zum Erstellen eines ROS-Projekts

Um ein ROS-Projekt zu starten, muss als letztes noch Python und Colcon eingerichtet werden. Hierfür müssen folgende Befehle in das Terminal eingegeben werden:

- `sudo apt install python3`
- `sudo apt install python3-colcon-common-extensions`

Da Colcon auf Python basiert, muss dies zunächst eingebunden werden. Colcon ist hierbei das Tool, welches ein ROS-Paket baut und die Auszuführenden erstellt.

2.4 Erklärung der Visual Studio Code Entwicklungsumgebung

Nachdem die Einrichtung der Entwicklungsumgebung abgeschlossen wurde, wird in diesem Abschnitt die Entwicklungsoberfläche von Visual Studio Code erläutert. Hierbei wird auf die wichtigsten Ordner und Dateien eingegangen, als auch deren Bedeutung erklärt. Die Oberfläche wird im nachfolgenden Bild dargestellt.

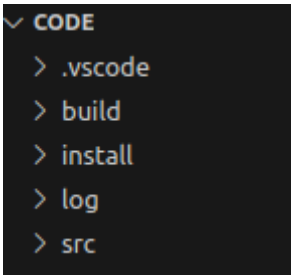


Abbildung 17 Eigener Ausschnitt des ROS-Projetordners in VSCode

Die wichtigsten Ordne sind:

- .vscode,
- install,
- src.

Funktion des .vscode Ordners

Der .vscode Ordner ist für Visual Studio Code selbst wichtig.

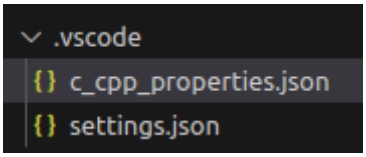


Abbildung 18 Eigener Ausschnitt des .vscode Ordners in VSCode

Hierbei wird dieser Ordner automatisch von Visual Studio Code nach dem Öffnen des Projektes angelegt. In der `settings.json` werden alle C++ relevanten Bibliotheken aufgelistet, während in der `c_cpp_properties.json` der C und C++ Standard, als auch die Include-Pfade für die Headers festgelegt werden. Zudem können weitere Präprozessorargumente übergeben werden.

```
{
  "configurations": [
    {
      "browse": {
        "databaseFilename": "${default}",
        "limitSymbolsToIncludedHeaders": false
      },
      "includePath": [
        "/opt/ros/humble/include/**",
        "/home/benjamin/Desktop/svn/Bachelor/Code/src/test_pkg/include/**",
        "/usr/include/**"
      ],
```

```
      "name": "ROS",
      "intelliSenseMode": "gcc-x64",
      "compilerPath": "/usr/bin/gcc",
      "cStandard": "gnu11",
      "cppStandard": "c++23",
      "compilerArgs": [
        "-fopenmp"
      ]
    }
  ],
  "version": 4
}
```

Hierbei muss nur die `c_cpp_properties.json` angepasst werden. Eine wichtige Anmerkung ist, dass diese Datei keinerlei Einfluss auf den Bauprozess des ROS-Paketes mittels Colcon hat. Sie dient alleine dazu, dass Visual Studio Code weiß, wo es nach Headern suchen soll, welche Definitionen und welcher C++ Standard verwendet wird und dementsprechend im Code keine Fehler anzeigt, weil es den Header oder die C++ Funktion nicht kennt oder findet.

2.4.1 Funktion des install Ordners

Der Installationsordner enthält das mit Colcon gebaute ROS-Paket, sozusagen die .exe in Form einer Setup-Datei.

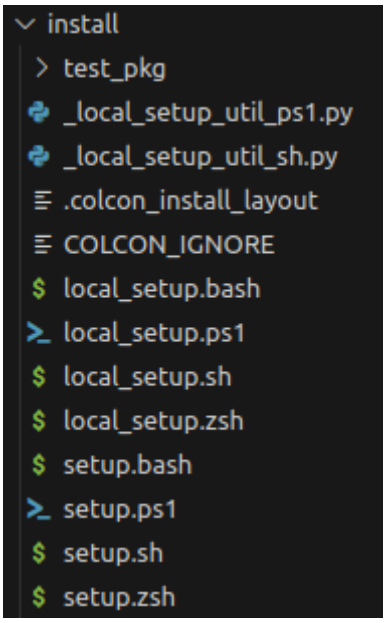


Abbildung 19 Eigener Ausschnitt des Installationsordners in VSCode

Damit ROS später das selbsterstellte Paket aufrufen und einer der darin enthaltenen Nodes ausführen kann, muss eine `local_setup` den Umgebungsvariablen als Pfad hinzugefügt werden. Welche dies ist, hängt von dem verwendeten Systemterminal ab.

In diesem Fall ist es die Endung `.bash`.

Das hinzufügen geschieht wie bereits beschrieben:

1) Öffnen eines Terminals,
2) Eingabe von `gedit ~/.bashrc`,
3) Zufügen von `source /Pfad_zu_ROS_Workspace/install/local_setup.bash`,
4) Speichern und Schließen der Textdatei.

Um die Änderung zu übernehmen, muss einmalig `source ~/.bashrc` eingegeben werden. Dies muss jedes Mal vorgenommen werden, wenn das ROS-Paket mittels Colcon neu gebaut wird.

Soll die `.bash` nicht dauerhaft verlinkt sein, reicht es aus, nach jedem Bauen des Projektes den `source`-Befehl mit dem Pfad in das Terminal einzugeben und auszuführen.

2.4.2 Bedeutung des src Ordners

Der Quellenordner beinhaltet zum einen die lokalen Header-Dateien, die CMakeList, die XML als auch die Nodes des Projekts.

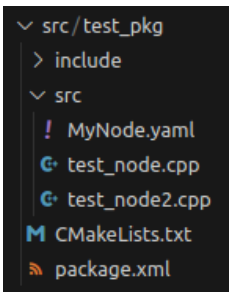


Abbildung 20 Eigener Ausschnitt des Quellenordners in VSCode

Die Nodes sind hierbei als einzelne .exe Dateien zu verstehen. Ein Paket kann mehrere dieser Auszuführenden besitzen, welche alle spezifische Aufgaben ausführen und untereinander kommunizieren können.

