HSP Projektbericht

Philipp Eidenschink, Florian Laufenböck, Tobias Schwindl
Matrikelnummern : 3080919, 2894759, 3080498

22. Oktober 2016

Inhaltsverzeichnis

1. Einleitung

Dieser Projektbericht beschreibt die Tätigkeiten der Autoren im Laufe des HSP[1] im Sommersemester 2016. Diese beinhalten im wesentlichen zwei Teile:

- Ersetzen eines ROS[2] Knotens durch eine selbstgeschriebene Applikation inkl. Wrapperapplikation für das Visualisierungstool RVIZ[3].
- Vergleich verschiedener SLAM[4] Algorithmen bzgl. ihrer Einsetzbarkeit für das ALF[5] Projekt.

Motivation Ersatz für ROS Knoten ROS ist ein Open-Source Framework das umfangreiche Bibliotheken für die Ansteuerung von Sensoren und Motoren, dem Arbeiten mit komplizierten Algorithmen und dem schnellen Aufbau von funktionierenden Roboterprototypen anbietet. Ein Nachteil einer solch umfangreichen Bibliothek ist es, dass der Überblick über die verschiedenen Module die es gibt, was diese tun und wie diese etwas tun sehr schnell verloren geht. Desweiteren ist die Einarbeitung recht kompliziert und arbeitsintensiv. Der Grund warum in diesem Hauptseminar ein vorhandener und funktionierender ROS Knoten ersetzt wurde ist aber ein anderer: Der Raspberry Pi, der aktuell als zentrale Steuereinheit des ALF dient, ist mit den darauf laufenden Applikationen ausgelastet bzw. teilweise überlastet. Ein möglicher Grund wurde in den umfangreichen Modulen und Aufbau der Kommunikation von ROS vermutet. Das Ziel dieser Projektarbeit ist es also einen vorhandenen ROS Knoten durch eine eigene Applikation zu ersetzen und zu validieren, ob dieser Aufwand den erhofften Performancegewinn bringt.

Vergleich verschiedener SLAM-Algorithmen Aufbauend auf den Ergebnissen der ersten Teilaufgabe sollten dann verschiedene Algorithmen zur Umweltkartenerzeugung und Positionsbestimmung verglichen werden, um ALF langfristig autonom durch die Gänge des Fakultätsgebäudes manövrieren zu lassen.

[1] Hauptseminar Projektstudium
[2] Robot Operating System
[3] ROS Visualization
[4] Simultaneous Localization and Mapping
[5] Autonomes Laser Fahrzeug

2. Portierung des ROS-urg-node

2.1. Aktuelle Situation und Motivation

Wie bereits erwähnt wurde bei der softwareseitigen Umsetzung des ALF-Projektes auf ROS gesetzt. Ein funktionaler Hauptbestandteil des ALF ist die Kartographierung der Umgebung und die Lokalisierung des Roboters in ebendieser mit Hilfe eines externen PCs. ROS stellt dazu Pakete zur Verfügung, welche eine Übertragung aller relevanten Daten, wie zum Beispiel der Entfernungsdaten eines Lidar[6] über eine Socketverbindung und die Entgegennahme mit anschließender Auswertung und Verwendung, erlauben. Im vorliegenden Projekt wird das Senden der Daten von dem ROS-Knoten *urg_ node* übernommen. Empfangen werden diese Daten von einer Gegenstelle auf welcher sich ebenfalls das ROS Basispaket zusammen mit dem Paket *hector_ mapping* befindet.
Um das vorhandene Lastproblem auf dem Raspberry Pi zu lösen, sollen nun Stück um Stück die Abhängigkeitem zu ROS entfernt werden. Um zum einen die SLAM-Funktionalität zu erhalten und zum anderen die Ergebnisse der Portierung validieren zu können, wird der ROS-urg-Knoten auf dem Raspberry Pi durch eine Komponente ersetzt, die ebenfalls das Senden aller notwendigen Daten ermöglicht. Auf der Gegenstelle wandelt ein Wrapper die Daten wieder in das ursprüngliche ROS-kompatible Format um.

Ziel war dabei die bisherigen Abhängigkeiten zu ROS aufzulösen. Sobald das ROS auf dem Raspberry Pi durch eigene Implementierungen abgelöst wird können diese freigewordenen Ressourcen dazu genutzt werden um andere Rechenaufgaben zu erledigen. Diese Reserven sollen dann vor allem in die aufwändige Berechnung der Karte bzw. des zu fahrenden Weges genutzt werden.

2.2. Architektur

Um einen möglichst hohen Performancegewinn zu erreichen, zielt der Aufbau der implementierten Softwarekomponenten darauf ab möglichst wenig Rechenleistung auf dem Raspberry Pi zu verbrauchen. Deswegen sollen alle Berechnungsschritte, die nicht aus

- Lidardaten abholen und
- zwischengespeicherte Lidardaten einem Client zur Verfügung zu stellen

bestehen, nicht auf dem Raspberry Pi ausgeführt werden. Bereits dieses Ziel wird so von dem ROS-urg-Knoten nicht erreicht, da für die Funktionalität des Knotens neben der

[6] Light detection and ranging

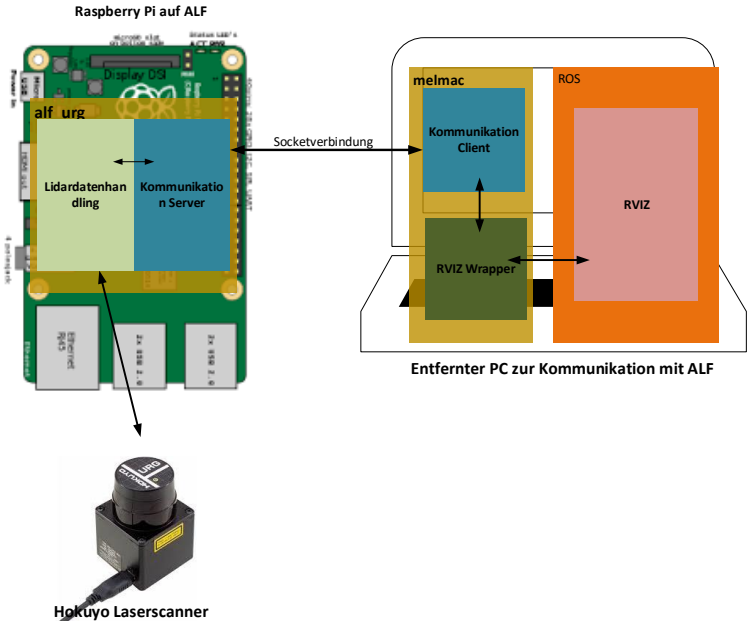


Abbildung 2.1.: Aufbau der Applikation und Nachrichtenlauf

Datenverbindung zur Übertragung der Punktwolken aus Entfernungen und dem Datenhandling mit dem Lidar außerdem noch Abhängigkeiten aus der *transform library* [1] ausgeführt werden.

Abbildung 2.1 zeigt einen groben Überblick über die vorhandenen Komponenten und deren Kommunikation. Gelb hinterlegte Teile wurden im Rahmen dieser Projektarbeit implementiert. Die Komponenten `Lidardatenhandling` und `Kommunikationsverbindung Server` werden als 2 unabhängige Threads, die über eine Queue Daten austauschen, dargestellt. Gesteuert werden die beiden Threads von einem `main` Prozess. Ähnlich verhält es sich auf der Client-Seite. Es gibt einen Prozess, der zwei Threads, einer für die Kommunikation mit dem ALF, einer als Wrapper für RVIZ, steuert. Die einzelnen Komponenten und ihre Aufgaben im Überblick:

- `Lidardatenhandling`: Kommuniziert mit dem Laserscanner und sammelt die einzelnen Punktwolken (alle $100ms$) ein. Diese werden zusammen mit Zeitstempel und einer laufenden Nummer in einer Queue gespeichert, sodass sie für die Kommunikation verfügbar sind.

- `Kommunikationsverbindung - Server/Client`: Diese Komponente ist für die Kommunikation mit dem melmac zuständig. Die Daten, die vom Lidardatenhandling über die Queue zur Verfügung gestellt werden, stellt diese Komponente über einen Socket zur Verfügung. Dazu werden die Daten in einen Bytestream umgewandelt und über den Socket versendet. Die Client Kommunikationsverbindung führt genau den umgekehrten Weg aus: Nach Empfangen der Daten werden diese in einem

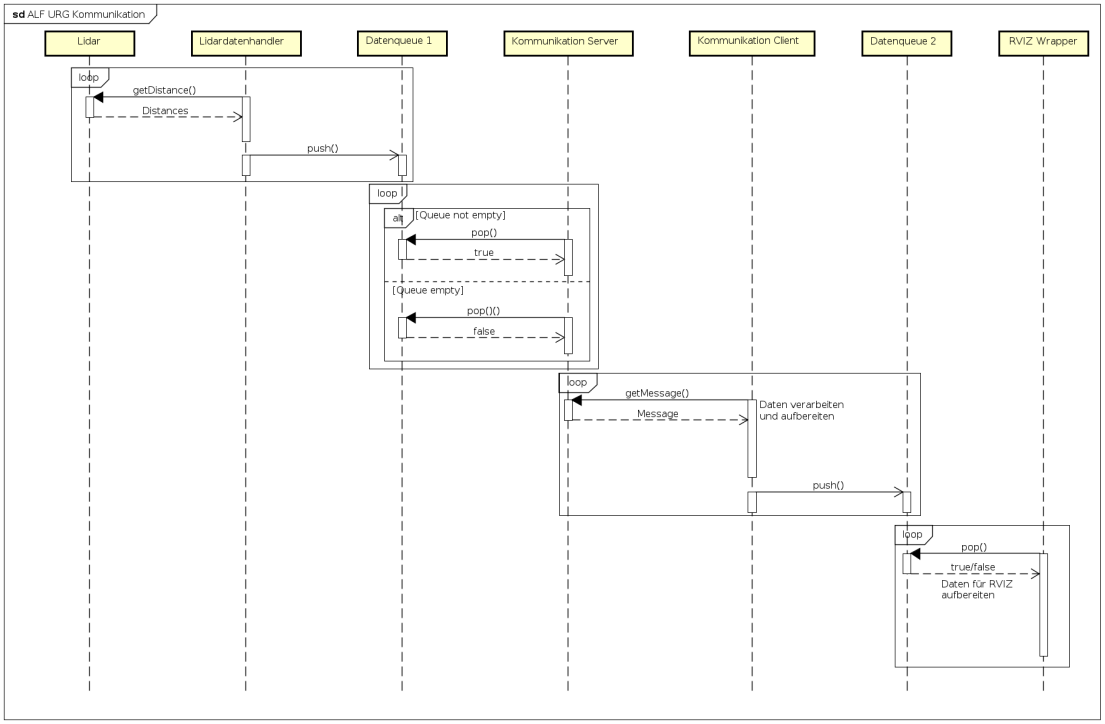


Abbildung 2.2.: Sequenzdiagramm einer möglichen Kommunikation zwischen den Komponenten

Datenspeicher für den RVIZ Wrapper gespeichert und bereitgestellt.

- `RVIZ Wrapper`: Diese Komponente wandelt die empfangene Lidardaten in ein Format um das RVIZ versteht und fügt dafür noch einige zusätzliche Informationen an bzw. ändert die Lidardaten so, dass RVIZ damit umgehen kann.

Mit der Kommunikationskomponente wurde eine Komponente geschaffen, die ohne große Änderungen auch andere Anfragen bzw. Kommunikation für denkbare andere Anwendungen[7] genutzt werden kann. Ein Sequenzdiagramm, das den typischen Ablauf der Kommunikation zeigt, ist in Abbildung 2.2 dargestellt. Die Initialisierungsnachrichten werden in dem Diagramm nicht betrachtet. Deutlich wird, dass die Kommunikation zwischen den Threads durch Queues abgebildet wird. Im Diagramm ist der vereinfachte Fall dargestellt, dass keine Fehler während der Kommunikation auftreten. Der abgebildete Fall ist auch nur einer von vielen, da es in vielen Fällen zu einer Unterbrechung der Kommunikation kommen kann:

- Die `Datenqueue 1` ist leer, während der `Server` davon lesen will. In diesem Fall kann das bedeuten, dass Fehler beim Lesevorgang vom Lidar aufgetreten sind, aber auch, dass der Server schneller aus der Queue liest, als Daten reingeschrieben werden (ein

[7] z.B. Übertragen der Geschwindigkeiten, Kontrolle der Beleuchtung etc.

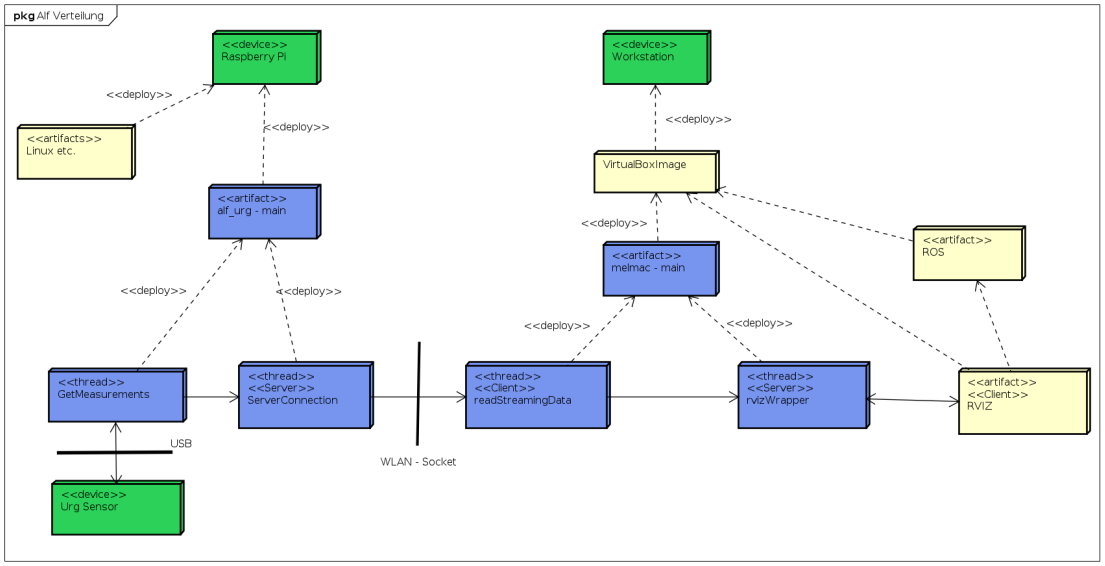


Abbildung 2.3.: Verteilungsdiagramm der Komponenten. Grün hinterlegt sind physikalische Objekte, blau hinterlegt sind selbst implementierte Objekte und gelb hinterlegt sind notwendige Objekte, die nicht selbst implementiert wurden.

Fall, der sehr häufig eintritt). Als Lösung wurde der Fall so implementiert, dass der Kommunikationsthread solange schlafen gelegt wird, wie es dauert, eine Punktwolke aus dem Lidar auszulesen (*hier:* $100ms$).

- Während der Übertragung der Daten zwischen dem `Server` und `Client` treten Fehler auf (Checksummenfehler etc.). Dann kann es passieren, dass mehrere Elemente in die `Datenqueue` gepusht werden, die dann schnell hintereinander wieder ausgelesen können.
- Weitere Unterbrechungen in der Kommunikation werden der Fantasie des Lesers überlassen.

2.3. Umsetzung

2.3.1. urg-Knoten

Im folgenden Kapitel werden die Komponenten und Klassen, die umgesetzt wurden, dargestellt und ihre Bedeutung und Funktionsweise innerhalb des Projekts erläutert. Ein detailiertes Verteilungsdiagramm der Komponenten ist in Abbildung 2.3 zu finden. Dies stellt sozusagen eine Erweiterung zur Abbildung 2.1 dar. Die nachfolgend beschriebenen Komponenten sind auch als Klassendiagramm in Abbildung 2.4 dargestellt.

