**Projekt-Dokumentation**

Log-Buch

Inhalt

[**04.01.2025** 8](#_Toc194514040)

[Start & Basisarchitektur des Arduino MEGA2560 8](#_Toc194514041)

[ Ziel: Entwicklung und Implementierung der grundlegenden Architektur 8](#_Toc194514042)

[ Übersicht der Module und deren Funktionen 8](#_Toc194514043)

[ Auswahl der Software & Tools (Sloeber IDE vs. Arduino IDE) 8](#_Toc194514044)

[**17.01.2025** 8](#_Toc194514045)

[GitHub und Versionsverwaltung 8](#_Toc194514046)

[Anlegen der Repos für Embedded Software, HAS und Dockerfiles 8](#_Toc194514047)

[Einrichtung von GitHub zur Verwaltung von Tasks 8](#_Toc194514048)

[**18.01.2025** 8](#_Toc194514049)

[Docker-Setup für HAS 8](#_Toc194514050)

[Auswahl von Ubuntu als Basis 8](#_Toc194514051)

[Entwicklung des Dockerfiles und Docker Compose mit Felix Latzer 8](#_Toc194514052)

[Integration von MongoDB & 3rd-Party-Plugins 8](#_Toc194514053)

[Tryouts auf verschiedenen Hosts (Raspberry Pi 4B) 8](#_Toc194514054)

[**24.01.2025** 9](#_Toc194514055)

[Meetings mit VAT & Entscheidungsfindung zur Kommunikation 9](#_Toc194514056)

[ Diskussion zur Nutzung des VAT-Slave uC 9](#_Toc194514057)

[ Entscheidung gegen EtherCAT & für Ethernetmodul von VAT 9](#_Toc194514058)

[ Anpassung des Ethernetmoduls für GET/SET-Kommunikation 9](#_Toc194514059)

[**25.01.2025** 9](#_Toc194514060)

[Meetings & Zusammenarbeit mit Felix 9](#_Toc194514061)

[ Zielsetzung der Meetings 9](#_Toc194514062)

[ Fortschritte bei der Kommunikation zwischen HAS und eSW 9](#_Toc194514063)

[ End-to-End Testing mit Sensoren, HAS & MongoDB 9](#_Toc194514064)

[**07.02.2025** 9](#_Toc194514065)

[Meetings mit Dominik 9](#_Toc194514066)

[Einführung in die Architektur 9](#_Toc194514067)

[Debugging-Ansätze und Architektur-Erweiterung 9](#_Toc194514068)

[Spikes und Tests für HighCurrent/Flyback-Modul 9](#_Toc194514069)

[**14.02.2025** 10](#_Toc194514070)

[Debugging-Methoden 10](#_Toc194514071)

[ Serial Prints 10](#_Toc194514072)

[ StackOverflow, Blogposts, LLMs 10](#_Toc194514073)

[ Austausch mit Kollegen 10](#_Toc194514074)

[**23.02.2025** 10](#_Toc194514075)

[Ergebnisse aus Debugging & Verbesserungen 10](#_Toc194514076)

[Änderungen der Kommunikationsinterfaces 10](#_Toc194514077)

[Updates für verschiedene Module 10](#_Toc194514078)

[**07.03.2025** 10](#_Toc194514079)

[Programmierung & Libraries 10](#_Toc194514080)

[Programmiersprache: C/C++ 10](#_Toc194514081)

[Externe Libraries: Auswahl & Anpassungen 10](#_Toc194514082)

[**13.03.2025** 11](#_Toc194514083)

[Code-Dokumentation mit Doxygen 11](#_Toc194514084)

[ Einrichtung auf GitHub 11](#_Toc194514085)

[ Generierung von HTML- und LaTeX-Dokumenten 11](#_Toc194514086)

[**14.03.2025** 11](#_Toc194514087)

[ Allgemeine Dokumentation 11](#_Toc194514088)

[ Strukturierte Dokumentation der Codebase 11](#_Toc194514089)

[ Zusätzliche Anleitungen & Informationen 11](#_Toc194514090)

[**15.03.2025** 11](#_Toc194514091)

[Zusammenfassung & Review aller Komponenten 11](#_Toc194514092)

[ Letzte Anpassungen und Optimierungen 11](#_Toc194514093)

[**16.03.2025** 11](#_Toc194514094)

[Abschluss & Lessons Learned 11](#_Toc194514095)

[Herausforderungen & Lösungen 11](#_Toc194514096)

[Reflexion des gesamten Projekts 11](#_Toc194514097)

[**21.03.2025** 12](#_Toc194514098)

[**29.03.2025** 12](#_Toc194514099)