Die `.yaml` Datei ist die Speicherform der Node-Parameter, während die `CMakeLists.txt` alle relevanten Informationen beinhaltet, die für das Kompilieren des Paketes notwendig sind.

Die package.xml Datei enthält die Informationen für Colcon, welches Buildtool verwendet wird, also ob C++ oder Python verwendet wird. Des Weiteren enthält es Informationen über Abhängigkeiten des aktuellen Projekts, sowie Informationen über Copyright und den Entwickler.

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>test_pkg</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="User@todo.todo">benjamin</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>
```

3 Bauen eines ROS-Paketes mit Colcon und Ausführen einer ROS-Node

Um ein ROS-Paket zu erstellen, muss zunächst ein Arbeitsordner mit einem Unterordner namens `src` erstellt werden. Im Anschluss muss mit dem Terminal in den übergeordneten Ordner navigiert und der Befehl `colcon build` ausgeführt werden.

Um das Paket zu erstellen, muss in den Quellenordner navigiert und der folgende Befehl ausgeführt werden:

```
ros2 pkg create --build-type ament_cmake <package_name>
```

Wobei `ament_cmake` angibt, dass CMake also C++ verwendet wird und `<package_name>` den Namen des Paketes beinhaltet, der nach dem Bauen durch ROS verwendet wird, um auf die Nodes zuzugreifen.

Als nächstes muss der C und C++-Standard in .vscode festgelegt werden.

Sollen andere ROS-Pakete oder auf dem System installierte Bibliotheken, also Funktionen aus denen verwendet werden, müssen diese der XML- und der CMake-Datei hinzugefügt werden.

Hierfür muss der XML-Datei die Basis-ROS-Bibliothek hinzugefügt werden, wofür folgendes eingegeben werden muss:

```
  <depend>rclcpp</depend>
```

Hängt das Paket von mehreren Bibliotheken ab, so können diese einfach nach dem gleichen Schema hinzugefügt werden:

```
  <depend>std_msgs</depend>
  <depend>tf2</depend>
```

Dies sollte nach der Deklaration des Build-Tools geschehen.

```
 <buildtool_depend>ament_cmake</buildtool_depend>
```

Als Nächstes sollte, überprüft werden, ob die folgenden Code-Zeilen am Ende der Datei vorhanden sind:

```
<test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

Damit das Paket erstellt werden kann, muss die CMake-Datei wie folgt angepasst werden. Hierfür muss zunächst die CMake-Version als auch der verwendete Compiler festgelegt werden.

```
cmake_minimum_required(VERSION 3.8)
project(test_pkg)

#Set specific Compiler options
if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic -fopenmp)
endif()
```

Als nächster Schritt werden die Pakete, also Bibliotheken definiert, von denen das Projekt abhängt. Hierbei sind die nachfolgenden grundlegend relevant:

```
find_package(ament_cmake REQUIRED)
find_package(ament_cmake REQUIRED)
```

Hängt das Projekt von mehreren Paketen ab, können diese einfach angehängt werden:

```
find_package(std_msgs REQUIRED)
find_package(tf2 REQUIRED)
```

Im Anschluss müssen die Auszuführenden, also die Nodes angegeben werden. Da es in diesem Beispiel zwei Nodes gibt, sieht dies wie folgt aus:

```
add_executable(test_node src/test_node.cpp)
add_executable(test_node2 src/test_node2.cpp)
```

Im Anschluss können Shortcuts für die Sprachstandards, die Include-Pfade als auch für die Abhängigkeiten, also den Bibliotheken und der Nodes erstellt werden. Dies vereinfacht die darauffolgenden Prozesse, insbesondere wenn mehrere Nodes in einem Paket enthalten sind.

```
#Set shortcuts (Defines for later use as variable)
set(LanguageSTD c_std_11 cxx_std_23) #Language standart
set(AmentDependencies "rclcpp" "std_msgs" "tf2") #Pachages to include for build
set(TargetIncludes $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include> $<IN-
STALL_INTERFACE:include>) #Inlcude path for headers
set(MyTargets test_node test_node2) #Targetlist for easier install by in-
stall_targets(<targets> <destination>)
```

Nachdem die Shortcuts festgelegt sind, muss CMake noch mitgeteilt werden, wo es für die Headers suchen soll. Dies entspricht dem Prozess, der auch für .vscode nötig ist und kann durch die folgenden Befehle geschehen und muss für jede Node gemacht werden:

```
#Setting Include paths (Headers) for each target executable
target_include_directories(test_node PUBLIC ${TargetIncludes})
target_include_directories(test_node2 PUBLIC ${TargetIncludes})
```

Wie zu sehen ist, wird hierbei der `TargetIncludes` als Shortcut verwendet. Das Dollarzeichen und die geschweiften Klammern sagen hierbei, dass es sich bei dem Ausdruck zwischen den

Klammern um eine Variable, also eine Definition handelt, welche weiterführende Informationen beinhaltet.

Nachdem die Abhängigkeiten, die Include-Pfade und die Auszuführende bekannt gegeben und festgelegt wurden, muss jetzt noch der Compiler initialisiert werden. Hierfür muss dieser Schritt wieder für jede Auszuführende, also für jede Node vorgenommen werden:

```
#Setting Compiler features for each target executable
target_compile_features(test_node PUBLIC ${LanguageSTD})
ament_target_dependencies(
  test_node
  ${AmentDependencies}
)
target_compile_features(test_node2 PUBLIC ${LanguageSTD})
ament_target_dependencies(
  test_node2
  ${AmentDependencies}
)
```

Wie zu sehen ist, wird hierbei dem Compiler mitgeteilt, welche Auszuführende er erstellen soll, welcher Sprachstandard für diese gilt (s. Shortcut `LanguageSTD`) und welche Abhängigkeiten (s. Shortcut `AmentDependencies`), also Bibliotheken, diese verwendet. Hierbei hängt jede Auszuführende im Ersteren natürlich von sich selbst ab.