- **alf-urg**: Der alf-urg Knoten läuft auf dem Raspberry Pi und stellt die Hauptappli-

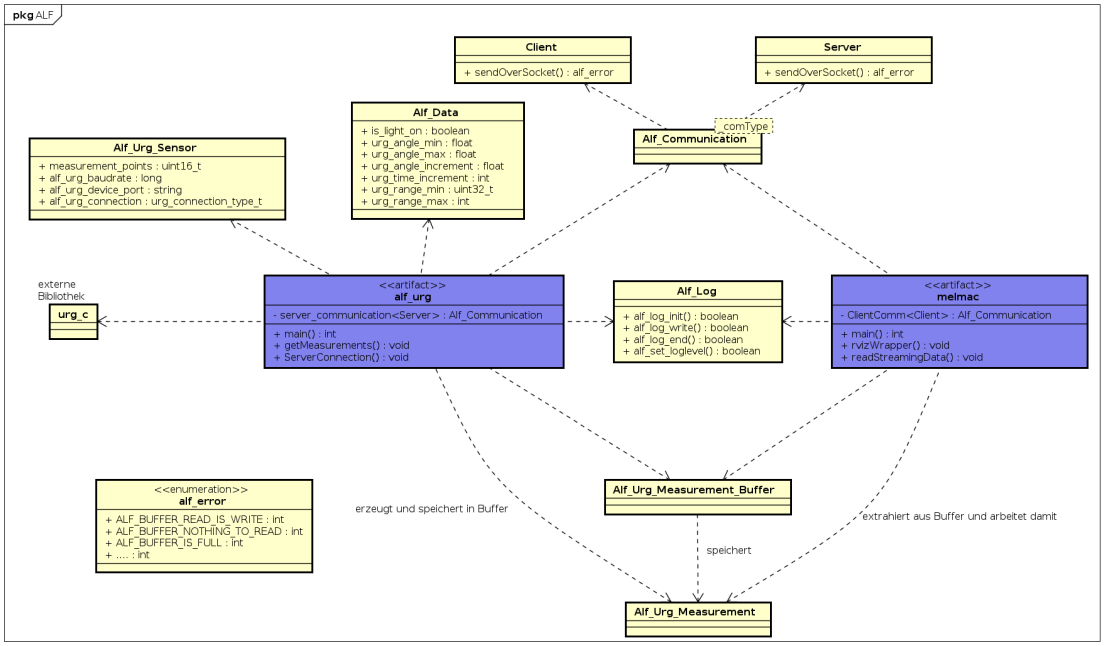


Abbildung 2.4.: Übersicht über die implementierten Klassen, deren Zusammenhang und von welcher Komponente sie verwendet werden. Der übersicht halber sind die Klassen vereinfacht dargestellt. Die blau hinterlegten Klassen stellen die beiden Applikationen dar die ausgeführt werden und sind nicht als Klassen implementiert.

kation für die gesamte Software auf dem Raspberry Pi dar. Dieser Knoten ist für das Aufbauen der Verbindung zwischen dem Pi und dem Lidar-Sensor zuständig. Ein weiterer Thread sorgt dann für die Weitergabe der gesammelten Daten über eine Socketverbindung. Im Moment findet die Weitergabe der Daten an einen weiteren Rechner statt. Es ist aber ebenso denkbar, diese Daten einer anderen, auch auf dem Raspberry Pi laufenden Applikation zur Verfügung zu stellen, die damit die Karte bzw. die Position des ALF in dieser berechnet.

- **alf-log**: Dieser Loghandler ist für das Erstellen ein oder mehrerer Logfiles zuständig um Informationen über den zeitlichen Ablauf einzelner Schritte zu erhalten. Dabei können solche Logmeldungen über verschiedene Prioritäten verfügen, die dann in das Logfile aufgenommen werden oder nicht. Auch die gleichzeitige Ausgabe auf die Konsole der Meldungen ist hiermit möglich. Dieser logging Mechanismus hat keine Abhängigkeiten zur Hardware und kann somit auch auf einem beliebigen anderem Rechner eingesetzt werden.

- **urg-c**: Diese Files stellen eine Art HAL[8] dar. Damit wird ermöglicht, dass die Funktionalität des Lidarsensors dargestellt werden kann. Es handelt sich hierbei um

[8] Hardware Abstraction Layer

eine externe Library des Lidar-Herstellers, die nur die für unser Projekt benötigten Dateien und Datenstrukturen enthält.

- **alf-sensors**: Dieses Modul stellt einige Informationen über den Sensor und dessen Kommunikation mit dem Pi dar, wie etwa die Anzahl der Messpunkte pro Messung oder den USB Port, an dem der Sensor angeschlossen ist. Dabei ist im Moment nur ein Sensor unterstützt, dessen Parameter in der Alf-Urg-Sensor Klasse verwirklicht werden.

- **alf-data**: Diese Komponente stellt die Nachrichten dar, die auf dem Raspberry Pi verwendet werden. Dazu wird der Laserscanner abgefragt und diese Daten werden in eine eigene Nachricht gepackt. Dabei werden weitere wichtige Informationen ergänzt, wie z. B. welche Bereiche der Messung gültig sind oder welchen Zeitstempel die Messung trägt. Solche Messungen werden in einem Buffer zwischengespeichert, sodass diese Messungen über die Socketverbindung weiterversendet werden können. Zusätzlich sind hier einige allgemeine Einstellungen verwaltet, wie etwa minimale(n) bzw. maximale(n) Winkel und Reichweite, für den der Sensor Werte liefern kann.

- **alf-communication**: Die komplette Kommunikationsinfrastruktur, die für das Projekt erforderlich ist, befindet sich hier. Die nach außen sichtbare Klasse Alf-Communication abstrahiert die eigentliche Kommunikation, die auf mehreren Wegen zustande kommen kann. Einerseits ist die "Kommunikation" mittels eines Files möglich. Des Weiteren ist eine TCP/IP Verbindung möglich. Für beide Wege der Kommunikation gilt:
  - Read/Write von definierten Nachrichten ist möglich
  - Read/Write von unspezifizierten Daten ist möglich
  - Init/Close der Kommunikation

  Falls weitere Möglichkeiten der Kommunikation hinzukommen sollen ist einiges zu beachten. Jeder Typ, mittels dem kommuniziert werden soll muss für das
  - Lesen
  - Schreiben
  - Initialisieren
  - Beenden

  eine eigene Funktionalität zur Verfügung stellen. Alf-Communication ist als Template definiert und kann somit explizit mit möglichen neuen Kommunikationstypen benutzt werden.

- **melmac**: Den Client und damit den Abnehmer der Lidardaten stellt melmac dar. Dabei läuft hier im Moment noch das ROS mit, um die Daten entsprechend zu visualisieren. In diesem Modul befinden sich zwei Hauptaufgaben, die zu erledigen sind. Die erste Aufgabe beschäftigt sich mit der Aufnahme der Sensordaten, die

über die Server/Client Verbindung hereinkommen. Dabei werden diese in einen Buffer gespeichert. Das wird solange erledigt, bis eine entsprechende Endnachricht kommt oder der Benutzer das Programm manuell abbricht. Das zweite Thema ist die Darstellung der empfangenen Daten mittels des RVIZ. Dazu werden die Daten wieder in ein ROS-kompatibles Format gebracht, sodass diese korrekt dargestellt werden können.

All diese Komponenten erledigen somit alles, was bisher einige ROS Knoten erledigt haben. Dabei war unter anderem auch ein Ziel die leichte Änderbarkeit und Erweiterbarkeit aller Komponenten zu ermöglichen.

2.3.2. melmac

Nachfolgend wird die softwareseitige Umsetzung des melmac, welcher weiterhin die Benutzung des ROS Pakets *hector_mapping* zur Kartographierung und Lokalisierung ermöglicht, beschrieben.
Der RVIZ-Wrapper basiert auf einer ROS Beispielapplikation, welche in der Lage ist, Demodaten im ROS internen Nachrichtenformat *LaserScan* zu übertragen. Die Nachrichten enthalten sowohl generelle Informationen zum verwendeten Lidar, als auch sämtliche Entfernungswerte einer einzelnen Messung. Diese Daten werden vom *ROS Navigation Stack* entgegengenommen und hector_mapping zur Verfügung gestellt. Der Aufbau der Nachrichten ist in Anhang A beschrieben.
Zur Erstellung einer Karte und der Lokalisierung in dieser ist es zudem notwendig sog. *transform* Nachrichten, welche Informationen zum Hardwareaufbau beinhalten, zu übertragen. Diese spezifizieren die Beziehung zwischen der Basis des Roboters, also dem fahrbaren Untergestell und dem Lidar genauer. Im Grunde wird darin die Verschiebung und/oder Drehung zwischen dem Lidar und dem eigentlichen Roboter, welche als getrennte Koordinatensysteme aufgefasst werden, beschrieben. Diese Daten wurden vorher vom Raspberry Pi versendet, was auch in Kapitel 2.2 als Abhängigkeit erläutert ist.
melmac besteht in seinem Aufbau aus zwei Threads. Der erste Thread empfängt die Daten welche vom Raspberry Pi zur Verfügung gestellt werden. Der zweite Thread verarbeitet diese, fügt weitere Daten hinzu und stellt sie hector_mapping zur Verfügung. Funktionen zum Empfangen und Zwischenspeichern der Nachrichten, werden durch Komponenten des urg-Knotens, welcher im vorherigen Kapitel beschrieben ist, bereitgestellt. Der Thread, der für die Eingehenden Nachrichten zuständig ist, empfängt diese über Methoden der Komponente alf-communication. Wird eine eingehende Nachricht empfangen, so werden die beinhalteten Daten in einem gemeinsamen Buffer, welcher in der Komponente alf-data implementiert wurde, festgehalten. Der zweite Thread überrpüft zyklisch den Inhalt des Buffers und extrahiert die Messungen aus dem Buffer. Die einzelnen Daten werden in die ROS Datenstrukturen verpackt und versendet. Dies geschieht alle 100ms. Dies entspricht der Frequenz in der der Lidar Daten zur Verfügung stellen kann. hector_mapping nimmt diese Daten entgegen und erstellt daraus eine Karte, in der der Roboter lokalisiert wird. Visualisiert wird diese Karte mit dem Tool RVIZ, das die Daten von hector_mapping entgegennimmt.

2.4. Verifikation und Vergleich mit ROS-urg-node

Die im vorhergehenden Kapitel dargestellten und erläuterten Aspekte des urg-Knotens und des ROS-Wrappers werden im folgenden mit dem orginalen ROS-urg-Knoten verglichen.

2.4.1. Performancevergleich auf Systemebene

Hier soll nun die Performanz der beiden verschiedenen Implementierungen verglichen werden. Dazu wurde eine simple Testumgebung aufgebaut:

- Der Laser ist mit dem Raspberry Pi verbunden.
- Der Laser steht auf dem Tisch und wird während der Messung nicht bewegt.
- Die Spannungsversorgung erfolgt über einen externen USB-Hub, sodass es keine Probleme mit der Stromversorgung gibt, die in verlorernen Paketen zwischen Raspberry Pi und Laserscanner enden.
- Die restlichen Kompontenen des ALF werden ausgeblendet und nicht benutzt. Die elektrischen Verbindungen zu diesen Komponenten wurden getrennt, sodass keine externen Interrupts die Ausführung unterbrechen können.

Gemessen wurde die Last im Systemkontext, die durch das Ausführen der Software auf dem Raspberry Pi zur Abfrage und Weiterleitung der Punktewolke an den Client entsteht. Das bedeutet, die Last die durch den ROS-Knoten bzw. unsere Implementierung entsteht, wurde während des Betriebs aufgenommen und ins Verhältnis zu allen anderen ausgeführten Prozessen gesetzt. Da Standard Linuxsystemprofilingtools wie *perf* mit dem auf dem Raspberry Pi vorhandenen SoC[9] (noch) nicht kompatibel sind bzw. für die Hardware nicht kompilierbar waren, wurde eine alternative Systemprofilinglösung gesucht. Die verwendete Lösung besteht aus einem selbstgeschriebenen Skript und der Verwendung des Tools *top*. Die Vorgehensweise ist folgende:

- Die Applikation (*hier: der ROS-urg-Knoten bzw. unsere Implementierung eines urg-Knotens*) wird gestartet und in einen arbeitsfähigen Zustand versetzt. Das bedeutet, die Gegenstelle muss sich mit der Applikation verbinden damit Daten gesendet werden.
- Mittels eines top-Kommandos
  `top -b -H -d 1 -n 60 > raspi1_our_code_top60.log`
  wird das Programm top dazu gebracht für eine bestimmte Zeitspanne (z.B. 60 Sekunden) alle d Sekunden (z.B. jede Sekunde) auszugeben, wie viel % Last jede ausgeführte Applikation auf dem System erzeugt hat. Das Ergebnis wird in eine Datei geschrieben, sodass man für den oben genannten Befehl eine Datei mit 60 Abbilder der Systemlast bekommt.

[9] System-on-a-Chip

	eigene Implementierung Suchwort `ourcode`	ROS-urg-Knoten Suchwörter `ros,urg_node,static_transf`
1 Minute	10.74%	2.81%+ 7.36%+ 4.81% = 14.98%
5 Minuten	10.28%	2.86%+ 7.09%+ 4.42% = 14.37%

Tabelle 2.1.: Übersicht der Systemprofilingergebnisse der beiden Knoten

- Abschließend können mit dem selbtgeschriebenen Skript `get_load_average.py` alle Einzelmessungen nach bestimmten Schlüsselwörtern durchsucht werden. Das Skript gibt als Ergebnis aus, wie viel Last alle Prozesse, die dieses Schlüsselwort in ihrem Namen enthalten, im Durchschnitt über alle Messungen erzeugt haben.

Diese Schritte wurden auf dem Raspberry Pi Model 1B+ durchgeführt. Die Messintervalle betrugen jeweils 5 Minuten und 1 Minute. Die Tests sollten zusätzlich noch auf einem Raspberry Pi Model 2 durchgeführt werden, allerdings gelang es nicht den ROS-urg-Knoten so zu konfigurieren, dass die Tests fehlerfrei durchlaufen werden konnten. Aus diesem Grund werden die Ergebnisse dieses Tests hier ausgeblendet. Die Rohdaten der Daten sind im Repository zu finden. Zur genauen Struktur dessen siehe Anhang.

Tabelle 2.1 fasst die Ergebnisse aus den Auswertungen zusammen. Da der ROS-Knoten mehrere Abhängigkeiten hat muss auch nach mehreren Begriffe gesucht werden. Der Begriff `ros` fasst dabei alle direkten Abhängigkeiten zusammen, die durch die Ausführung des `rosmasters` etc. entstehen. `urg_node` ist selbsterklärend, `static_transf` steht für static_transform, das auf dem gleichen Gerät, auf dem der urg-Knoten läuft, ausgeführt werden muss. Aufgezeichnet wurden alle Artefakte des Codes, d.h. erzeugte threads sind in den top-Ausgaben als einzelne Punkte aufgeführt. Es zeigt sich, dass der Performanceunterschied des ROS-Knoten ggü. der eigenen Implementierung auf die Ausführung des `static_transform` Codes zurückzuführen ist. Während der Projektphase wurde keine Möglichkeit gefunden, diesen Teil auszulagern und zusammen mit RVIZ auf der eingerichteten Virtuellen Maschine auszuführen.

2.4.2. Codeprofiling der Applikation

Zusätzlich zur Analyse im Systemkontext wurde der `alf_urg` noch einem Codeprofiling unterzogen. Dazu wurden Debug-Symbole in den Code mit hineinkompiliert, durch die während der Ausführung Informationen extrahiert werden können, die wiederrum mit dem Tool `gprof` analysiert werden können. Diese Analyse beantwortet die Frage welche Codeteile wieviel Prozent der gesamten Ausführungszeit beanspruchen und soll nur kurz ausfallen.

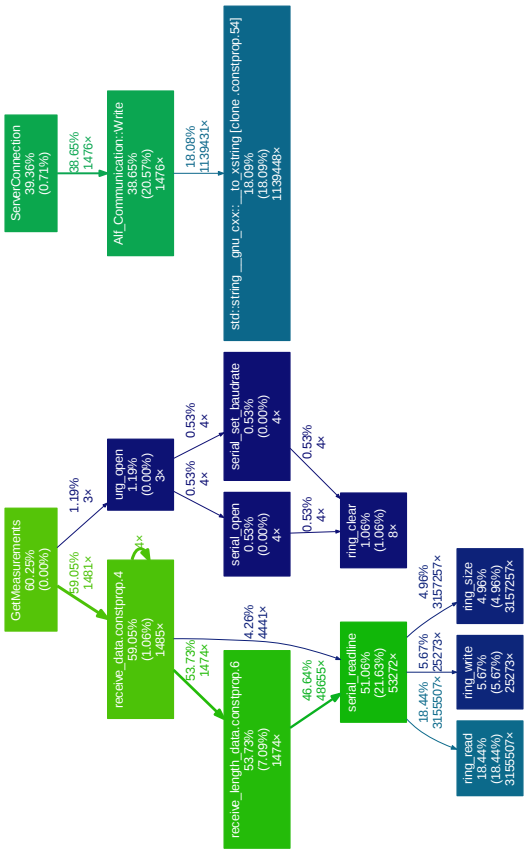


Abbildung 2.5.: Prozentuale Ausführungszeit der Funktionen, die am meisten Zeit beanspruchen. Diese Graphik wurde mit `gprof2dot`[10] aus den grpof Daten generiert.