Die eigentliche Installation bzw. Erstellung der Auszuführenden wird durch den folgenden Ausdruck ermöglicht:

```
install(TARGETS ${MyTargets}
  DESTINATION lib/${PROJECT_NAME})
```

Wie zu sehen ist, kommt auch hier wieder ein Shortcut zu tragen. Hierbei ermöglicht `MyTargets` beide Auszuführenden, also Nodes, in nur einem Befehl zu erstellen, da er beide beinhaltet. Wäre dies nicht der Fall, müsste der Ausdruck wie bereits beim Compiler für beide separat aufgeschrieben werden. Als Letztes muss noch folgendes geschrieben werden:

```
if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  # the following line skips the linter which checks for copyrights
  # comment the line when a copyright and license is added to all source files
  set(ament_cmake_copyright_FOUND TRUE)
  # the following line skips cpplint (only works in a git repo)
  # comment the line when this package is in a git repo and when
  # a copyright and license is added to all source files
  set(ament_cmake_cpplint_FOUND TRUE)
  ament_lint_auto_find_test_dependencies()
endif()

ament_package()
```

Dies sollte jedoch nach dem Erstellen des Paketes mittels ROS vorhanden sein.

Sind die Nodes angelegt worden und das Projekt soll erstellt werden, so muss in den Hauptordner navigiert werden. Durch das Ausführen des Befehl `colcon build` werden alle Pakete, die sich in dem Ordner befinden, erstellt. Man kann jedoch auch angeben, welche Pakete erstellt und welche nicht erstellt werden sollen.

Nach dem erstellen muss erneut `source ~/.bashrc` ausgeführt werden und um dann eine Node auszuführen muss folgendes in das Terminal eingegeben werden:

```
ros2 run test_pkg test_node
```

Hiermit wird ROS gesagt, dass es die Node mit dem Namen `test_node` aus unserem Paket mit dem Namen `test_pkg` ausführen soll.

Ist ROS richtig eingerichtet, sollte die Fußleiste von Visual Studio Code auf der linken Seite die ROS-Version und auf der rechten Seite den Marker ROS neben C++ aufführen. Dies bedeutet, dass die ROS-Erweiterung von Visual Studio Code erfolgreich die installierte ROS-Version gefunden hat und dessen Funktionen unterstützt.

4 ROS-Kurzeinführung

Die Informationen für dieses Kapitel, und weiterführende Informationen zu Nodes und was mit ihnen zusammenhängt, wie die Parametrisierung, können den ROS-Tutorials (19) entnommen werden.

ROS teilt die Aufgaben eines autonomen Systems in viele kleine Teilaufgaben auf. Diese werden dann auf die Nodes verlagert, was bedeutet, dass jede Node für eine ganz spezielle Aufgabe, wie dem Auswerten von Sensordaten, Navigieren usw. zuständig ist.

Abbildung 21 Eigene Darstellung der Nodes

Jeder dieser Nodes läuft hierbei als separates Programm. Dies hat den Vorteil, dass Prozesse „echt“ parallel ablaufen können, sodass bei vier Kernen, vier Nodes parallel arbeiten können und gleichzeitig den Vorteil der ausführenden Kerne, also Threads nutzen können. Dies ist vorteilhafter, als wenn das gesamte Programm unter einem Kern läuft und alle ausführenden Kerne, Threads, nutzen würde. In diesem Falle würde der Prozessor nicht vollends genutzt werden und andere Programme könnten Rechenzeit in Anspruch nehmen, die dem aktuellen Programm entzogen wird. Damit die Nodes Daten austauschen können, verfolgt ROS zwei Funktionsprinzipien:

1) Topics,
2) Services
3) Und Actions.

Die Kommunikation mittels Topics verläuft nach dem Abonnentenprinzip. Es kann bspw. eingesetzt werden, wenn eine Node zyklische Werte von bspw. einem Sensor benötigt. In diesem Fall bietet die Sensor-Node ein Abonnement an, indem sie alle 10 Millisekunden die neuen Sensorwerte veröffentlicht. Alle Nodes, die an diesen Werten interessiert sind, können dieses Abo abschließen und werden alle 10 Millisekunden über das Eintreffen der neuen Werte informiert und können die bis dato veröffentlichten Werte auslesen.

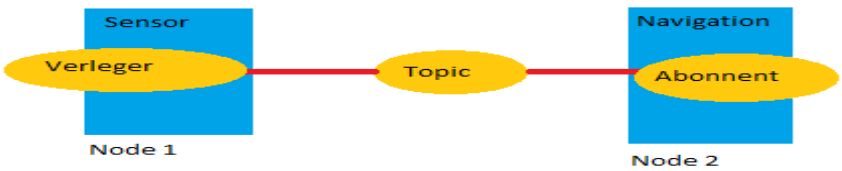


Abbildung 22 Eigene Darstellung der Topic-Kommunikation

Die Service-Kommunikation funktioniert hierbei nach dem „Internetprinzip“, das bedeutet, dass eine Node eine Anfrage an eine andere Node stellt und damit aktiv eine Aktion mit Ergebnis von dieser anfordert. Die angefragte Node führt dann die auf die Anfrage passende Reaktion aus, und sendet eine Antwort zurück. Diese kann entweder eine Bestätigung für das Ausführen der Aktion sein oder auch das Ergebnis, welches durch diese generiert wurde. Hierbei läuft die Anfrage, deren Weiterleitung und die Rückleitung der Antwort über die ROS-Service-Node, bei der alle allen aktiven Nodes registriert sind. Diese wird automatisch mitgestartet, sobald ROS angewiesen wird, eine beliebige Node auszuführen.