Abbildung 2.5 zeigt einen Ausführungsgraph der aus den Profilindaten gewonnen wurde. Bei den einzelnen Knoten ist der Funktionsname und die prozentuale Ausführungszeit angegeben. Der ProzentWert in Klammern sagt aus, wie viel Prozent der Ausführzungszeit in der Funktion, nach der der Knoten benannt ist, verbracht wurde. So hat der Knoten `GetMeasurements` zwar eine Ausführungszeit von 65%, die Funktion `GetMeasurements` wurde dabei aber quasi nicht ausgeführt, sondern nur Funktionen, die sie selbst aufgerufen hat. Bei näherer Betrachtung des Graphen wird ersichtlich, dass alleine das Übertragen der Messwerte vom urg-Sensor zum Raspberry Pi bereits 46.64% der Gesamtausführungszeit ausmacht. Die Verbindung zum Client verbraucht hingegen ca. 40%, wobei dabei knapp 20% auf die Umwandlung in einen string benötigt werden. Der kleine Prozentsatz, der auf 100% fehlt, wird durch die eigentliche main-Funktion herbeigeführt,

[10] https://github.com/jrfonseca/gprof2dot

die die Initialisierung und den Aufbau der Verbindung handhabt. Etwas verwunderlich ist die Tatsache, dass die Funktion `urg_open` Rechenleistung verbraucht, da innerhalb des threads GetMeasurements der urg-Knoten eigentlich schon verbunden und initalisiert wurde. Dies kommt durch das Hinzufügen der Profilinginformationen zustande, da die Verbindung zum Laserscanner völlig nicht-detereministisch in Timeouts läuft oder bei der Übertragung Checksummefehler auftreten. Dies passiert nur wenn das Profiling aktiviert wurde und lies sich während der vielen Testfahrten ohne Profiling nicht einmal beobachten. Die in der Abbildung 2.5 dargestellten Daten sind im Unterordner `gprof_RaspiWithoutROS_REV86_maxCompiler` gespeichert. Dieser Datensatz steht beispielhaft für das Profiling welches öfter ausgeführt wurde und immer ähnliche Ergebnisse geliefert hat. Die folgenden Rahmenbedingungen galten bei der Datenerhebung:

- aktuellstes Raspian-Betriebssystem
- keine ROS-Installation auf dem Raspberry Pi
- Compileroptimierungseinstellungen auf $-O3$, d.h. maximale Compileroptimierungen, obwohl diese durch den nächsten Punkt wieder abgeschwächt werden
- Debugeinstellungen auf $-g3$
- gprof Elemente wurden mit $-pg$ einkompiliert
- der Softwarestand entsprach dem der SVN-Revision 86

2.4.3. Verifikation

Neben des Profilings wurde noch eine Verifikation durchgeführt. Ziel war es sicherzustellen, dass die eigene Implementierung keine schlechteren Ergebnisse als der orginale ROS-urg-Knoten liefert. Dazu wurden zwei Tests durchgeführt, die im folgenden vorgestellt werden.

Abbildung 2.6 zeigt ein Aktivitätsdiagram, dass den Validierungsprozess, der aus zwei Testpfaden besteht, des RVIZ_Wrappers darstellt. Die Ausgangsbasis beider Pfade bilden Lidardaten, welcher durch eine Referenzfahrt generiert und aufgenommen wurden. Die Daten liegen hierbei als Textdatei im ROS eigenem Format vor.
Zur Validierung der Ergebnisse des rviz_wrappers werden diese Daten mithilfe eines ROS Tools wieder „abgespielt“ und dabei versendet. Somit ist es möglich, mit hector_mapping aus diesen Daten eine Karte zu erstellen und den Weg der Referenzfahrt zu ermitteln. Dadurch ergibt sich das Referenzergebnis, welches allein mit ROS Komponenten erstellt wurde.
Um den rviz_wrapper testen zu können, werden die aufgenommenen Daten in das Format des eigenen urg Knoten umgewandelt. Anschließend lassen sich diese mithilfe der unterstützten Kommunikation über eine Datei versenden, so dass der rviz_wrapper diese empfangen kann und sie wie in Kapitel 2.3.2 beschrieben, hector_mapping zur Verfügung stellt. Im direkten Vergleich beider erstellten Karten, lassen sich weder Defizite im

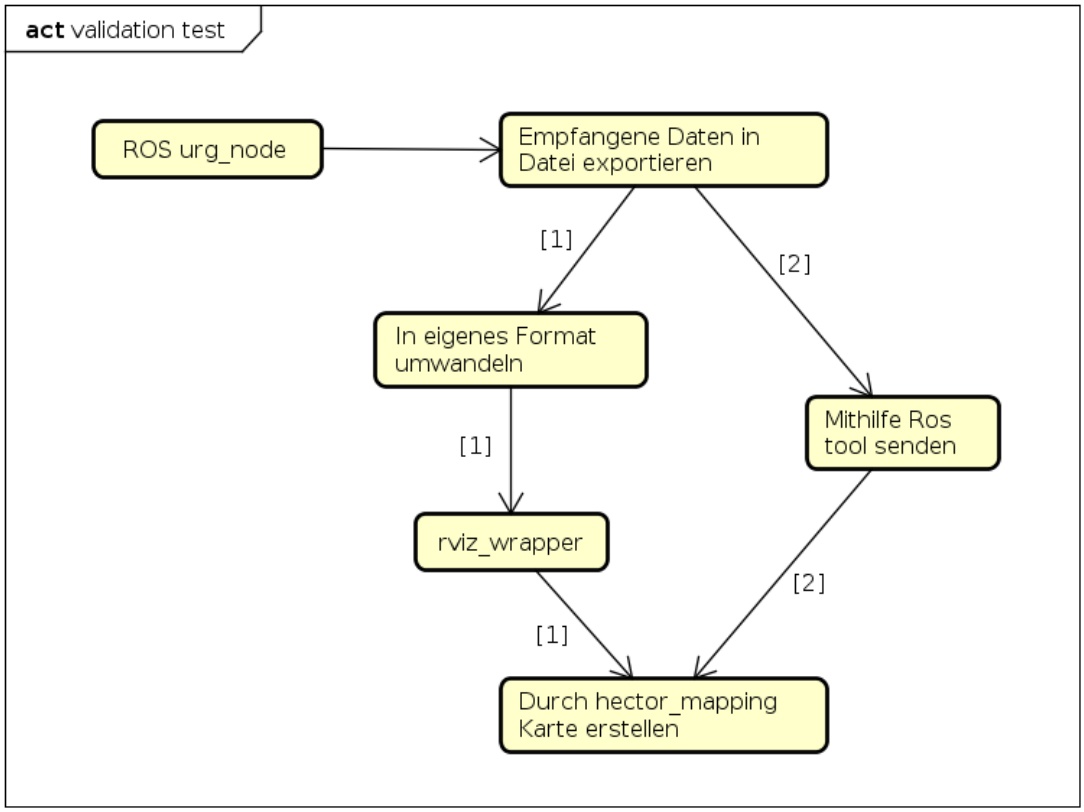


Abbildung 2.6.: Aktivitätsdiagramm zur Veranschaulichung des Ablaufs des Validierungstests zur Untersuchung des rviz_wrappers.

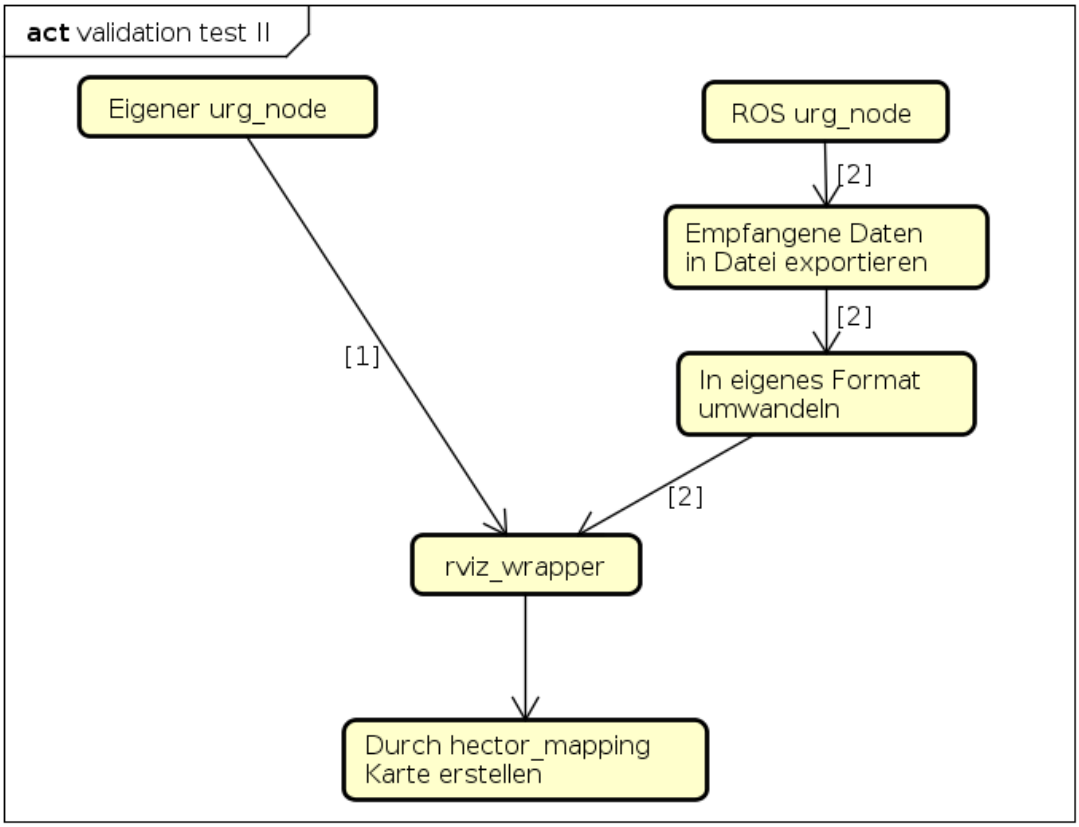


Abbildung 2.7.: Aktivitätsdiagramm zur Veranschaulichung des Ablaufs des Validierungstests zur Untersuchung des eigenen urg_Knotens.

Ablauf, noch in den Ergebnissen feststellen.

Abbildung 2.7 zeigt den Ablauf des Tests, der durchgeführt wurde um die Funktionalität des urg-Knotens zu untersuchen.
Auch dabei gibt es zwei Testpfade, um einen direkten Vergleich beider Ergebnisse zu ermöglichen. Im ersten Pfad wird auf dem Roboter der eigene urg Knoten ausgeführt, welcher sämtliche Daten aus dem Lidar ausliest und diese versendet. Der rviz_wrapper empfängt diese Daten und stellt sie hector_mapping zur Verfügung. Dort wird eine Karte der empfangenen Daten erstellt.
Im zweiten Pfad wird der ursprüngliche ROS-urg-Knoten ausgeführt, um die Daten aus dem Lidar auszulesen. Diese werden wie im ersten Test exportiert und in das eigene Format umgewandelt. Anschließend wird auch mit diesen Daten durch den rviz_wrapper und hector_mapping eine Karte der Daten erstellt. Auch bei diesem Test waren die Ergebnisse des eigenen urg Knotens dem des ROS-urg-node mindestens gleichwertig.

Die Ergebnisse dieser Tests zeigen, dass die selbst entwickelten Komponenten die Mög-

lichkeit bietet, bis auf die Kartographierung und Lokalisierung, komplett auf ROS zu verzichten, ohne Einschränkungen hinnehmen zu müssen.

3. Vergleich verschiedener SLAM Algorithmen

Nachdem im vorhergehenden Kapitel nun bereits eine Datenerhebung, Datenübertragung und einfache Datenverarbeitung der Daten vom Laserscanner vorgestellt wurden sollen nun verschiedenene SLAM-Algorithmen vorgestellt und verglichen werden. Das gesamte Fernziel des ALF ist, dass sich, das Fahrzeug autonom und ohne Hilfe von außen in einer unbekannten Umgebung zurechtfinden und navigieren kann. Dazu ist es notwendig eine Karte der Umgebung Schritt für Schritt aufzubauen und die Position dieses Fahrzeugs möglichst exakt darin zu bestimmen. Diese Problemstellung wird in der Robotik allgemein als SLAM-Problem bezeichnet. Es gibt unzählige von Algorithmen dafür, einige mit Codebeispielen, bei anderen werden nur die theoretischen Lösungen mit Beispielen dargestellt. Nutzt man für Lokalisierung und den Aufbau der Karte ein vorhandenes Framework wie ROS entsteht ein notwendiger Overhead (siehe letztes Kapitel), der sich durch eine auf die vorliegende Situation speziell angepasste Implementierung soweit wie möglich reduzieren lässt. Dazu ist es aber u. U. notwendig eine eigene Implementierung vom Scratch auf zu schaffen. Der Nachteil einer solchen Lösung ist der Zeitfaktor, eine eigene Implementierung kostet durch Einarbeitung, Entwicklung und Debugging viel Zeit, und die notwendige Einarbeitung und die Vorauswahl der überhaupt in Frage kommender Algorithmen.
Um für zukünftige Projekte am ALF bereits eine Vorarbeit durch konkretes Ausschließen bzw. in Frage kommendener Algorithmen zu leisten, soll im folgenden Kapitel diese Arbeit geleistet werden.

3.1. Zielsetzung

Bei der Auswahl eines SLAM Algorithmus als geeignete Basis sind einige wichtige Punkte zu beachten, welche im folgenden kurz erläutert werden.

- **Resourcenverbrauch**: Bei allen betrachteten SLAM Algorithmen wurde das Hauptaugenmerk auf den Resourcenverbrauch gelegt. Da alle SLAM Funktionalitäten in einem späterem Schritt auf dem Raspberry Pi durchgeführt werden sollen, gibt dieser die Grenze nach oben vor.
- **Ergebnisse**: Bei Eingabe von Lidardaten ist vor allem auf ein möglichst genaues und valides Ergebnis Wert zu legen. Dieser Punkt wird in den nächsten beiden Kapiteln genauer beschrieben.

- **Erweiterbarkeit**: Um verschiedenen Anforderungen gerecht zu werden, ist eine einfache Erweiterbarkeit des SLAM Algorithmus notwendig. So ist sichergestellt, dass sich das System auf den verwendeten Lidar anpassen lässt und z.B. auch die Verwendung von Beschleunigungs-/ oder Raddrehzahlsensoren durch moderate Änderungen möglich ist.
- **Abhängigkeiten**: Ein weiterer Punkt sind die Abhängigkeiten, die der SLAM Algorithmus hat. Damit sind Pakete gemeint, die z.B. notwendig zur Berechnung oder Darstellung der erzeugten Karte sind. Um ein flexibles System zu erzeugen, sollten daher so wenig Abhängigkeiten zu Fremdpaketen vorhanden sein.
- **Graphische Oberfläche**: Ein weniger wichtiger Punkt, der jedoch trotzdem betrachtet wurde, ist das Vorhandensein einer graphischen Oberfläche. Ist diese bereit enthalten, so lassen sich erste Validierungstests einfach durchführen. Durch das Visualisierungspaket RVIZ ist es jedoch auch möglich, eine sog. *Occupancy Grid Map* wie sie von SLAM Algorithmen erstellt wird, darzustellen.

3.2. Übersicht und Vergleich der betrachteten Algorithmen

Es wurden einige verschiedene Algorithmen betrachtet und wenn möglich auch kurz sowohl auf einem Desktop/Notebook-Rechner als auch auf dem Raspberry Pi (2) getestet um anschließend daraus einige, für das Projekt relevante Schlussfolgerungen zu ziehen. In diesem Kapitel werden nun diese verschiedenen Ansätze kurz skizziert und diskutiert:

- **TinySLAM**: Dieser Algorithmus zielt darauf ab möglichst wenig komplex zu sein (ca. 200 Lines of Code) und damit auch weniger Ressourcen zu verbrauchen. Dieser Ansatz ermöglicht dabei zusätzlich zu den Lidar-Daten auch Odometriedaten mit einzubeziehen. Allerdings konnte hier in der wenigen Zeit keine Lauffähige Version erstellt und getestet werden. Aufgrund der schnellen Berechnungen die durchgeführt werden, wird damit das Ergebniss höchstwahrscheinlich etwas ungenauer sein, als bei anderen Algorithmen. Die geringe Komplexität des Codes wird aus diesem Grund andererseits eine einigermaße leichte Erweiterbarkeit ermöglichen und enthält wenig Abhängigkeiten. Eine graphische Oberfläche wird mit dem SDL Framework erreicht.
  https://openslam.org/tinyslam.html
- **Mobile Robot Programming Toolkit**: Diese Sammlung an verschiedenen Blibliotheken bzw. Applikationen enthalten verschiedenste Möglichkeiten zur Aufnahme von Sensordaten um damit eine Karte aufzubauen. In diesem Paket sind nicht nur SLAM Algorithmen zu finden, sondern auch andere Applikationen wie aus der Dokumentation des MRPT zu entnehmen ist. Dabei wurden Folgende genauer untersucht:
  http://www.mrpt.org/
  - ICP-SLAM: Dieser Ansatz verwendet nur reine Lasersensordaten um eine entprechende Karte aufzubauen. Dabei wird ein ICP Algorithmus verwendet, der

in diesem Toolkit vorhanden ist. Mit diesem Algorithmus konnten jeweils einige Tests gefahren werden. Da in diesem Fall viele Abhängigkeiten zum ganzen MRPT Toolkit bestehen ist eine genauere Analyse der gesamtem beteiligten Pakete notwendig, um eine Anpassung bzw. Erweiterung des gesamten Algorithmus' zu erreichen. Auf einem Notebook konnten noch sehr gute Ergebnisse, sowohl was die Karte und deren Anzeige, als auch was die benötigte Rechenleistung betrifft, erreicht werden. Ein Port auf den Raspberry Pi 2 ergab dann allerdings ein anderes Bild. Sobald der Algorithmus gestartet wurde, war der Raspberry Pi 2 fast komplett ausgelastet (ca. 300% bei jeweils 100% pro Kern zur Verfügung stehende Leistung). Ohne eine weitere Software/Hardware Anpassung ist somit dieser Ansatz mobil direkt auf dem ALF nicht zu gebrauchen.

  - Alle anderen Algorithmen, die im Toolkit zur Verfügung standen, wurden nur vereinzelt noch auf ihre theoretische Einsetzmöglichkeit hin untersucht. Da allerdings bereits beim ICP-SLAM die komplette Rechenleistung des Raspberry Pi genutzt wurde, wird das bei all diesen Algorithmen des MRPT ein Problem sein (Das wurde allerdings nicht verifiziert).

- **libICP**: Die vom Max-Planck Institut für intelligente Systeme in Tübingen entwickelte und zur Verfügung gestellte libICP konnte auch getestet werden. Dabei ist dieser Algorithmus ein reiner ICP und damit grundsätzlich für die Kartenerstellung geeignet. Für die Positionsbestimmung innerhalb der Karte müsste dieser Ansatz weiter angepasst werden. Da dieses Paket wenig Abhängigkeiten benötigt und einfach zu erweitern und anzupassen ist wurde dieser Algorithmus auch praktisch untersucht (Eine graphische Anzeige ist allerdings ohne Matlab im Moment nicht möglich). Allerdings zeigte sich hier, dass die benötigte Rechenzeit für eine kleine Karte schon viel zu hoch ist. Für eine größere Karte, die einigermaßen in Echtzeit aktualisiert werden soll, werden somit die Ressourcen nicht ausreichen. Deshalb wird auch für diesen Ansatz eine leistugsfähigere Hardware als der Raspberry Pi (2) benötigt.
  http://www.cvlibs.net/software/libicp/

- **hectorMapping**: Der bereits vom ROS auf der VM eingesetzte hectorMapping ist für eine Portierung auf einen Raspberry Pi nicht geeignet, da das komplette ROS-Paket und dessen Abhängigkeiten benötigt werden. Außerdem zeigt sich bereits auf der VM, dass dieser Ansatz mit der Rechenleistung des Raspberry Pi 2 nicht möglich ist. Damit wurde dieser Ansatz nicht mehr weiter verfolgt.

- **gMapping**: Auch gMapping kann mit ROS eingesetzt werden. Da dies nicht gewünscht ist konnte noch ein weiteres Paket gefunden werden, dass kein ROS benötigt. Aber auch bei diesem Paket werden einige Abhängigkteien benötigt, die eine weitere Anpassung nicht ohne weiteres möglich machen. Aus diesem Grund ist auch eine genauere Einarbeitung in dieses Paket erfoderlich, um weitere Aussagen treffen zu können. Dieser Ansatz kann sowohl Lidar, als auch Odometriedaten verwerten,

um eine Karte aufzubauen.
https://www.openslam.org/gmapping.html

- **breezySLAM**: Dieser auf TinySLAM aufbauende Ansatz ist etwas komplexer, dafür aber deutlich flexibler als TinySLAM alleine. Dieser Algorithmus ermöglicht zusätzlich zu den Lidardaten auch Odometriedaten für die Berechung einer Position und Erstellung einer Karte mit einzubeziehen. Der Aufbau der Karte wird durch die Komponenten des TinySLAM erledigt, während die Positionsbestimmung innerhalb der Karte durch verschiedene Filter ermöglicht wird. Solche Filter können auch einfach in die bestehende Umgebung integriert werden, wenn das nötig sein sollte. Die Anwendung ist darauf ausgelegt möglichst einfach aufgebaut zu sein und ermöglicht mittels einer einzigen C++ Klasse alle nötigen Informationen aus diesem Algorithmus einzuspeisen (alle LIDAR Daten) und wieder zur Verfügung zu stellen (Karte + Position). Leider konnte nicht mehr untersucht werden, wie genau die Lidardaten aussehen müssen, um mit diesem Algorithmus kompatibel zu sein.
https://github.com/simondlevy/BreezySLAM

Für alle diese Algorithmen gilt allerdings, dass nur wenig Zeit zur Verfügung stand, um eine vollständige Analyse des jeweiligen Ansatzes zu ermöglichen. Dementsprechend sind weitere Analysen notwendig um eine genauere Einschätzung zu geben.

3.3. Zusammenfassung

Wie bereits erwähnt, müssen für eine endgültige Entscheidung noch weitere Analysen, der verschiedenen vorgestellten Algorithmen erledigt werden. Allerdings soll hier trotzdem eine erste Einschätzung hinsichtlich der Möglichkeiten der verschiedenen Ansätze abgegeben werden.

Bleibt man bei der aktuellen Hardware und hat damit sehr wenig Ressourcen für die komplette Berechnung der Karte und der aktuellen Position zur Verfügung, so wird der TinySLAM wahrscheinlich die beste Alternative darstellen. Aufgrund der wenigen Berechnungen, die hier durchgeführt werden könnten hier die Leistungsreserven noch am eheseten ausreichen.

Besteht allerdings die Möglichkeit auf eine andere Hardware auszuweichen, oder einen zusätzlichen Raspberry Pi bzw. stärkere Hardware zu installieren ergeben sich andere Möglichkeiten. Dabei muss vor allem beachtet werden, was für die Applikation schlussendlich am Wichtigsten ist. Eine **einfache Anpassung** wird sicherlich am besten von **breezySLAM, libICP und TinySLAM** ermöglicht. Die **flexibelste Lösung** für die aktuelle Situation ist wahrscheinlich der **breezySLAM**. Allerdings stellt das **Mobile Robot Programming Toolkit** wohl die **größtmöglichen Einsatzmöglichkeiten** bereit, die eventuell ganz ohne Anpassungen eingesetzt werden können. Dabei kann allerdings - durch die vielen Abhängikeiten - wieder das gleiche Problem wie bei einem System mit ROS entstehen.

4. Fazit und Ausblick

4.1. Fazit

Im Rahmen dieses HSP-Projekts konnte ein ROS-Knoten des ALF erfolgreich von ROS wegportiert werden. Diese Portierung hat den am Projektanfang gewünschten und erhofften Performancevorteil ohne nennenswerte Nachteile gebracht. Durch den Einsatz eines Wrappers zum ROS-Projekt und zu der graphischen Anzeige mittels RVIZ konnte eine schnelle visuelle Verifikation der Daten vorgenommen werden. Vor allem die Einarbeitung in die ROS-Umgebung und den richtigen Einstiegspunkt zu finden fiel uns schwer. Der eigentliche Implementierungsaufwand war für dieses Teilthema dann überschaubar. Durch eine vernünftige Dokumentation können zukünftige Projekte am ALF mit Hilfe der geschaffenen Infrastruktur schnell starten.

Der Vergleich der SLAM-Algorithem fördert zwei Ergebnisse zutage. Zum einen erfordert bereits die Auswahl eines geeigneten Algorithmus jede Menge Aufwand, was für uns innerhalb des einen verbliebenden Monats nicht zufriedenstellend gelöst werden konnte. Zum anderen sind die Algorithmen händisch aufgrund der dahintersteckenden Mathematik schwer zu implementieren oder vorhandener Quellcode ist unzureichend dokumentiert um es auf unsere Situation anzupassen oder in einen Softwareframework eingebettet, der Anpassungen sehr schwer macht und die wiederrum eine komplexe Einarbeitung erfordert. Für eine Projektarbeit, die sich mit SLAM am ALF beschäftigt, sollte ein ganzes HSP einkalkuliert werden.

4.2. Ausblick

Für die weitere Arbeit am ALF können nun verschiedene Wege eingeschlagen werden. Zum einen könnte man sich bemühen, alle noch verbliebenen ROS Abhängigkeiten zu entfernen um damit weitere Ressourcen freizulegen. Dies wäre nötig, da im Moment auch noch das ferngesteuerte Fahren einen ROS Knoten benötigt. Es gibt allerdings auch eine weitere Alternative: Man könnte für das ganze System eine neue Hardwaregrundlage schaffen, um damit deutlich flexibler und leistungsfähiger zu werden.

Der nächste Schritt für das autonome Fahren muss sicherlich sein, dass auf der zur Verfügung stehenden Hardware (egal ob PI oder andere Hardware) ein geeigneter Algorithmus zur Kartenerstellung und Positionsbestimmung implemntiert werden muss. Um dann ein einem weiteren Schritt eine passende Wegberechnung innerhalb dieser Karte zu ermöglichen.

Abbildungsverzeichnis

Abkürzungsverzeichnis

ALF	Autonomes Laser Fahrzeug
HAL	Hardware Abstraction Layer
HSP	Hauptseminar Projektstudium
Lidar	Light detection and ranging
ROS	Robot Operating System
RVIZ	ROS Visualization
SLAM	Simultaneous Localization and Mapping
SoC	System-on-a-Chip

Literaturverzeichnis

[1] Tully Foote. “tf: The transform library”. In: *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*. Open-Source Software workshop. Apr. 2013, S. 1–6. DOI: 10.1109/TePRA.2013.6556373.

Anhang

A.

sensor_msgs/LaserScan Message

File: sensor_msgs/LaserScan.msg

Raw Message Definition

```
# Single scan from a planar laser range-finder
#
# If you have another ranging device with different behavior (e.g. a sonar
# array), please find or create a different message, since applications
# will make fairly laser-specific assumptions about this data

Header header            # timestamp in the header is the acquisition time of
                         # the first ray in the scan.
                         #
                         # in frame frame_id, angles are measured around
                         # the positive Z axis (counterclockwise, if Z is up)
                         # with zero angle being forward along the x axis

float32 angle_min        # start angle of the scan [rad]
float32 angle_max        # end angle of the scan [rad]
float32 angle_increment  # angular distance between measurements [rad]

float32 time_increment   # time between measurements [seconds] - if your scanner
                         # is moving, this will be used in interpolating position
                         # of 3d points
float32 scan_time        # time between scans [seconds]

float32 range_min        # minimum range value [m]
float32 range_max        # maximum range value [m]

float32[] ranges         # range data [m] (Note: values < range_min or > range_max should be discarded)
float32[] intensities    # intensity data [device-specific units].  If your
                         # device does not provide intensities, please leave
                         # the array empty.
```

Compact Message Definition

```
std_msgs/Header header
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

autogenerated on Sun, 02 Oct 2016 03:26:30

B.

- /
  - `alf` ..... ROS alf package
  - `attiny45` ..... Attiny 45 Software für Ultraschallsensoren
  - `Code` ..... Software Projektverzeichnis
    - `alf_urg` ..... Verzeichnis für alf_urg Komponente
    - `alf_urg_Debug` ..... Debug Verzeichnis von alf_urg
    - `alf_urg_for_Raspi` ... Alf_urg Verzeichnis für Crosscompilierte Komponente
    - `common` ..... Common Verzeichnis mit *alf_erno* und *alf_log*
    - `Documentation` ..... Doxygen Dokumentationsverzeichnis
    - `HAL` ..... Projektverzeichnis für HAL Komponente
    - `melmac_Client` ..... Verzeichnis für melmac Debug Client
    - `melmac_Client_Debug` ..... Debug Verzeichnis für melmac_Client
    - `melmac_rviz` ..... Projektverzeichnis des melmac_Clients mit RVIZ Wrapper
    - `Skripts` ..... Python Skripts zur Auswertung und Datenverarbeitung
    - `Alf_Urg_Example.alf` ..... Alf File mit Demodaten I
    - `Einmal_ums_Labor.alf` ..... Alf File mit Demodaten II
  - `documentation_ss2016` ..... Dokumentationsverzeichnis des aktuellen HSP
  - `documentation_WS1516` ..... Dokumentationsverzeichnis des vorherigen HSP
  - `etc` ..... Verzeichnis für diverses
  - `melmac` ..... Verzeichnis für melmac Komponente

C.

Doxygen Dokumentation

Alf

1

Generated by Doxygen 1.8.11

Contents

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

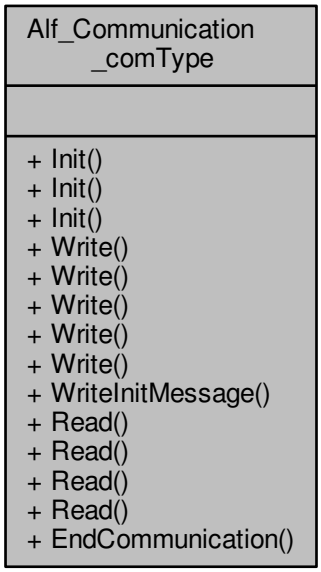
Class Documentation

3.1 Alf_Communication< _comType > Class Template Reference

CommunicationClass that handles all the communication. Possible template parameters are at the moment std::fstream, Client and Server. No other com-types are supported.