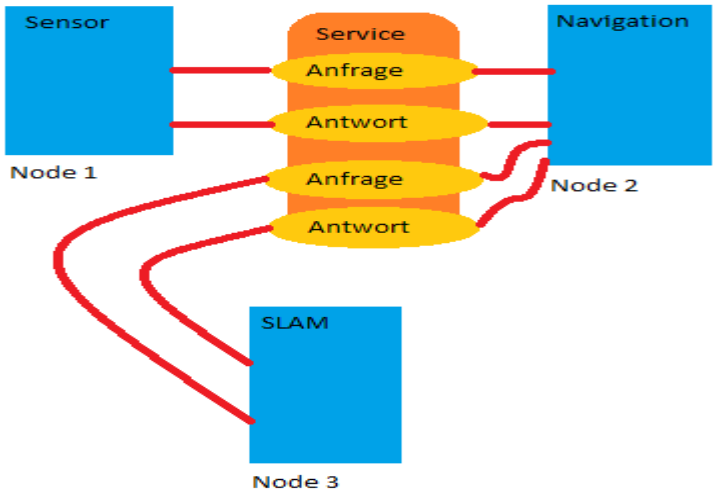


Abbildung 23 Eigene Darstellung der Service-Kommunikation

Die Kommunikation mittels Actions basiert auf den bereits genannten Methoden. Hierbei wird diese Art eingesetzt, wenn eine Aufgabe sehr zeitintensiv ist und abgebrochen werden kann, wenn sie zu lange braucht. Hierbei stellt eine Node bspw. eine Anfrage für die aktuelle Position des Roboters an die SLAM-Node.

Diese gibt zunächst eine Antwort, dass die Schätzungsberechnung gestartet wird. Anschließend sendet sie in regelmäßigen Abständen den Fortschritt über ein Topic an die anfragende Node.

Die betreffende Node kann prüfen, ob die Zeit zwischen zwei Feedbacks zu lange gedauert hat, wodurch die SLAM-Node den vorgegebenen zeitlichen Rahmen sprengen könnte, indem das Ergebnis vorliegen sollte.

Wenn dies registriert wird, sendet die fragende Node eine Beendigungsanfrage an die SLAM-Node, welche dann ihre Aktion beendet, und eine Bestätigung zurückschickt. Kann die SLAM-Node ihre Aufgabe in der Zeit erledigen, so sendet sie das Ergebnis zurück und beendet das Feedback für diese Anfrage.

Hierbei können mehrere Anfragen gleichzeitig vorliegen, die dann seriell abgearbeitet werden und immer nur die Node eine Behandlungsbestätigung bekommt, deren Anfrage aktuell ausgeführt wird.

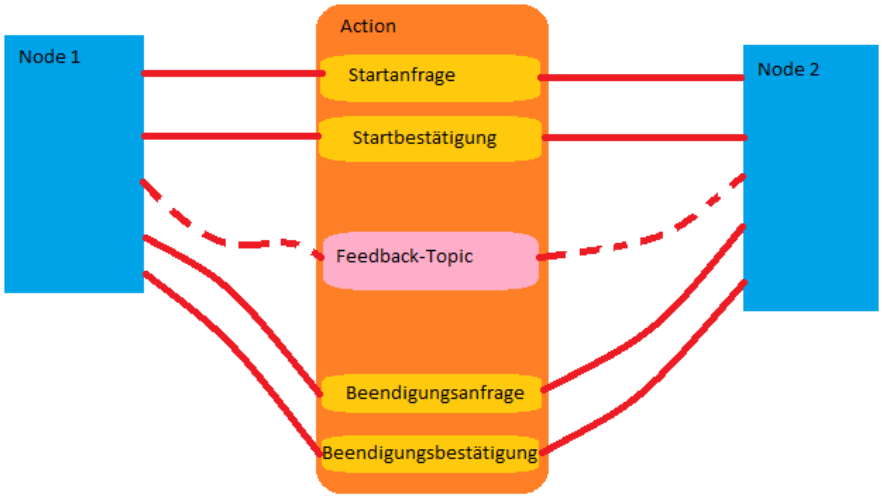


Abbildung 24 Eigene Darstellung der Action-Kommunikation

5 CMake-Einführung

In diesem Abschnitt wird genauer auf die Verwendung von CMake im Zusammenhang mit Visual Studio Code, C++ und ROS2 eingegangen.

5.1 Erstellen eines CMake Projektes

Jedes CMake Projekt besteht aus mindestens einer CMakeLists.txt Datei. **Diese muss immer diesen Namen tragen**. Um eine CMakeLists.txt in mehrere kleinere aufzuteilen, welche jeweils eine bestimmte Aufgabe des Build-Prozesses bearbeiten, müssen diese dann in den jeweiligen Ordnern abgelegt werden.

💡 **Bsp.** Das Projekt besteht aus einer Main-Datei. Diese bindet diverse Header mit Funktionsdeklarationen. Die Definitionen der Funktionen wurden in dazugehörigen .cpp Dateien vorgenommen. Damit jetzt nicht die .cpp eingebunden werden muss, sondern ganz normal die .hpp's müssen für die .cpp's eine jeweilige temporäre Bibliothek erstellt werden, welche dann an die Main-Datei gelinkt werden.

💡 Dieser Schritt ist notwendig, da der Compiler ansonsten die Funktionsdefinitionen nicht finden würde, da die entsprechenden .cpp's ja zu keinem Zeitpunkt mittels #include<Funktionsdefinition.cpp> einbezogen wurden.

💡 Wir müssen also die Aufgabe des Linkers übernehmen, welcher dies bspw. bei Visual Studio Community automatisch macht.

Da in diesem Beispiel die Haupt-CMakeLists.txt nur die Main.cpp erstellen und die wichtigsten Compile-Optionen setzen soll, wollen wir das Erstellen der temporären Bibliotheken in jeweils separate CMake-Files aufteilen, welche dann später in die Haupt-CMake eingebunden werden. Dies ist zu vergleichen mit dem #include<something> Befehl von C++.

5.2 CMake If-Abfrage

```
if(something)
  #do somethng
endif()
```

5.3 CMake Variablen

In CMake können wie auch bei C++ Variablen angelegt werden, welche einen bestimmten Wert besitzen. Die Variablen sind hierbei jedoch eher als Makros zu verstehen. Diesbezüglich werden sie auch in Großbuchstaben und mit Unterstrich getrennt geschrieben wie in C++. Bei der Verwendung des Unterstriches sollte jedoch darauf geachtet werden, dass man nicht aus Versehen eine bestehende CMake-Variable überschreibt.

Sie können verwendet werden, um in `If`-Abfragen Wahrheiten zu prüfen oder plattformspezifische Eigenschaften zu setzen, wie zu verwendende Compiler, Include-Pfade und ähnliches. Sie können auch dafür verwendet, um einen häufig verwendeten Verweis einmalig zu definieren oder mehrere Parameter zu einer Variablen zu bündeln.

```
set(MY_DIRECTORY_PATH /path/directory_name)
set(MY_ARGUMENT_LIST arg1 arg2 arg3 ... argn)
set(MY_RENAMED_EXECUTABLE_OBJECT executable_object)
```

Um eine definierte Variable zu “löschen” kann man das Folgende tun:

```
unset(MY_DIRECTORY_PATH)
```

💡 Info: Wenn ein Target-Objekt mit einer Variablen erstellt wurde, die den Namen dieses Objektes enthält, mittels den Funktionen `add_executable` und `add_library` und die Variable wird danach gelöscht, also `unset`, so bleibt der Objektname erhalten, das Objekt wird also nicht gelöscht.

Um die Variable wieder zu verwenden, muss erneut `set()` aufgerufen werden.

Um auf den Wert der Variablen zugreifen zu können, muss diese folgendermaßen aufgerufen werden:

```
${MY_DIRECTORY_PATH}
```

Die Rückgabe ist dann: `/path/my_directory_name`

Um den Pfad zu erweitern, damit bspw. auf untergeordnete Ordner zugegriffen werden kann, kann eine neue Pfadvariable angelegt werden:

```
set(NEW_PATH ${MY_DIRECTORY_PATH}/additional_path)
```

Danach würde die Variable `NEW_PATH` den folgenden Pfad enthalten:
`/path/my_directory_name/additional_path`

Um zu überprüfen, ob eine Variable existiert, kann bspw. folgende `If`-Abfrage durchgeführt werden:

```
if(MY_VARIABLE)
...
endif()
```

Um zu prüfen, ob ein Pfad existiert, kann dies mittels der folgenden Zusatzkommandos abgefragt werden:

```
if(EXISTS MY_PATH)
...
endif()

if(NOT EXISTS MY_PATH)
...
endif()
```

5.4 CMake Pfade

Um Pfadangeben in CMake machen zu können, gibt es drei elementare Befehle:

1. ${CMAKE_CURRENT_SOURCE_DIR}

Dieser Befehl gibt den Pfad zum Ordner zurück, in dem sich die CMakeLists.txt befindet.

2. get_filename_component(MY_NEW_PATH ../ ABSOLUTE)

Dieser Befehl kann genutzt werden, wenn man ausgehend von seinem aktuellen Quellenordner (`${CMAKE_CURRENT_SOURCE_DIR}`) in darüberliegende Ordnerstrukturen navigieren möchte. In diesem Beispiel gehen wir von dem aktuellen `${CMAKE_CURRENT_SOURCE_DIR}` - Pfad eine Ordnerstufe zurück und bekommen für diesen Ordner den absoluten Pfad in der Variablen `MY_NEW_PATH` zurück. Es kann auch ein relativer Pfad erzeugt werden (`RELATIVE`).

3. ${CMAKE_CURRENT_LIST_DIR}

Dieser Ausdruck gibt den Pfad zum Ordner der CMakeList zurück, welche gerade bearbeitet wird.

D.h. wenn mehrere CMake-Files in eine CMakeList eingebunden werden (`include`) wird dann nicht mehr der Pfad zum Ordner der Einbindenden, sondern der Pfad zu der CMakeList ausgegeben, welche gerade in diesem Moment eingebunden wird. Hierbei macht die Verwendung dieser Pfadvariable auch nur in diesen CMakeLists Sinn, wenn der explizit der Pfad zu dieser CMake gelesen werden soll und nicht der der Einbindenden.

5.5 Zusammenführen von CMakeLists.txt

Wie im obigen Beispiel angesprochen kann es nützlich sein eine CMakeLists.txt in Einzelne aufzuteilen. Um diese dann wieder zu einer einzigen zusammenzuführen muss der folgende Befehl ausgeführt werden:

```
include(path_to_target_CMakeLists_Folder/CMakeLists.txt)
```

Es muss also der Pfad zu dem Ordner der einzubindenden CMakeLists und dieselbige am Ende des Pfades angegeben werden.

 Der `include` -Befehl bewirkt hierbei, dass der Code, der in der einzubindenden CMakeLists steht an der Stelle des `includes` in die einbindende CMakelists.txt eingefügt wird. Dies bewirkt also, dass alle in der einzubindenden definierten Variablen und Objekte unter dem gleichen Namen nach dem `include` in der Einbindenden aufgerufen werden können.

💡 Dies bedeutet auch dass alle Variablen und Objekte aus CMakeLists.txt, welche vorher bereits eingebunden wurden auch in der danach eingebundenen CMakelists.txt aufrufbar/erreichbar oder abfragbar, bspw. mit `If-EXISTS` dann tue das, sind.

💡 Zudem sind ab dem Zeitpunkt wo eine CMakeLists.txt in einer anderen mittels `include` eingebunden wird, alle Pfadangaben nicht mehr die für den Ordner, indem sich die Einzubindende befindet, sondern die für den Ordner in dem sich die einbindende CMakeLists.txt befindet. D.h., dass `${CMAKE_CURRENT_SOURCE_DIR}` den Pfad des Ordners für die Einbindende CMakeLists.txt in der Einzubindenden zurückgibt.

5.6 CMake-Objekt vs. Variable

Der Unterschied besteht darin, dass ein Objekt eine zu generierende Auszuführende oder Bibliothek bezeichnet. Diese werden immer bei ihrem Namen ohne Dollarzeichen und Klammer aufgerufen, im Gegensatz zu den Variablen. Der Name des Objektes ist auch der der generierten Auszuführenden.