```
#include <alf_communication.hpp>
```

Collaboration diagram for Alf_Communication< _comType >:



Public Member Functions

- bool Init (const string &filename)

  Init, for communication as a file.
- bool Init (const string &server, const uint32_t &portno)

  Init, for communication as a client.
- bool Init (const uint32_t &portno)

  Init, for communication as a server.
- bool Write (std::fstream &file, const char *data, const uint32_t &len)

  Writes len bytes from data.
- bool Write (Client &cl, const char *data, const uint32_t &len)
- bool Write (Server &ser, const char *data, const uint32_t &len)
- alf_error Write (Alf_Urg_Measurements_Buffer &buffer)

  This function writes the a buffer to the communication type. Only calling the internal Write(Alf_Urg_Measurement&) function until the buffer is empty.
- alf_error Write (Alf_Urg_Measurement &meas)

  Creates a string with all, for our application, relevant information for one laser-scanner measurement. The structure of this string is described in Alf_Messages.ods, outside this inline documentation.
- alf_error WriteInitMessage ()

  Writes the init message over the choosen communication type with information about the urg sensor.
- bool Read (std::fstream &file, char *readPtr, const uint32_t &len)

  Reads len bytes and stores them into readPtr.
- bool Read (Client &cl, char *readPtr, const uint32_t &len)
- bool Read (Server &ser, char *readPtr, const uint32_t &len)
- alf_error Read (Alf_Urg_Measurements_Buffer &readBuffer, alf_mess_types &msgType, const uint32_t &nr↩PackToRead=1)

  Function reads nrPackToRead Messages and stores them to the readBuffer. If the buffer has not enough free entries, no data is read and nothing is changed. Then another read is possible when the buffer has enough free entries.
- bool EndCommunication (void)

  Function to end the communication.

3.1.1 Detailed Description

template<class _comType>
class Alf_Communication< _comType >

CommunicationClass that handles all the communication. Possible template parameters are at the moment std↩::fstream, Client and Server. No other com-types are supported.

Definition at line 152 of file alf_communication.hpp.

3.1.2 Member Function Documentation

3.1.2.1 template<class _comType > bool Alf_Communication< _comType >::EndCommunication (void)

Function to end the communication.

Returns

- true if everything works, false otherwise

Definition at line 259 of file alf_communication.tpp.

Here is the caller graph for this function:

3.1.2.2 template<class _comType > bool Alf_Communication< _comType >::Init (const string & *filename*)

Init, for communication as a file.

Parameters

in	*filename*	- for the file, which will be used as communication

Returns

true when everything fine, false otherwise

Definition at line 26 of file alf_communication.tpp.

Here is the caller graph for this function:

3.1.2.3 template<class _comType > bool Alf_Communication< _comType >::Init (const string & *server,* const uint32_t & *portno*)

Init, for communication as a client.

Parameters

in	*server*	- the server IP as a string for the connection
in	*portno*	- the portnumber for the communication

Returns

true when everything fine, false otherwise

Definition at line 39 of file alf_communication.tpp.

3.1.2.4 template<class _comType > bool Alf_Communication< _comType >::Init (const uint32_t & *portno*)

Init, for communication as a server.

Parameters

in	*portno*	- the portnumber for the communication

Returns

true when everything fine, false otherwise

Definition at line 50 of file alf_communication.tpp.

3.1.2.5 template<class _comType > bool Alf_Communication< _comType >::Read (std::fstream & *file,* char * *readPtr,* const uint32_t & *len*)

Reads len bytes and stores them into readPtr.

Parameters

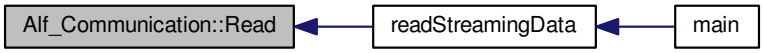
in	*file*	- the fstream from which shall be readed
in,out	*readPtr*	- where the function shall store the readed data
in	*len*	- how much bytes shall be readed

Returns

-

Definition at line 122 of file alf_communication.tpp.

Here is the caller graph for this function:



3.1.2.6 template<class _comType > bool Alf_Communication< _comType >::Read (Client & *cl,* char * *readPtr,* const uint32_t & *len*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Parameters

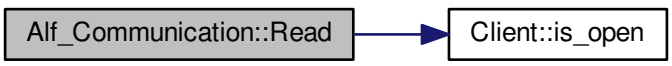
in	*cl*	- the client socket where the data shall be readed
in,out	*readPtr*	- where the function shall store the readed data
in	*len*	- how much bytes shall be readed

Returns

-

Definition at line 134 of file alf_communication.tpp.

Here is the call graph for this function:



3.1.2.7 template<class _comType > bool Alf_Communication< _comType >::Read (Server & *ser,* char * *readPtr,* const uint32_t & *len*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Parameters

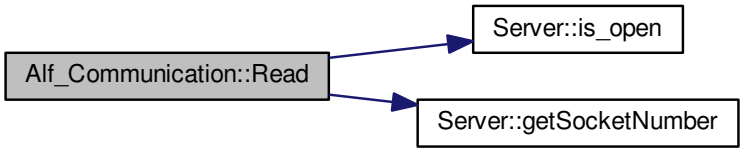
in	*ser*	- the server socket where the data shall be readed
in,out	*readPtr*	- where the function shall store the readed data
in	*len*	- how much bytes shall be readed

Returns

-

Definition at line 145 of file alf_communication.tpp.

Here is the call graph for this function:



3.1.2.8 template<class _comType > alf_error Alf_Communication< _comType >::Read (Alf_Urg_Measurements_Buffer & *readBuffer,* alf_mess_types & *msgType,* const uint32_t & *nrPackToRead* = 1)

Function reads nrPackToRead Messages and stores them to the readBuffer. If the buffer has not enough free entries, no data is read and nothing is changed. Then another read is possible when the buffer has enough free entries.

Parameters

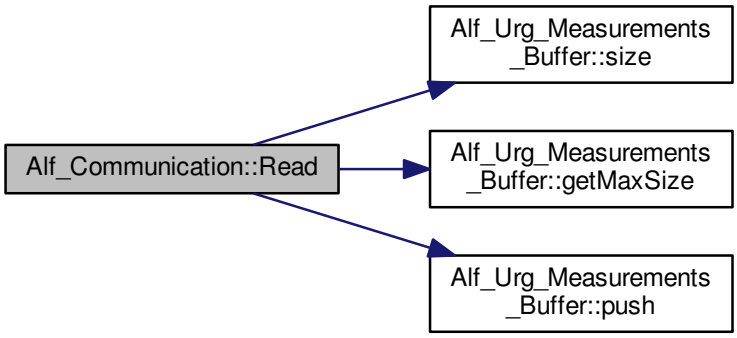
in	*readBuffer*	- This buffer is the memory location for the read data
in	*nrPackToRead*	- default is one packet, otherwise this is the number of packets which will be read

Returns

the first error that occurred or ALF_NO_ERROR when successful

Definition at line 156 of file alf_communication.tpp.

Here is the call graph for this function:



3.1.2.9 template<class _comType > bool Alf_Communication< _comType >::Write (std::fstream & *file,* const char * *data,* const uint32_t & *len*)

Writes len bytes from data.

Parameters

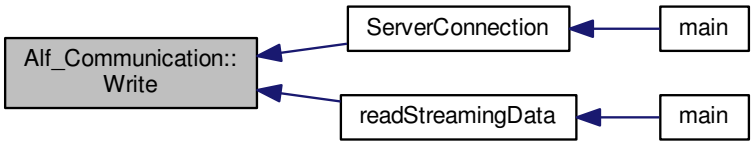
in	*file*	- the fstream, where the bytes should be written
in	*data*	- the pointer to the data which shall be written to the file
in	*len*	- number of bytes from data, which should be written

Returns

- true when everything is fine, false otherwise

Definition at line 60 of file alf_communication.tpp.

Here is the caller graph for this function:



3.1.2.10 template<class _comType > bool Alf_Communication< _comType >::Write (Client & *cl,* const char * *data,* const uint32_t & *len*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Parameters

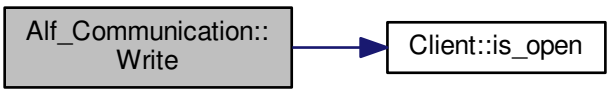
in	*cl*	- the client, writes to a socket
in	*data*	- the pointer to the data which shall be written to the file
in	*len*	- number of bytes from data, which should be written

Returns

- true when everything is fine, false otherwise

Definition at line 70 of file alf_communication.tpp.

Here is the call graph for this function:



3.1.2.11 template<class _comType > bool Alf_Communication< _comType >::Write (Server & *ser,* const char * *data,* const uint32_t & *len*)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Parameters

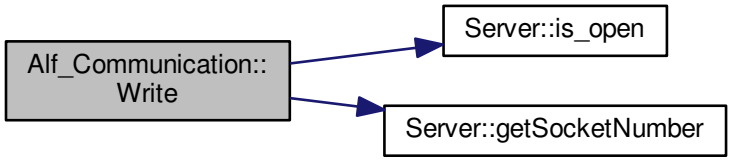
in	*ser*	- the server, writes to a socket
in	*data*	- the pointer to the data which shall be written to the file
in	*len*	- number of bytes from data, which should be written

Returns

- true when everything is fine, false otherwise

Definition at line 80 of file alf_communication.tpp.

Here is the call graph for this function:



3.1.2.12 template<class _comType > alf_error Alf_Communication< _comType >::Write (Alf_Urg_Measurements_Buffer & *buffer*)

This function writes the a buffer to the communication type. Only calling the internal Write(Alf_Urg_Measurement&) function until the buffer is empty.

Parameters

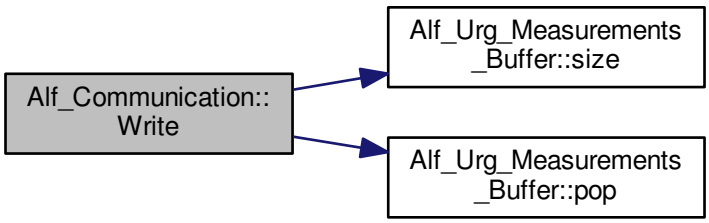
in,out	*buffer*	- the queue which includes all of the measurmenets which was taken to the moment, the function is called. It will be changed on calling this function.

Returns

- alf_error code

Definition at line 90 of file alf_communication.tpp.

Here is the call graph for this function:



3.1.2.13 template<class _comType > alf_error Alf_Communication< _comType >::Write (Alf_Urg_Measurement & *meas*)

Creates a string with all, for our application, relevant information for one laser-scanner measurement. The structure of this string is described in Alf_Messages.ods, outside this inline documentation.

Parameters

in	*meas*	- one laser scanner measurement

Returns

Definition at line 103 of file alf_communication.tpp.

3.1.2.14 template<class _comType > alf_error Alf_Communication< _comType >::WriteInitMessage ( )

Writes the init message over the choosen communication type with information about the urg sensor.

Returns

- ALF_NO_ERROR if all is ok and it works
- ALF_CANNOT_SEND_MESSAGE if the communication does not work

Definition at line 273 of file alf_communication.tpp.

Here is the caller graph for this function:

The documentation for this class was generated from the following files:

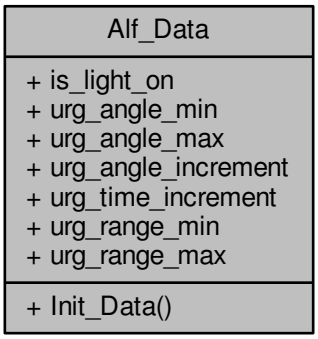
- alf_communication.hpp
- alf_communication.tpp

3.2 Alf_Data Class Reference

contains all the data about the vehicle which could be exchanges between the vehicle and other applications so serves as interface between a controller and the hardware

`#include <alf_data.hpp>`

Collaboration diagram for Alf_Data:



Static Public Member Functions

- static bool Init_Data (float, float, float, int32_t, int32_t, int32_t)

  initialise the Alf_Data

Static Public Attributes

- static bool is_light_on = false

  are the lights on?
- static float urg_angle_min = 0.0

  the min angle which the urg laser scanner can provide
- static float urg_angle_max = 0.0

  the max angle which the urg laser scanner can provide
- static float urg_angle_increment = 0

  the increment between two measurments of the laser scanner
- static int urg_time_increment = 100

  the time between two measurements of the laser scanner in ms, with our laser scanner this is 100ms
- static uint32_t urg_range_min = 0

  the minimal distance the laser scanner can measure
- static uint32_t urg_range_max = 0

  the maximal distance the laser scanner can measure

3.2.1 Detailed Description

contains all the data about the vehicle which could be exchanges between the vehicle and other applications so serves as interface between a controller and the hardware

Definition at line 29 of file alf_data.hpp.

The documentation for this class was generated from the following files:

- alf_data.hpp
- alf_data.cpp

3.3 Alf_Log Class Reference

This class handle all the log informations. There will be always a log file, additional the log can be printed to standard output.

`#include <alf_log.hpp>`

Collaboration diagram for Alf_Log:

Static Public Member Functions

- static bool alf_log_init (const std::string &filename="dummy.alf_log", const alf_log_level_e &log_level=log_debug, const bool &console_output=false)

  Initialize the logging functionality (performed with a file)
- static bool alf_log_write (const std::string &log_entry, const alf_log_level_e &log_level=log_debug)

  Writes a log entry.
- static bool alf_log_end (void)

  close the logging
- static void alf_set_loglevel (const alf_log_level_e &log_level)

  Set the log level.

3.3.1 Detailed Description

This class handle all the log informations. There will be always a log file, additional the log can be printed to standard output.

Definition at line 45 of file alf_log.hpp.

3.3.2 Member Function Documentation

3.3.2.1 bool Alf_Log::alf_log_end (void) [static]

close the logging

Parameters

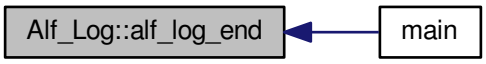
in	-	

Returns

true if successful otherwise false

Definition at line 52 of file alf_log.cpp.

Here is the caller graph for this function:



3.3.2.2 bool Alf_Log::alf_log_init (const std::string & *filename* = "dummy.alf_log", const alf_log_level_e & *log_level* = log_debug, const bool & *console_output* = false) [static]

Initialize the logging functionality (performed with a file)

Parameters

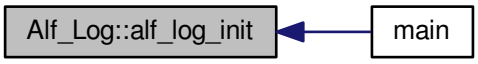
in	*filename*	Path to File
in	*loglevel*	All Messages with level above will be logged
in	*consoleoutput*	If true all messages will be printed on console ouptut

Returns

true if successful otherwise false

Definition at line 28 of file alf_log.cpp.

Here is the caller graph for this function:



3.3.2.3 bool Alf_Log::alf_log_write (const std::string & *log_entry*, const alf_log_level_e & *log_level* = log_debug) [static]

Writes a log entry.

Parameters

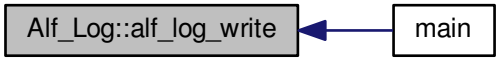
in	*log_entry*	the message to be logged
in	*log_level*	the significance of the message

Returns

true if successful otherwise false

Definition at line 63 of file alf_log.cpp.

Here is the caller graph for this function:



3.3.2.4 void Alf_Log::alf_set_loglevel (const alf_log_level_e & *log_level*) [static]

Set the log level.

Parameters

in	*log_level*	which messages should be logged from now on

Returns

-

Definition at line 93 of file alf_log.cpp.

The documentation for this class was generated from the following files:

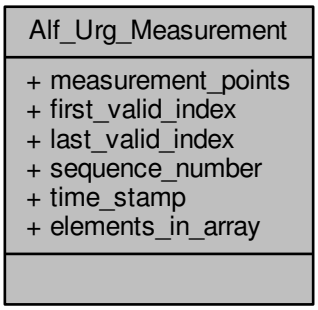
- alf_log.hpp
- alf_log.cpp

3.4 Alf_Urg_Measurement Class Reference

This class stands for **one** whole measurement of the laser scanner and provides additional informations It contains all measurement values, also this one, which are invalid in case of the datasheet.

`#include <alf_data.hpp>`

Collaboration diagram for Alf_Urg_Measurement:



Public Attributes

- long int measurement_points [elements_in_array]

  The storage for the measurement points. Each index represents one urg_angle_increment.
- uint32_t first_valid_index

  The first index of the measurement_points which should be used (derived from the data sheet)
- uint32_t last_valid_index

  The last index of the measurement_points which should be used.
- uint32_t sequence_number

  To provide a chronological sequence of the various measurements.
- long int time_stamp

  The timestamp of the measurement. Its no absolut time, just the internal counter, so several measurements can be set in an chronologically relation.

Static Public Attributes

- static constexpr uint32_t elements_in_array = URG_NUMBER_OF_MEASUREMENT_DATA + 1

  how much measurement points do we have for one measurement

3.4.1 Detailed Description

This class stands for **one** whole measurement of the laser scanner and provides additional informations It contains all measurement values, also this one, which are invalid in case of the datasheet.

Definition at line 56 of file alf_data.hpp.

The documentation for this class was generated from the following file:

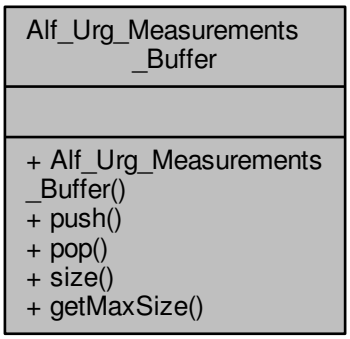
- alf_data.hpp

3.5 Alf_Urg_Measurements_Buffer Class Reference

This buffer can store a set of Alf_Urg_Measurement . It use the std::queue for storing the data and have a maximum size to determine the maximum RAM size which can be used.

`#include <alf_data.hpp>`

Collaboration diagram for Alf_Urg_Measurements_Buffer:



Public Member Functions

- Alf_Urg_Measurements_Buffer (uint32_t size=MAX_SIZE_OF_MEASUREMENT_BUFFER_DEFAULT)

  constructor for the Alf_Urg_Measurement_Buffer set _max_size to the given value or default to the macro MAX_S↩ IZE_OF_MEASUREMENT_BUFFER_DEFAULT
- alf_error push (const Alf_Urg_Measurement &)

  append one Alf_Urg_Measurement to the buffer
- alf_error pop (Alf_Urg_Measurement ∗)

  pops one element of the buffer and stores it in the memory given by a pointer
- uint32_t size () const

  returns the actual size of the queue (so how much elements are stored within)
- uint32_t getMaxSize (void) const

  returns the maximal number of elements which could be stored

3.5.1 Detailed Description

This buffer can store a set of Alf_Urg_Measurement . It use the std::queue for storing the data and have a maximum size to determine the maximum RAM size which can be used.

Definition at line 76 of file alf_data.hpp.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 Alf_Urg_Measurements_Buffer::Alf_Urg_Measurements_Buffer (uint32_t *size* = MAX_SIZE_OF_MEASUREMENT_BUFFER_DEFAULT)

constructor for the Alf_Urg_Measurement_Buffer set _max_size to the given value or default to the macro MAX_SIZE_OF_MEASUREMENT_BUFFER_DEFAULT

Parameters

in	*size*	- the size, default MAX_SIZE_OF_MEASUREMENT_BUFFER_DEFAULT

Returns

-

Definition at line 38 of file alf_data.cpp.

3.5.3 Member Function Documentation

3.5.3.1 uint32_t Alf_Urg_Measurements_Buffer::getMaxSize (void) const

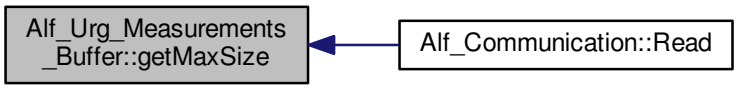
returns the maximal number of elements which could be stored

Returns

the maximal number of elements

Definition at line 71 of file alf_data.cpp.

Here is the caller graph for this function:



3.5.3.2 alf_error Alf_Urg_Measurements_Buffer::pop (Alf_Urg_Measurement * *a*)

pops one element of the buffer and stores it in the memory given by a pointer

Parameters

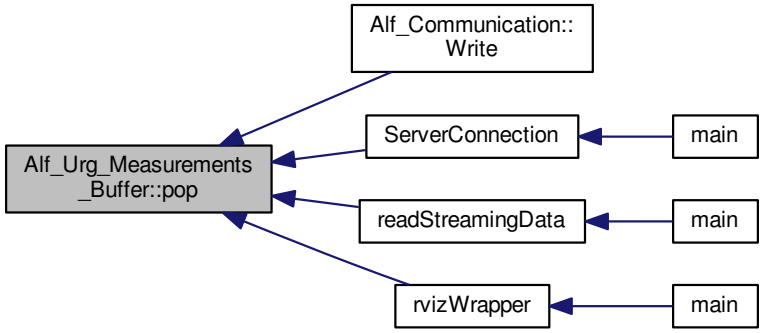
in,out	*a*	- the memory where the Alf_Urg_Measurement shall be stored

Returns

- ALF_NO_ERROR if everything works
- ALF_NOTHING_IN_BUFFER if there is no more element in the queue which could be removed

Definition at line 54 of file alf_data.cpp.

Here is the caller graph for this function:



3.5.3.3 alf_error Alf_Urg_Measurements_Buffer::push (const Alf_Urg_Measurement & *a*)

append one Alf_Urg_Measurement to the buffer

Parameters

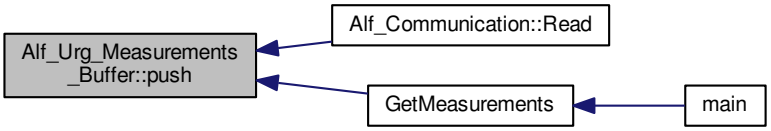
in	*a*	- the measurement

Returns

- ALF_NO_ERROR if the element can be appended to the queue
- ALF_BUFFER_IS_FULL if the queue is full and cannot store any additonal elements

Definition at line 42 of file alf_data.cpp.

Here is the caller graph for this function:



3.5.3.4 uint32_t Alf_Urg_Measurements_Buffer::size ( ) const

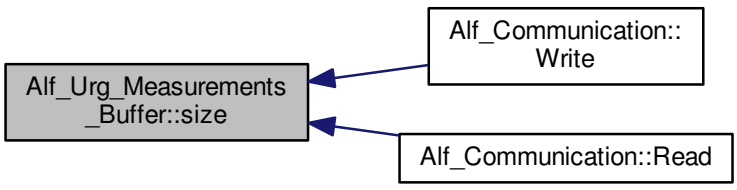
returns the actual size of the queue (so how much elements are stored within)

Returns

the number of elements

Definition at line 67 of file alf_data.cpp.

Here is the caller graph for this function:



The documentation for this class was generated from the following files:

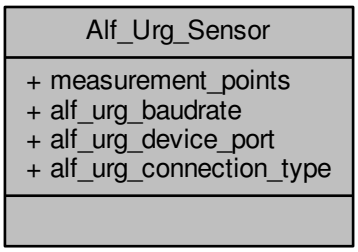
- alf_data.hpp
- alf_data.cpp

3.6 Alf_Urg_Sensor Class Reference

Represents the laser scanner on the alf vehicle and provide common settings etc.

`#include <alf_sensors.hpp>`

Collaboration diagram for Alf_Urg_Sensor:



Static Public Attributes

- static const uint16_t measurement_points = 768

  how much measurement points does the sensor have
- static const long alf_urg_baudrate = 115200

  the baudrate to communicate with the scanner
- static const std::string alf_urg_device_port = "/dev/ttyACM0"

  the port on which the scanner is connected with the hardware
- static const urg_connection_type_t alf_urg_connection_type = URG_SERIAL

  which communication type we use

3.6.1 Detailed Description

Represents the laser scanner on the alf vehicle and provide common settings etc.

Attention

this settings are only vaild with the URG-04LX

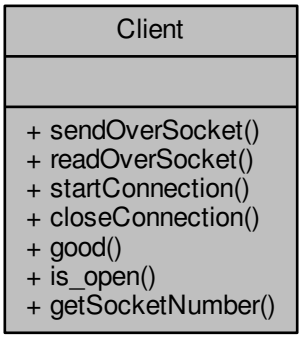
Definition at line 18 of file alf_sensors.hpp.

The documentation for this class was generated from the following files:

- alf_sensors.hpp
- alf_sensors.cpp

3.7 Client Class Reference

Collaboration diagram for Client:



Public Member Functions

- alf_error sendOverSocket (const string &data)

  Sending the string to over the socket via the underlying linux functaion.
- alf_error readOverSocket (string &s)

  reads a string object over the socket. three conditions for read ending are given 1) if the end delimiter is reached ';' 2) no more readable data is available 3) more than 20 characters were read and no delimiter '|' or end delimiter ';' was read
- uint8_t startConnection (const uint32_t &_portno, const string &_server)

  starts the socket connection
- void closeConnection (void)

  closes the connection, communication is no longer possible
- bool good ()

  dummy function to satisfy the compiler (std::fstream, Server/Client all have the good() function so no explicit type handling must be done
- bool is_open ()

  returns the state of the socket connection
- int32_t **getSocketNumber** (void)

3.7.1 Detailed Description

Definition at line 31 of file alf_communication.hpp.

3.7.2 Member Function Documentation

3.7.2.1 bool Client::is_open ( ) [inline]

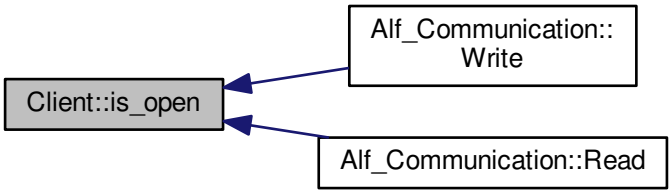
returns the state of the socket connection

Returns

true if connection is good, false otherwise

Definition at line 73 of file alf_communication.hpp.

Here is the caller graph for this function:



3.7.2.2 alf_error Client::readOverSocket (string & *s*)

reads a string object over the socket. three conditions for read ending are given 1) if the end delimiter is reached ';' 2) no more readable data is available 3) more than 20 characters were read and no delimiter '|' or end delimiter ';' was read

Parameters

in	*the*	string data object were the read data is stored (no appending string gets overwritten)

Returns

- ALF_NO_ERROR if the read works
- ALF_CANNOT_READ_SOCKET if it does not work

Definition at line 58 of file alf_communication.cpp.

3.7.2.3 alf_error Client::sendOverSocket (const string & *data*)

Sending the string to over the socket via the underlying linux functaion.

Parameters

in	*data*	- the string with the message which shall be transmitted

Returns

- ALF_NO_ERROR if the message can be transmitted
- ALF_SOCKET_NOT_READY if the socket is not initialised and
- ALF_CANNOT_SEND_MESSAGE if there are errors in the linux functionalitys, typical triggered by a too long message etc.

Definition at line 43 of file alf_communication.cpp.

3.7.2.4 uint8_t Client::startConnection (const uint32_t & *_portno,* const string & *_server*)

starts the socket connection

Parameters

in	*the*	portnumber
in	*servername*	

Returns

1 if successful otherwise error < 0

Definition at line 18 of file alf_communication.cpp.

The documentation for this class was generated from the following files:

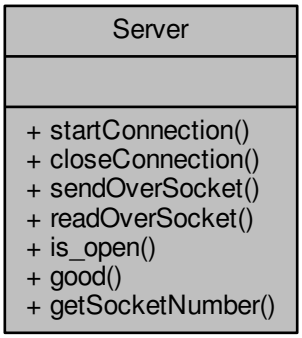
- alf_communication.hpp
- alf_communication.cpp

3.8 Server Class Reference

Represents the serverside of an communication for the whole application.