Bsp.:

```
add_executable(my_executable relative_path/my_executable.cpp)
```

Die Auszuführende hätte dann den Namen `my_executable.exe` (Bei Windows *.exe)

5.7 Praktisches Beispiel

5.7.1 Ausgangssituation

Wir haben folgende Ausgangssituation:

1. Einen Arbeitsordner mit den folgenden Komponenten:
   1.1. src-Ordner für die main.cpp
   1.2. include-Ordner für die Headers und deren .cpp's
   1.3. Eine “Master” CMakeLists.txt
2. In dem Include-Ordner ist ein Unterordner namens BaseFunctions, welcher die Dateien:
   2.1. BaseFunctions.hpp (Deklarationen),
   2.2. BaseFunctions.cpp (Definitionen) und
   2.3. CMakeLists.txt (zum Erstellen der temporären BaseFunctions.cpp-Bibliothek) besitzt.

Die Funktionen aus BaseFunctions.hpp sollen in der main.cpp verwendet werden. Wie zu Beginn des Abschnitts bereits beschrieben muss eine temporäre Bibliothek für die BaseFunctions.cpp zum Linken erstellt werden.

5.7.2 Master-CMakeLists-Header-Anpassung

Wir beginnen mit der “Master” CMakeLists.txt.

Zu Beginn der CMake müssen die Folgenden Informationen angegeben werden:

1. CMake-Build-Version,
2. Projektname und
3. zusätzliche Compiler-Attribute an.

```
cmake_minimum_required(VERSION 3.8)
project(my_test_project)

#Compiler options
if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic -fopenmp)
endif()
```

Hierbei sind die Compiler-Optionen optional, wobei die If-Abfrage dazu genutzt werden kann um je nach Betriebssystem für den Compiler richtige Optionen einzustellen.

5.7.3 ROS-Spezifische Paketeinbindung (find_packages)

Da ROS auf einer “Paketstruktur” basiert. Also ein Paket viele verschiednene Nodes umfasst, welche eine spezielle Aufgabe oder einen speziellen Aufgabenbereich bearbeiten, müssen wir um Funktionen aus diesen Nodes, wie Nachrichtenstrukturen, Navigation und Kommunikation nutzen zu können, diese nicht nur im Quellcode richtig Verlinken `#include<geometry_msgs/msg/Twist>`, sondern diese auch zu unserer “Master”-CMake hinzufügen. Hierfür müssen die notwendigen Pakete gefunden werden.

Dies geschieht mittels dem `find_package()` Ausdruck, wie in den nachfolgenden Beispielen zu sehen ist:

```
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
find_package(tf2 REQUIRED)
find_package(geometry_msgs REQUIRED)
```

Hierbei sind `ament_cmake` und `rclcpp` die wichtigsten Basis-Pakete.

1. `ament_cmake`: Build-Tool welches für das Erstellen von paketbasierten Projekten spezialisiert ist.
2. `rclcpp`: Basis-Paket von ROS

5.7.4 Erstellen eines Targets

Im Folgenden wird der Ausdruck `my_target` äquivalent für die Auszuführende mit dem Target-Namen `my_exe_target` und die Bibliothek mit dem Target-Namen `my_lib_target` verwendet. Ein Target beschreibt also das Erstellen einer Bibliothek oder einer Auszuführenden, also das Ziel.

5.7.4.1 Bibliotheks-Target

```
add_library(my_lib_target SHARED
${CMAKE_CURRENT_SOURCE_DIR}/include/BaseFunctions.cpp)
```

Um ein Bibliotheks-Target zu erstellen geben müssen die folgenden Informationen gegeben sein:

- Name der Bibliothek (hier `my_lib_target`),
- Typ der Bibliothek (STATIC|SHARED) und
- den Pfad zu der `.cpp` von der die Bibliothek erstellt werden soll.

💡 Ist die Bibliothek temporär, also wenn sie nicht extra installiert wird oder nur für Definitionen von Funktionen oder Klassen in den .cpp Dateien benötigt wird, so muss sie STATIC sein. Soll dies mit einer SHARED-Bibliothek gemacht werden, so muss diese installiert, also eine Bibliotheksdatei erzeugt werden. Anschließend muss der Pfad zu dieser Datei verlinkt oder in den Systemumgebungsvariablen abgelegt werden, damit die Auszuführenden die geteilte Bibliothek während der Laufzeit laden, auf enthaltene Funktionen zugreifen und diese ausführen können.

5.7.4.2 Auszuführenden-Target

```
add_executable(my_exe_target ${CMAKE_CURRENT_SOURCE_DIR}/src/main.cpp)
```

Um ein Auszuführenden-Target zu erstellen, müssen die folgenden Inforationen gegeben sein:

- Name der Auszuführenden (hier `my_exe_target`) und
- den Pfad zu der `.cpp` von der die Auszuführende erstellt werden soll.

5.7.4.3 Target-Include-Verzeichnis

Damit die einzubeziehenden Header vom Compiler gefunden werden können müssen die dafür benötigten Include-Pfade jedem Target einzeln zugewiesen werden.

```
target_include_directories (
my_target
PUBLIC
$<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
$<INSTALL_INTERFACE:include>
additional_paths
)
```

Die Zuweisung der Include-Pfade geschieht wie Folgt:

- Zielobjekt (`my_target`), also `my_lib_target` oder `my_exe_trget`
- `$<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>`: Pfad zum Include-Ordner des Arbeitsordners (Ordner in dem die "Master" CMake liegt)
- `$<INSTALL_INTERFACE:include>`: Installationsort der Standard-Bibliotheken
- `additional_paths`: Zusätzliche Pfade (Absolut) zu Third-Party-Bibliotheks-Header

5.7.4.4 Target-Compile-Features

Nachdem die Include-Verzeichnisse festgelegt wurden, muss für jedes Target der Sprachstandard festgelegt werden, der bei der Kompilierung des Codes verwendet werden soll.

Für C++ wäre der neuste Standard `cxx_std_23` und für C wäre es `c_std_11`.