```
#include <alf_communication.hpp>
```

Collaboration diagram for Server:



Public Member Functions

- alf_error startConnection (const uint32_t &)

  Trys to open the given port and listen to incoming connections It is using the underlying linux functions for socket handling.
- void closeConnection (void)

  Closing the binded socket and close the server connection.
- alf_error sendOverSocket (const string &)

  Sending the string to over the socket via the underlying linux functaion.
- alf_error readOverSocket (string &s)

  read from the underlying socket
- bool is_open ()

  returns the state of the socket connection
- bool good ()

  dummy function to satisfy the compiler (std::fstream, Server/Client all have the good() function so no explicit type handling must be done
- int32_t getSocketNumber (void)

  returns the socket handler id given from linux at initalisation of the socket

3.8.1 Detailed Description

Represents the serverside of an communication for the whole application.

Attention

at the moment this server implementation can only handle **ONE** connection!

Definition at line 89 of file alf_communication.hpp.

3.8.2 Member Function Documentation

3.8.2.1 int32_t Server::getSocketNumber (void) [inline]

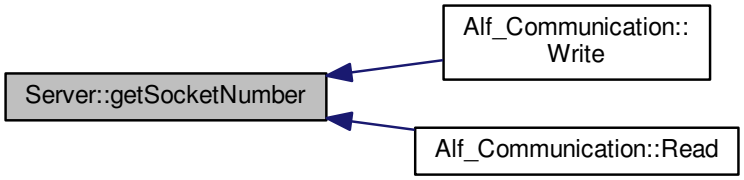
returns the socket handler id given from linux at initalisation of the socket

Returns

the socket handler number

Definition at line 136 of file alf_communication.hpp.

Here is the caller graph for this function:



3.8.2.2 bool Server::is_open ( ) [inline]

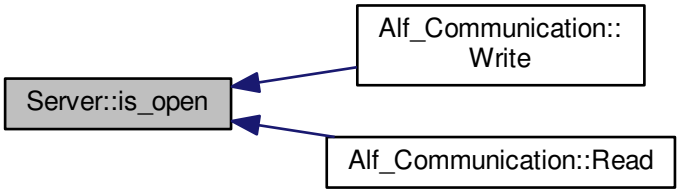
returns the state of the socket connection

Returns

true if connection is good, false otherwise

Definition at line 126 of file alf_communication.hpp.

Here is the caller graph for this function:



3.8.2.3 alf_error Server::readOverSocket (string & *s*)

read from the underlying socket

Parameters

in	*s*	- a string reference

Returns

at this moment -> nothing

Attention

this is just the dummy function, the implementation of this function is missing

3.8.2.4 alf_error Server::sendOverSocket (const string & *data*)

Sending the string to over the socket via the underlying linux functaion.

Parameters

in	*data*	- the string with the message which shall be transmitted

Returns

- ALF_NO_ERROR if the message can be transmitted
- ALF_SOCKET_NOT_READY if the socket is not initialised and
- ALF_CANNOT_SEND_MESSAGE if there are errors in the linux functionalitys, typical triggered by a too long message etc.

Definition at line 131 of file alf_communication.cpp.

3.8.2.5 alf_error Server::startConnection (const uint32_t & *portno*)

Trys to open the given port and listen to incoming connections It is using the underlying linux functions for socket handling.

Parameters

in	*portno*	- the portnumber on which the socket should be opened

Returns

- ALF_SOCKET_SERVER_NOT_READY if something goes wrong (the port is blocked, the function gets no socket handler from os etc.) and
- ALF_NO_ERROR if the port can be catched and the port is working

Definition at line 87 of file alf_communication.cpp.

The documentation for this class was generated from the following files:

- alf_communication.hpp
- alf_communication.cpp

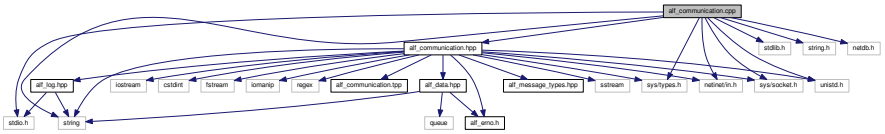
Chapter 4

File Documentation

4.1 alf_communication.cpp File Reference

```
#include "alf_communication.hpp"
#include <string>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
#include <netinet/in.h>
#include <netdb.h>
```

Include dependency graph for alf_communication.cpp:

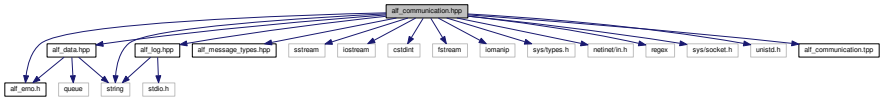


4.2 alf_communication.hpp File Reference

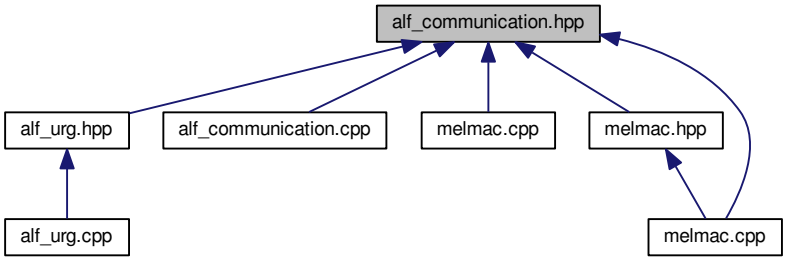
a library for handling all the communication between a client and a server. This file contains all types of communications like writing to files or socket communication over LAN

```
#include "alf_data.hpp"
#include "alf_log.hpp"
#include "alf_erno.h"
#include "alf_message_types.hpp"
#include <string>
#include <sstream>
#include <iostream>
#include <cstdint>
#include <fstream>
#include <iomanip>
#include <sys/types.h>
#include <netinet/in.h>
#include <regex>
#include <sys/socket.h>
#include <unistd.h>
#include "alf_communication.tpp"
```

Include dependency graph for alf_communication.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class Client
- class Server

  Represents the serverside of an communication for the whole application.
- class Alf_Communication< _comType >

  CommunicationClass that handles all the communication. Possible template parameters are at the moment std::fstream, Client and Server. No other com-types are supported.

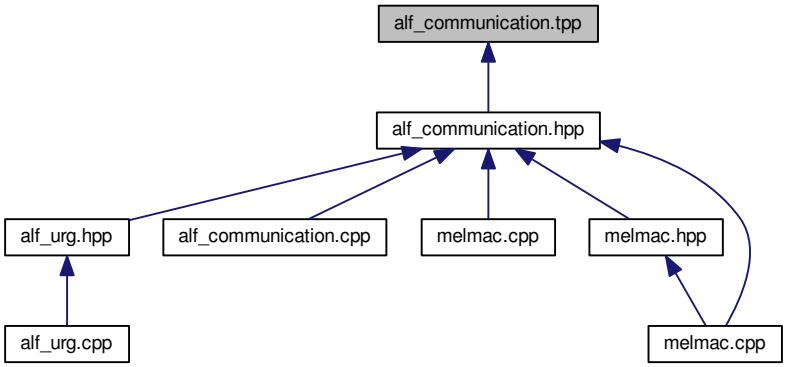
4.2.1 Detailed Description

a library for handling all the communication between a client and a server. This file contains all types of communications like writing to files or socket communication over LAN

4.3 alf_communication.tpp File Reference

contains the implementations for template functions to be outside of the hpp

This graph shows which files directly or indirectly include this file:

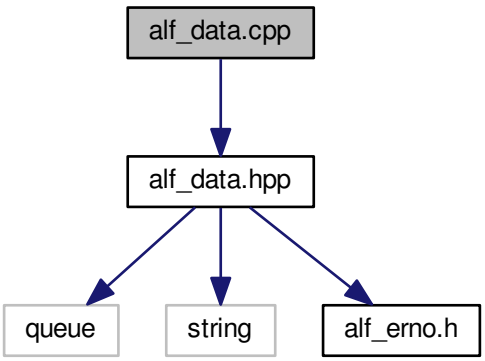


4.3.1 Detailed Description

contains the implementations for template functions to be outside of the hpp

4.4 alf_data.cpp File Reference

```
#include "alf_data.hpp"
```
Include dependency graph for alf_data.cpp:

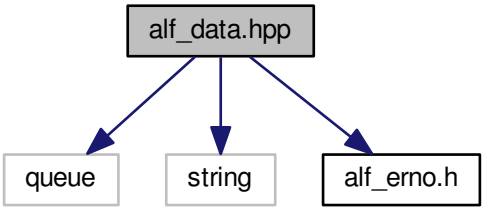


4.5 alf_data.hpp File Reference

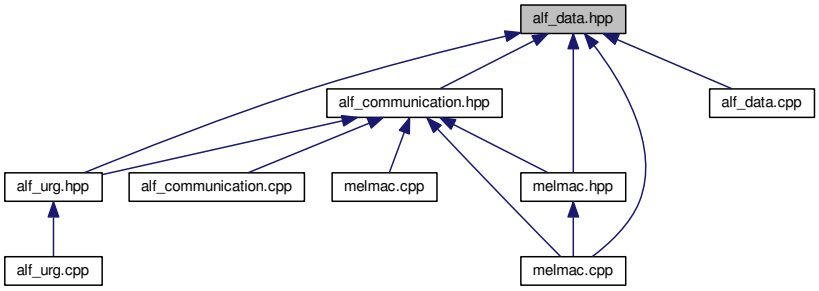
a library for collect all classes which represents any physical data