Das Festlegen funktioniert so:

```
target_compile_features(my_target PUBLIC c_std_11 cxx_std_23)
```

5.7.4.5 ROS-Spezifische Target-Abhängigkeiten

Da ROS auf einer “Paketstruktur” basiert und wir die Pakete wie Header einbinden, um auf für das Paket spezifische Funktionen und Klassen zugreifen zu können, müssen wir die verwendeten ROS-Pakete dann, wie die Header hinzufügen. Dies geschieht mit dem ament Build-Tool-Ausdruck:

```
ament_target_dependencies(
my_target
rclcpp          #packages
std_msgs
tf2
geometry_msgs
...
)
```

Hierbei ist `rclcpp` das Basis-ROS-Paket.

5.7.4.6 Binden einer Bibliothek zu einem Target

Um eine Bibliothek einem Auszuführenden-Target zur Verwendung zur Verfügung zu stellen, muss der Ausdruck `target_link_libraries(target lib1 lib2 ... libn)` verwendet werden.

Dies kann auf zwei Arten geschehen:

1. **Interne Verwendung**: Wenn die Library nur zur Erstellung der Auszuführenden benötigt wird. Dies ist der Fall, wenn bspw. die Funktionsdefinitionen einer .cpp verlinkt werden sollen. Hierbei wird einfach das Bibliotheks-Target verlinkt. Wichtig ist, dass bevor das Bibliotheks-Target verlinkt werden kann, muss das Target komplett erstellt sein, also alle `include`- und `compile_features` sowie `dependencies` gesetzt sein.

```
target_link_libraries(my_exe_target my_lib_target)
```

1. **Externe Bibliothek**: Wenn eine externe Bibliothek einer Third-Party verwendet werden soll, kann auch die Bibliothek direkt verlinkt werden.

```
target_link_libraries(my_exe_target /path_to_lib_folder/third_party_lib.lib)
```

5.8 Ausgabenerstellung

In diesem Abschnitt geht es darum, wie aus einem Bibliotheks- und Auszuführenden-Target eine ausführbare Datei generiert werden kann.

5.8.1 Erzeugen einer Bibliotheksdatei aus Bibliothekstarget & einer Auszuführenden aus Auszuführenden-Target

Eine Ausführbare Bibliothek kann mit dem folgenden Befehl installiert werden:

```
install(TARGETS
my_target
...
DESTINATION /path
)
```

In diesem Fall würde `my_lib_target` mit den spezifischen Endungen:

- Linux: .a (statisch) .so | .o (dynamisch)
- Windows: .lib (statisch) .dll (dynamisch)

Und `my_exe_target` mit den Endungen:

- Linux: .run .sh .bin .tar.gz.
- Windows: .exe

an dem Zielort, festgelegt durch den Pfad `/path`, installiert werden.

Hiermit können mehrere Targets an den gleichen Ort installiert werden. sollen die Targets an verschiedene Orte installiert werden, so muss dann jeweils ein neues `install()` vorgenommen werden.

5.8.2 Export und Installation von Ordnern für Auszuführende

Damit die Auszuführenden auch auf die benötigten Daten aus den Ordnern der Arbeitsmappe zugreifen können, wenn sie an einem anderen Ort installiert werden, ist es notwendig die Dokumente und Ordner an dem gleichen Ort zu installieren:

```
install(DIRECTORY
path_to_folder_1_starting_from_ CMakeLists _folder
...
path_to_folder_n_starting_from_CMakeLists_folder
DESTINATION /path
)
```

Files können so installiert werden:

```
install(FILES
path_to_file_1_with_file_ending_starting_from_ CMakeLists _folder
...
path_to_file_n_with_file_ending_starting_from_CMakeLists_folder
DESTINATION /path
)
```

Auch hier gilt wieder: Sollen Files oder Ordner an verschiedene Orte installiert werden, so muss jedes Mal ein extra `install` mit diesem Pfad vorgenommen werden.

5.8.3 ROS-Installationsortangabe

Bei ROS muss der Installationsort dieses Format haben:

```
install(<...>
<...>
DESTINATION lib/${PROJECT_NAME}/some_additional_path
)
```

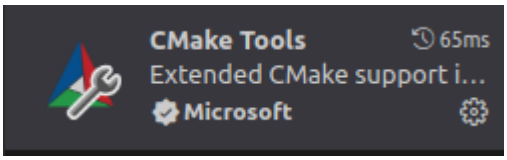
Hierbei wird alles unter `install/PROJECT_NAME/lib/PROJECT_NAME/some_addional_path` installiert. Wobei `PROJECT_NAME` die CMake-Standardvariable ist, welche den Projekt-/Paketnamen enthält, den wir für unser Projekt festgelegt haben.

6 Debugging mit CMake und VSCode

6.1 Voraussetzung

Um ein CMake-Projekt in Visual Studio Code zu debuggen, werden die Folgenden Elemente/Strukturen benötigt:

1. `.vscode`-Ordner mit den Files:
   1.1. `c_cpp_properties.json`
   1.2. `settings.json`
2. Und die “CMake Tools” Erweiterung von Microsoft:



3. Zudem wird ein Ordner mit einer CMakeLists.txt, welche eine bestimmte oder alle Auszuführenden (Programme/ .exe) ihres Projektes erstellt.

6.1.1 c_cpp_properties.json

Die Erstellung der `c_cpp_properties.json` ist beim Arbeiten mit CMake Proform und ist nur beim Schreiben des Codes von Relevanz.

Das Erstellen dieser Datei dient dazu, dass IntelliSense in der Lage dazu ist, eingebundene Header, welche mittels `#include` zum Code hinzugefügt wurden, zu finden und die darin enthaltenen Methodennamen zu extrahieren, um dann bei der Eingabe Methodenvorschläge machen zu können.

Da jedoch die erstellende CMakeLists.txt ebenso die Include-Informationen enthält, und das Debuggen mit CMake und der CMakeLists.txt vorgenommen wird, ist CMake nicht auf die `c_cpp_properties.json` angewiesen.

Die `c_cpp_properties.json` wäre im Punkto Debugging nur relevant, wenn das Debugging mittels Visual Studio Code selbst in Kombination mit zusätzlichen Files, wie `launch.json` vorgenommen werden würde.

Die `c_cpp_properties.json` hat bspw. die folgende Struktur:

```
{

 "configurations": [
  {
   "browse": {
    "databaseFilename": "${default}",
    "limitSymbolsToIncludedHeaders": false
   },
   "includePath": [
    "/usr/include/**",
    "/opt/ros/humble/include/**",
    "/path_to_my_project/src/test_pkg/include/**"
   ],
   "name": "ROS",
   "intelliSenseMode": "gcc-x64",
   "compilerPath": "/usr/bin/gcc",
   "compileCommands": "${workspaceFolder}/build/compile_commands.json",
   "cStandard": "gnu11",
   "cppStandard": "c++23",
   "compilerArgs": [
    "-fopenmp"
   ]
  }
 ],
 "version": 4
}
```

Wie bereits angesprochen ist hierbei "`includePath`" der wichtigste Teil. Dieser gibt die Pfade zu den Ordnern an, in denen die einzubindenden Header zu finden sind, wie die Standardheader.

Die anderen Informationen, wie der für die Kompilierung zu verwendete C und C++ Standard können ebenso noch angegeben werden.

Diese sowie auch der Pfad zum Compiler (dessen Auszuführenden), wenn nicht der standardmäßig auf dem System vorhandene Compiler verwendet werden soll, müssen in der CMakeLists.txt angebenden werden.

6.1.2 settings.json

Die für das Debuggen mit CMake wichtige Datei ist die `settings.json` diese wird auch wie die `c_cpp_properties.json` automatisch von VSCode zur Erstellung der `.cpp` generiert.

Um das Debuggen mit CMake zu ermöglichen, muss dann

```
"cmake.sourceDirectory": [
"${workspaceFolder}/src/fast_slam2_0",
"${workspaceFolder}/src/test_pkg"
]
```

hinzugefügt werden.

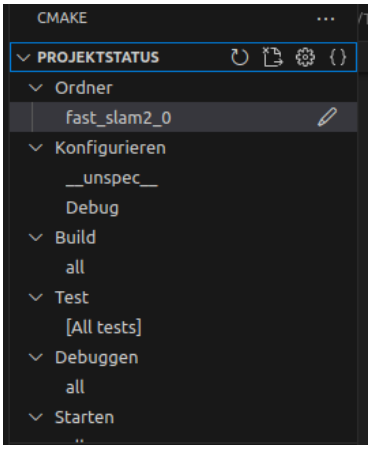
Mit diesem Befehl wird bei jedem Starten von VSCode die CMake-Tools-Erweiterung zum Arbeitsordner hinzugefügt.

Daraufhin erscheint das folgende Symbol in der linken Tool-Leiste von VSCode.

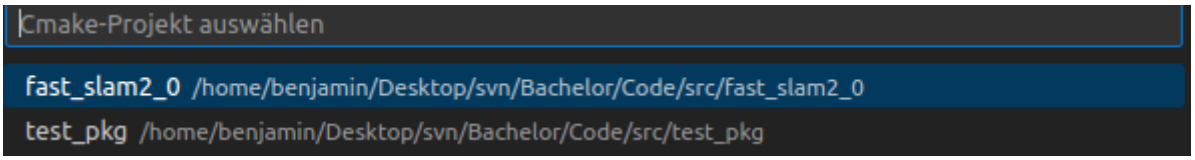
Mit dem Befehl "`cmake.sourceDirectory`" werden die Pfade zu den Ordnern angegeben, in denen sich die CMakeLists.txt befinden, welche die Auszuführenden des aktuellen Projektes erstellen.

6.2 CMake Debugging ausführen

Um jetzt das Debugging zu starten, muss auf das neu erschienene Symbol geklickt werden, worauf ein Menü ausklappt.



Unter **Ordner** kann jetzt der Ordner der CMakeLists.txt ausgewählt werden, welche erstellt werden soll. Zum Auswählen muss auf das Symbol "*Aktiven Ordner auswählen*" geklickt werden. Daraufhin erscheint ein **Auswahlfenster**.



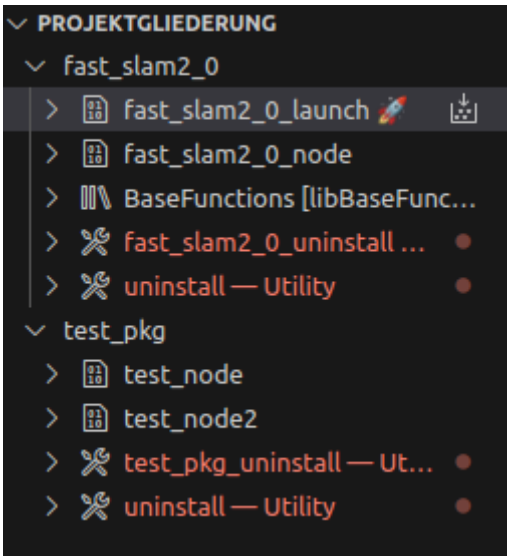
In diesem Auswahlfenster sind die unter "`cmake.sourceDirectory`" angegebenen Ordner zu finden.

Zum Ausführen des Debuggings:

- Muss zunächst der Debugg-Typ und der verwendete Compiler unter **Projektstatus>Konfigurieren** eingestellt werden.
- Müssen alle Auszuführenden der CMakeLists.txt erstellt werden, mit der Auswahl des Build-Symbols unter **Projektstatus>Build**.
- Müssen die zum Debuggen Auszuführende unter **Projektstatus>Debuggen** ausgewählt und dann auf das Debugg-Symbol geklickt werden.

Soll die Auszuführende einfach gestartet werden, ohne sie zu Debuggen, kann dies unter **Starten** vorgenommen werden.

Nachdem dies gemacht wurde, erscheint im unteren Teil des “Debugg-Menüs” die nachfolgende Ansicht:



Es ist zu sehen, dass die CMakeLists unter den Ordnern **fast_slam2_0** und **test_pkg** jeweils zwei Auszuführende erstellen:

- **fast_slam2_0**:
    1. fast_slam2_0_launch
    2. fast_slam2_0_node
- **test_pkg**:
    1. test_node
    2. test_node2

Zwischen diesen Auszuführenden kann, wie zuvor beschrieben, zum Debuggen, für den jeweils aktiven Ordner ausgewählt werden.