```
#include <queue>
#include <string>
#include "alf_erno.h"
```

Include dependency graph for alf_data.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class Alf_Data

  contains all the data about the vehicle which could be exchanges between the vehicle and other applications so serves as interface between a controller and the hardware

- class Alf_Urg_Measurement

  *This class stands for **one** whole measurement of the laser scanner and provides additional informations It contains all measurement values, also this one, which are invalid in case of the datasheet.*

- class Alf_Urg_Measurements_Buffer

  This buffer can store a set of Alf_Urg_Measurement . It use the std::queue for storing the data and have a maximum size to determine the maximum RAM size which can be used.

Macros

- #define MAX_SIZE_OF_MEASUREMENT_BUFFER_DEFAULT 10

  the number of elements the measurement buffer can store by default.
- #define URG_NUMBER_OF_MEASUREMENT_DATA 768

  number of the measurements the urg_sensors made. These number varies from sensor to sensor, so with another sensor this value must be adjusted

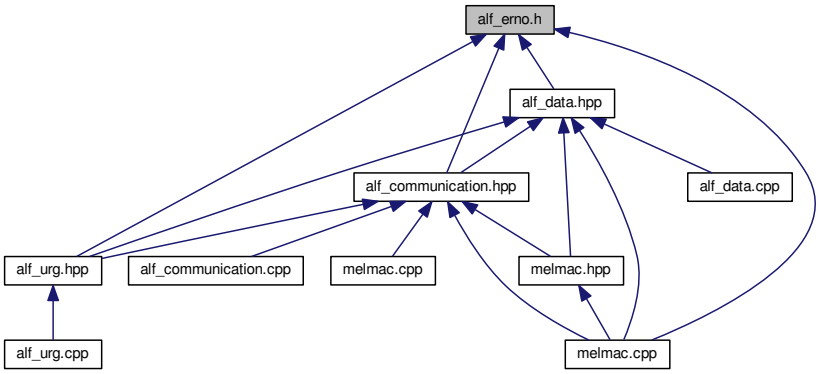
4.5.1 Detailed Description

a library for collect all classes which represents any physical data

4.6 alf_erno.h File Reference

contains various means for error coding

This graph shows which files directly or indirectly include this file:



Typedefs

- typedef enum ALF_ERROR_CODES alf_error

  the error codes are available within a type

Enumerations

- enum ALF_ERROR_CODES {
  ALF_BUFFER_READ_IS_WRITE = -100, **ALF_BUFFER_NOTHING_TO_READ**, **ALF_BUFFER_IS_FULL**, **ALF_NOTHING_IN_BUFFER**,
  ALF_NO_COMMUNICATION_FILE, **ALF_IO_ERROR**, **ALF_SOCKET_NOT_READY**, ALF_SOCKET_SERVER_NOT_READY,
  ALF_CANNOT_SEND_MESSAGE, **ALF_CANNOT_READ_SOCKET**, **ALF_UNKNOWN_ERROR** = -1, ALF_NO_ERROR = 1 }

  contains error codes for all errors which could occur during execution of the application and the information could be interesting for error handling

4.6.1 Detailed Description

contains various means for error coding

4.6.2 Enumeration Type Documentation

4.6.2.1 enum ALF_ERROR_CODES

contains error codes for all errors which could occur during execution of the application and the information could be interesting for error handling

Enumerator

ALF_SOCKET_SERVER_NOT_READY the serverconnection can not be opened, there are some errors in catching the port, opening the file etc.

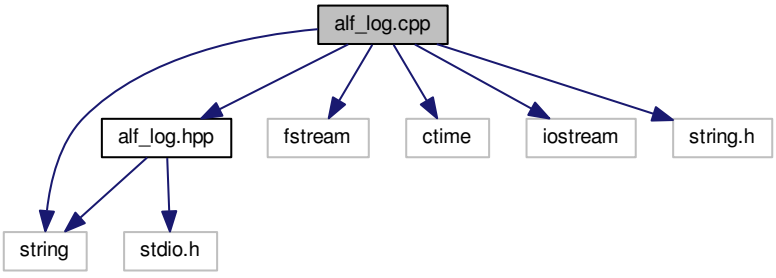
ALF_NO_ERROR alright, there was no error in the functionality

Definition at line 13 of file alf_erno.h.

4.7 alf_log.cpp File Reference

```
#include "alf_log.hpp"
#include <fstream>
#include <ctime>
#include <iostream>
#include <string.h>
#include <string>
```

Include dependency graph for alf_log.cpp:

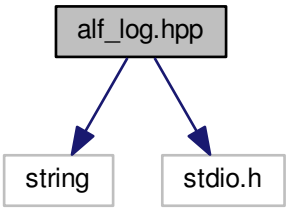


4.8 alf_log.hpp File Reference

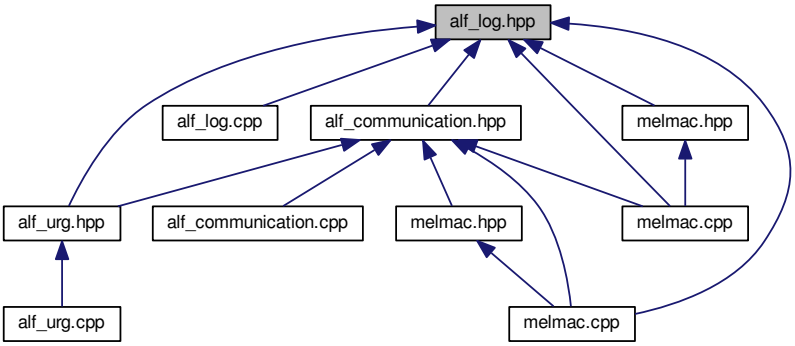
a library give access to log variants and functionality for this

```
#include <string>
#include <stdio.h>
```

Include dependency graph for alf_log.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class Alf_Log

  This class handle all the log informations. There will be always a log file, additional the log can be printed to standard output.

Macros

- #define LOG_ENABLE

  LOG_ENABLE does enabling the log, with LOG_DISABLE there are no further log informations.
- #define **ALF_LOG_INIT**(args...) Alf_Log::alf_log_init(args)
- #define **ALF_LOG_WRITE**(args...) Alf_Log::alf_log_write(args)
- #define **ALF_LOG_END**() Alf_Log::alf_log_end()
- #define **ALF_LOG_SET_LEVEL**(a) ALF_Log::alf_set_loglevel(a)

Enumerations

- enum alf_log_level_e { log_error = 0, log_warning, log_info, log_debug }

  all log leves which are available
  the log levels are based on each other, which means, that every log_error is also a log_warning, log_info, log_debug, but a log_info is no log_warning but a log_debug

4.8.1 Detailed Description

a library give access to log variants and functionality for this

4.8.2 Enumeration Type Documentation

4.8.2.1 enum alf_log_level_e

all log leves which are available
the log levels are based on each other, which means, that every log_error is also a log_warning, log_info, log_debug, but a log_info is no log_warning but a log_debug

Enumerator

log_error strongest error, should be used if the desired function of the application could not be provided

log_warning a warning should be used it the execution of the application is in danger, but it is still running

log_info just for info messages, which could be later used in case of errors or warnings to see the control flow etc.

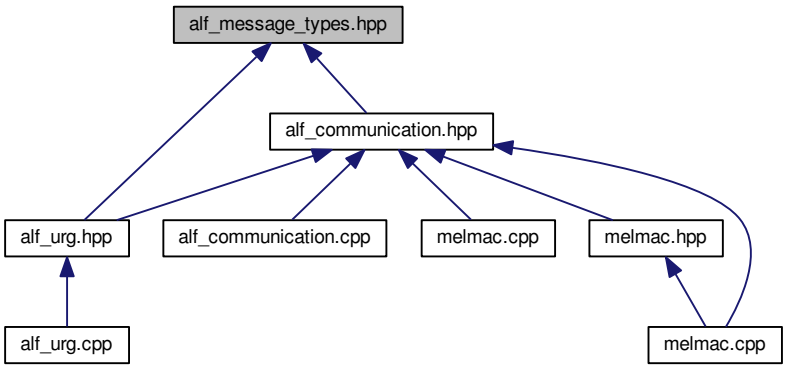
log_debug developer informations

Definition at line 31 of file alf_log.hpp.

4.9 alf_message_types.hpp File Reference

contains enumeration for easy identification of various messages

This graph shows which files directly or indirectly include this file:



Typedefs

- typedef enum ALF_MESSAGE_TYPES **alf_mess_types**

Enumerations

- enum ALF_MESSAGE_TYPES { ALF_INIT_ID = 2, ALF_MEASUREMENT_DATA_ID = 1, ALF_END_ID = 255 }

  contains the IDs for all of the messages which can be sended

4.9.1 Detailed Description

contains enumeration for easy identification of various messages

4.9.2 Enumeration Type Documentation

4.9.2.1 enum ALF_MESSAGE_TYPES

contains the IDs for all of the messages which can be sended

Enumerator

ALF_INIT_ID initalisation data of the laser scanner

ALF_MEASUREMENT_DATA_ID a measurement is sended

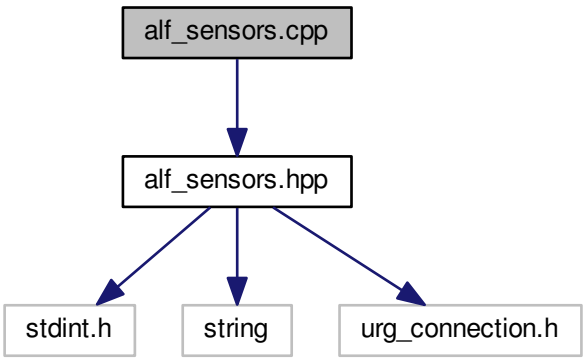
ALF_END_ID the communication should stop or interrupt now

Definition at line 12 of file alf_message_types.hpp.

4.10 alf_sensors.cpp File Reference

```
#include "alf_sensors.hpp"
```

Include dependency graph for alf_sensors.cpp:

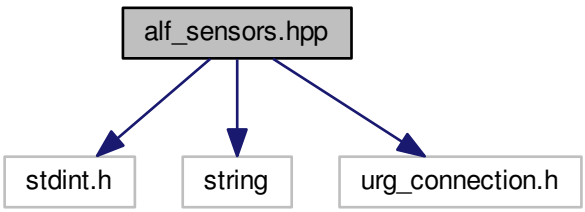


4.11 alf_sensors.hpp File Reference

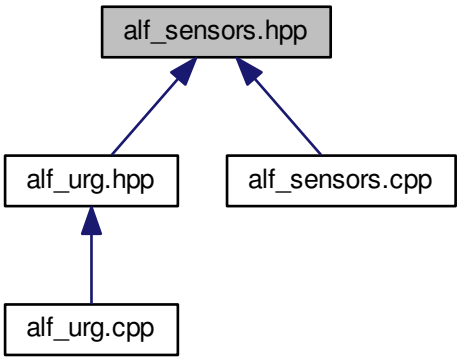
contains datatypes and functionalitys for sensors on the alf vehicle

```
#include <stdint.h>
#include <string>
#include <urg_connection.h>
```

Include dependency graph for alf_sensors.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class Alf_Urg_Sensor

  Represents the laser scanner on the alf vehicle and provide common settings etc.

4.11.1 Detailed Description

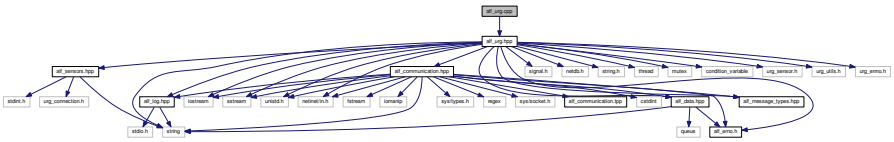
contains datatypes and functionalitys for sensors on the alf vehicle

4.12 alf_urg.cpp File Reference

contains the main application to collect measurements from the URG Lidar and offer the collected data in a properitary format other applications

```
#include "alf_urg.hpp"
```

Include dependency graph for alf_urg.cpp:



Macros

- #define **COMMSERVICE** Server
- #define **COMMFILE** 6666
- #define **msleep**(a) usleep(a∗1000)

Functions

- void GetMeasurements ()

  function for collecting data from a urg_sensor and pushing them into a the Alf_Measurements_Buffer
- void ServerConnection ()

  function for sending collected measurement data over the socket connection
- void Stop_Program (int sig)

  dummy function which wake up the main thread from "sleep". This is needed for a clean stop of the programm with a SIGINT of the OS (typical CTRL+C)
- int main ()

  the main process of this application this does

Variables

- Alf_Urg_Measurements_Buffer Alf_Measurements_Buffer (100)

  the buffer with the Size of 100 for all measurements
- std::mutex Alf_Measurements_Buffer_Mutex

  mutex to lock the Alf_Measurements_Buffer
- urg_t urg_sensor

  struct for the ONE connected sensor
- bool Run_Measurement_Task

  control variable for the thread which collects the measurements
- bool Run_Server_Task

  control variable for the thread which handles the communication
- std::condition_variable Run_Main_Task_cond

  variable to let sleep the main thread
- std::mutex Run_Main_Task_mut

  mutex to for the main thread
- Alf_Communication< COMMSERVICE > server_communication

  the communication which shall be handled

4.12.1 Detailed Description

contains the main application to collect measurements from the URG Lidar and offer the collected data in a properitary format other applications

4.12.2 Function Documentation

4.12.2.1 void GetMeasurements ( ) [inline]

function for collecting data from a urg_sensor and pushing them into a the Alf_Measurements_Buffer

Attention

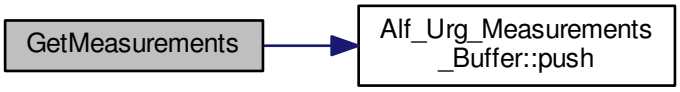
needs a initialized and running urg_sensor, given by the global variable urg_sensor

Note

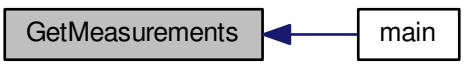
normally executed as a standalone thread/task

Definition at line 41 of file alf_urg.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



4.12.2.2 int main ( )

the main process of this application this does

- initializing the urg_sensor
- initializing the server connection
- starting the two threads
- ending the application in a clean way (after CTRL+C)

Definition at line 134 of file alf_urg.cpp.

4.12.2.3 void ServerConnection ( ) [inline]

function for sending collected measurement data over the socket connection

Attention

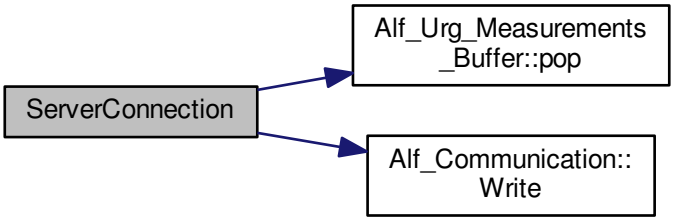
the server connection should be established before calling

Note

normally executed as an own thread

Definition at line 99 of file alf_urg.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

4.12.2.4 void Stop_Program (int *sig*)

dummy function which wake up the main thread from "sleep". This is needed for a clean stop of the programm with a SIGINT of the OS (typical CTRL+C)

Parameters

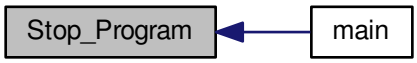
in	*sig*	- SIGINT

Returns

-

Definition at line 122 of file alf_urg.cpp.

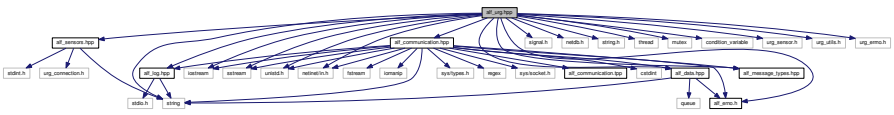
Here is the caller graph for this function:



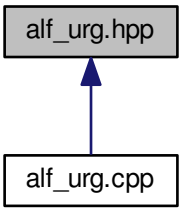
4.13 alf_urg.hpp File Reference

```
#include <iostream>
#include <sstream>
#include <string>
#include <signal.h>
#include <unistd.h>
#include <netdb.h>
#include <netinet/in.h>
#include <string.h>
#include <thread>
#include <mutex>
#include <condition_variable>
#include "alf_log.hpp"
#include "alf_data.hpp"
#include "alf_erno.h"
#include "alf_communication.hpp"
#include "alf_message_types.hpp"
#include "alf_sensors.hpp"
#include "urg_sensor.h"
#include "urg_utils.h"
#include "urg_errno.h"
```

Include dependency graph for alf_urg.hpp:



This graph shows which files directly or indirectly include this file:



Functions

- int main ()
  the main process of this application this does

4.13.1 Function Documentation

4.13.1.1 int main ( )

the main process of this application this does

- initializing the urg_sensor
- initializing the server connection
- starting the two threads
- ending the application in a clean way (after CTRL+C)

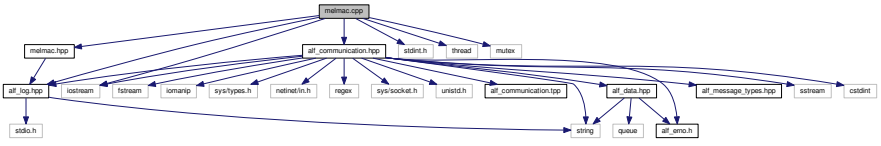
Definition at line 134 of file alf_urg.cpp.

4.14 melmac.cpp File Reference

Test Application to collect data from the server to a remote pc with this application.

```
#include "melmac.hpp"
#include "alf_log.hpp"
#include "alf_communication.hpp"
#include <stdint.h>
#include <thread>
#include <iostream>
#include <mutex>
```

Include dependency graph for melmac_Client/melmac.cpp:



Functions

- void readStreamingData (void)

  function for reading the measurement data from the socket connection. If and end message was read the function returns and the user can end or reopen the communication
- int main ()

  the main process of this application this does

4.14.1 Detailed Description

Test Application to collect data from the server to a remote pc with this application.

4.14.2 Function Documentation

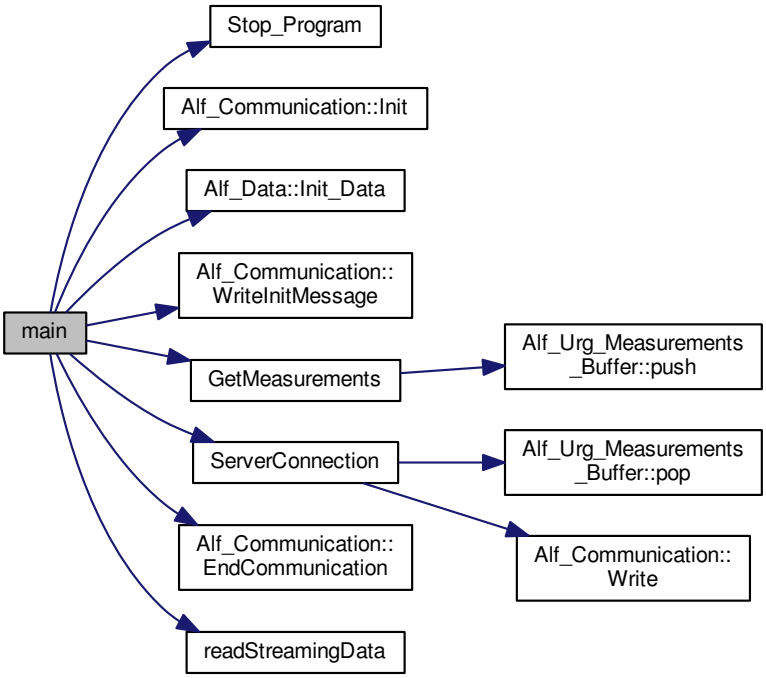
4.14.2.1 int main ( )

the main process of this application this does

- initializing the urg_sensor
- initializing the server connection
- starting the two threads
- ending the application in a clean way (after CTRL+C)

Definition at line 62 of file melmac_Client/melmac.cpp.

Here is the call graph for this function:



4.14.2.2 void readStreamingData (void)

function for reading the measurement data from the socket connection. If and end message was read the function returns and the user can end or reopen the communication

Parameters

in	-	

Returns

-

Definition at line 27 of file melmac_Client/melmac.cpp.

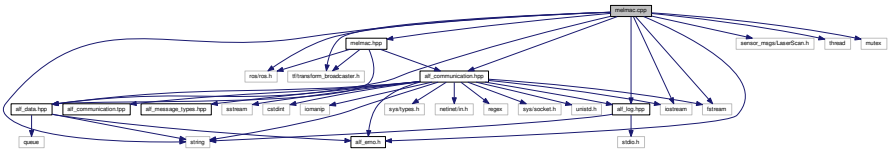
Here is the caller graph for this function:

4.15 melmac.cpp File Reference

contains the main application for wrapping data which are collected with the alf_urg application and sended to this client

```
#include <ros/ros.h>
#include <sensor_msgs/LaserScan.h>
#include <tf/transform_broadcaster.h>
#include <iostream>
#include <string>
#include <fstream>
#include <thread>
#include <mutex>
#include "melmac.hpp"
#include "alf_erno.h"
#include "alf_data.hpp"
#include "alf_log.hpp"
#include "alf_communication.hpp"
```

Include dependency graph for melmac_rviz/src/melmac.cpp:



Macros

- #define BUF_SIZE 1322
- #define LIDAR_FREQ 10
- #define ANGLE_INC 0.006136
- #define TIME_INC 0.000098

Functions

- void rvizWrapper (ros::NodeHandle ∗n, ros::Publisher ∗scan_pub, tf::TransformBroadcaster ∗broadcaster, ros::Rate ∗r)

  This function represents the sendThread.
- void readStreamingData (void)

  function for reading the measurement data from the socket connection. If and end message was read the function returns and the user can end or reopen the communication
- int main (int argc, char ∗∗argv)

  Main function of rviz_wrapper.

4.15.1 Detailed Description

contains the main application for wrapping data which are collected with the alf_urg application and sended to this client

Attention

can only be build within a working ROS environment

4.15.2 Macro Definition Documentation

4.15.2.1 #define ANGLE_INC 0.006136

Better working ANGLE_INC which works better than the commented calculation

Definition at line 29 of file melmac_rviz/src/melmac.cpp.

4.15.2.2 #define BUF_SIZE 1322

This defines the size of AlfMeasBuffer

Definition at line 25 of file melmac_rviz/src/melmac.cpp.

4.15.2.3 #define LIDAR_FREQ 10

The frequence of the Lidar. It is needed for the ros loop and scan_time

Definition at line 27 of file melmac_rviz/src/melmac.cpp.

4.15.2.4 #define TIME_INC 0.000098

Better working TIME_INC which works better than the commented calculation

Definition at line 30 of file melmac_rviz/src/melmac.cpp.

4.15.3 Function Documentation

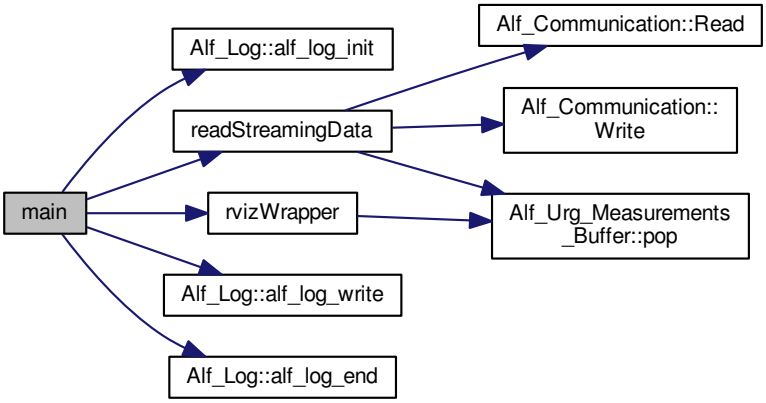
4.15.3.1 int main (int *argc,* char ** *argv*)

Main function of rviz_wrapper.

It opens the socket communication, starts the two threads (readThread and sendThread) etc.

Definition at line 125 of file melmac_rviz/src/melmac.cpp.

Here is the call graph for this function:



4.15.3.2 void readStreamingData (void)

function for reading the measurement data from the socket connection. If and end message was read the function returns and the user can end or reopen the communication

Parameters

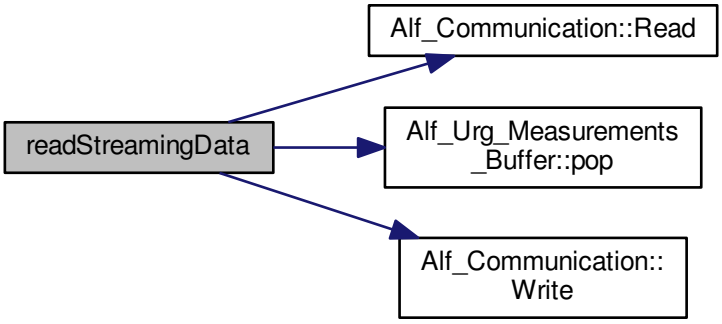
in	-	

Returns

-

Definition at line 93 of file melmac_rviz/src/melmac.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

4.15.3.3 void rvizWrapper (ros::NodeHandle * *n*, ros::Publisher * *scan_pub*, tf::TransformBroadcaster * *broadcaster*, ros::Rate * *r*)

This function represents the sendThread.

It takes all data from Alf Measurement Buffer and maps the data to the ros data structure

Parameters

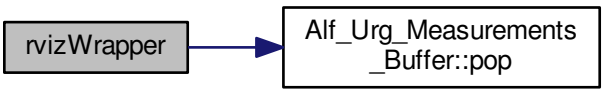
in	*n*	is the nodehandler which checks the status
in	*scan_pub*	is the Scan Publisher which sends all data to rviz
in	*broadcaster*	is the broadcaster to send tf messages to rviz
in	*r*	is necessary for creating a ros loop with the frequence of the lidar (here: 10 Hz)

Returns

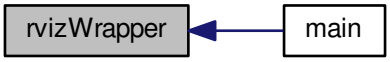
void

Definition at line 42 of file melmac_rviz/src/melmac.cpp.

Here is the call graph for this function:



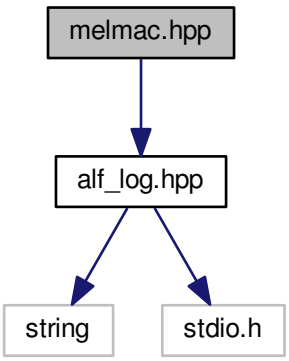
Here is the caller graph for this function:



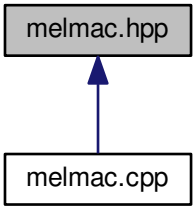
4.16 melmac.hpp File Reference

```
#include "alf_log.hpp"
```

Include dependency graph for melmac_Client/melmac.hpp:



This graph shows which files directly or indirectly include this file:



Functions

- int main ()

  the main process of this application this does
- void **readStreamingData** (void)

4.16.1 Function Documentation

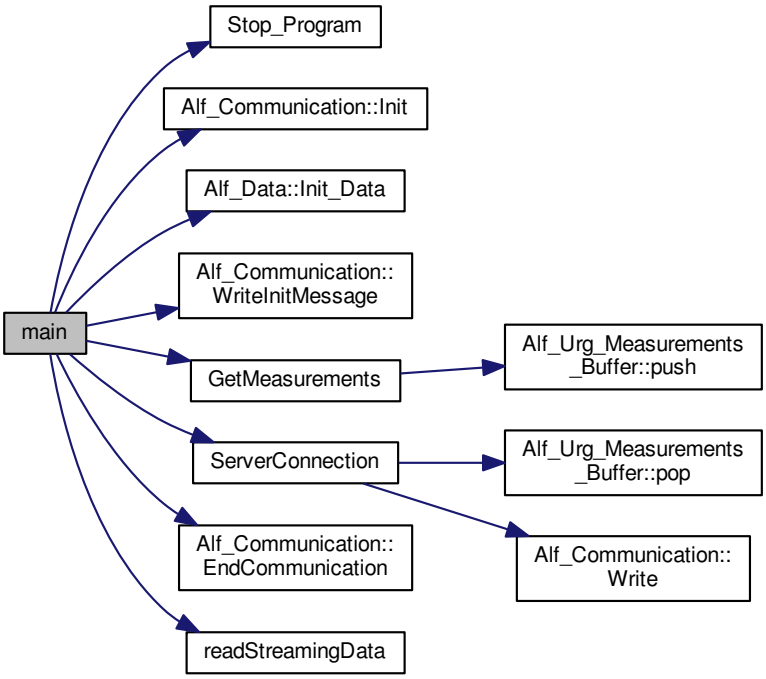
4.16.1.1 int main ( )

the main process of this application this does

- initializing the urg_sensor
- initializing the server connection
- starting the two threads
- ending the application in a clean way (after CTRL+C)

Definition at line 134 of file alf_urg.cpp.

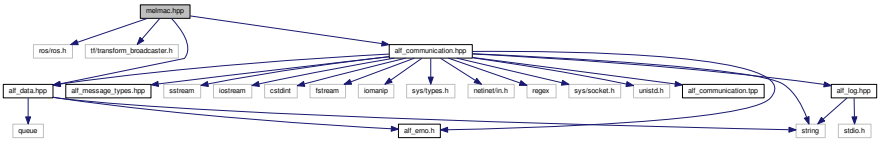
Here is the call graph for this function:



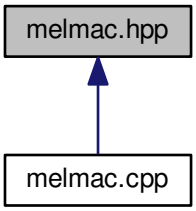
4.17 melmac.hpp File Reference

```
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>
#include "alf_data.hpp"
#include "alf_communication.hpp"
```

Include dependency graph for melmac_rviz/src/melmac.hpp:



This graph shows which files directly or indirectly include this file:



Functions

- void rvizWrapper (ros::NodeHandle ∗n, ros::Publisher ∗scan_pub, tf::TransformBroadcaster ∗broadcaster, ros::Rate ∗r)

  This function represents the sendThread.
- void readStreamingData (void)

  function for reading the measurement data from the socket connection. If and end message was read the function returns and the user can end or reopen the communication
- int main (int argc, char ∗∗argv)

  Main function of rviz_wrapper.

4.17.1 Detailed Description

All global variables, defines and the two functions which represents the threads are declared here

4.17.2 Function Documentation

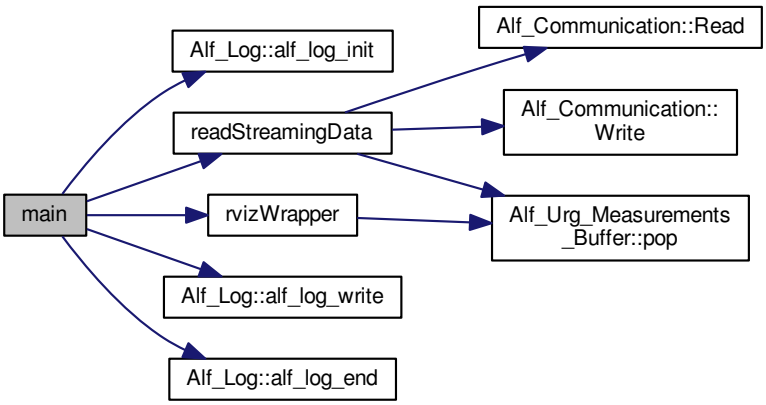
4.17.2.1 int main (int *argc,* char ∗∗ *argv*)

Main function of rviz_wrapper.

It opens the socket communication, starts the two threads (readThread and sendThread) etc.

Definition at line 125 of file melmac_rviz/src/melmac.cpp.

Here is the call graph for this function:



4.17.2.2 void readStreamingData (void)

function for reading the measurement data from the socket connection. If and end message was read the function returns and the user can end or reopen the communication

Parameters

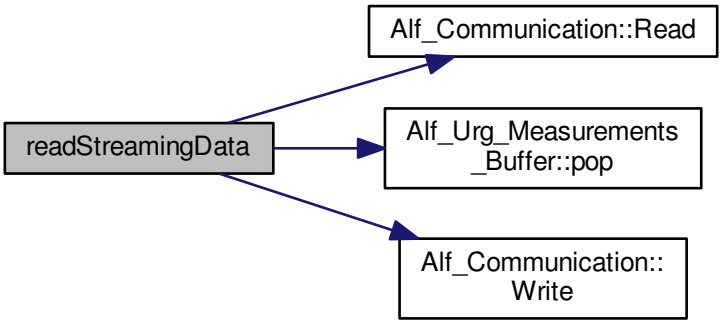
in	-	

Returns

-

Definition at line 27 of file melmac_Client/melmac.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:

4.17.2.3 void rvizWrapper (ros::NodeHandle * *n*, ros::Publisher * *scan_pub*, tf::TransformBroadcaster * *broadcaster*, ros::Rate * *r*)

This function represents the sendThread.

It takes all data from Alf Measurement Buffer and maps the data to the ros data structure

Parameters

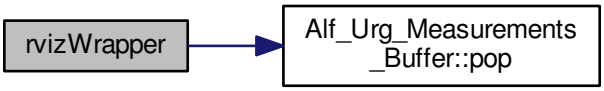
in	*n*	is the nodehandler which checks the status
in	*scan_pub*	is the Scan Publisher which sends all data to rviz
in	*broadcaster*	is the broadcaster to send tf messages to rviz
in	*r*	is necessary for creating a ros loop with the frequence of the lidar (here: 10 Hz)

Returns

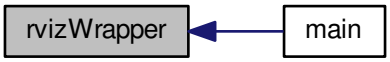
void

Definition at line 42 of file melmac_rviz/src/melmac.cpp.

Here is the call graph for this function:



Here is the caller graph for this function:



Index