# **04.01.2025**

# Start & Basisarchitektur des Arduino MEGA2560

# Ziel: Entwicklung und Implementierung der grundlegenden Architektur

# Übersicht der Module und deren Funktionen

# Auswahl der Software & Tools (Sloeber IDE vs. Arduino IDE)

# **17.01.2025**

# GitHub und Versionsverwaltung

# Anlegen der Repos für Embedded Software, HAS und Dockerfiles

# Einrichtung von GitHub zur Verwaltung von Tasks

# **18.01.2025**

# Docker-Setup für HAS

# Auswahl von Ubuntu als Basis

# Entwicklung des Dockerfiles und Docker Compose mit Felix Latzer

# Integration von MongoDB & 3rd-Party-Plugins

# Tryouts auf verschiedenen Hosts (Raspberry Pi 4B)

# **24.01.2025**

# Meetings mit VAT & Entscheidungsfindung zur Kommunikation

# Diskussion zur Nutzung des VAT-Slave uC

# Entscheidung gegen EtherCAT & für Ethernetmodul von VAT

# Anpassung des Ethernetmoduls für GET/SET-Kommunikation

# **25.01.2025**

# Meetings & Zusammenarbeit mit Felix

# Zielsetzung der Meetings

# Fortschritte bei der Kommunikation zwischen HAS und eSW

# End-to-End Testing mit Sensoren, HAS & MongoDB

# **07.02.2025**

# Meetings mit Dominik

# Einführung in die Architektur

# Debugging-Ansätze und Architektur-Erweiterung

# Spikes und Tests für HighCurrent/Flyback-Modul

# **14.02.2025**

Meetings mit Bonelli

Kommunikation zwischen Manual-Board und eSW

Erweiterungsmöglichkeiten der Architektur

Schnittstellen & verwendete Hard

**22.02.2025**

# Debugging-Methoden

# Serial Prints

# StackOverflow, Blogposts, LLMs

# Austausch mit Kollegen

# **23.02.2025**

# Ergebnisse aus Debugging & Verbesserungen

# Änderungen der Kommunikationsinterfaces

# Updates für verschiedene Module

# **07.03.2025**

# Programmierung & Libraries

# Programmiersprache: C/C++

# Externe Libraries: Auswahl & Anpassungen

# **13.03.2025**

# Code-Dokumentation mit Doxygen

# Einrichtung auf GitHub

# Generierung von HTML- und LaTeX-Dokumenten

# **14.03.2025**

# Allgemeine Dokumentation

# Strukturierte Dokumentation der Codebase

# Zusätzliche Anleitungen & Informationen

# **15.03.2025**

# Zusammenfassung & Review aller Komponenten

# Letzte Anpassungen und Optimierungen

# **16.03.2025**

# Abschluss & Lessons Learned

# Herausforderungen & Lösungen

# Reflexion des gesamten Projekts

# **21.03.2025**

Agenda Auf Gitub:

**Flyback Module**  
-Flyback check (testingbranch, main branch)  
-Interrupt für Frequenzmessung (andere Lösung)

**Flyback working**

* Kommunikation für HAS (Endpoint)

**Versuchsaufbau**

* Flyback funktionalität
* Mit TEST SOFTWARE auslesen
* Vakuum testen (Ziel Vakuum erreicht oder in der Nähe)

**Thermosensor**

* Bestellen

**Steuerung**

* Besprechung der Funktion (Kraus Naimer Schalter)

**Klemmenmodul**

* gemeinsame Lösung ( wenn ja --> Illwerke)

**VAT Komponenten**

* HAS REQUESTS GET SET (ENDPOINT)
* Fully Implement GET; SET Compound (1,2,3)
* Testing mit HAS

**Schaltplan**  
-BONELLI erklären

**DISPLAY**

* PI Lösung finden

# **29.03.2025**

**Vakuumpumpe Programm**

* Building up vacuumpump logic //not done

**Vakuumpumpe working**

* Kommunikation für HAS (Endpoint) //not done

**Zusammenfügung Flyback und Vakuumpumpe**

* Sollte Logic funktionieren ( Architektur für einen Controlling Task) //not done

**Versuchsaufbau**

* Vakuum testen (Ziel Vakuum erreicht oder in der Nähe) // Nochmal für Doku Aufnahme DONE

**Thermosensor**

* Sensoren müssen morgen bestellt werden //MONTAG BONELLI
* Bausteine SPI für Auslesung sind bestellt //DONE

**Simulation Relaissteuerung Arduino**

* LTSPICE Simulation für Relaissteuerung //not done

**Arduino ExtenderBoard**

* Erstellen des Projektes (Idee)

**Klemmenmodul**

* gemeinsame Lösung ( wenn ja --> Illwerke) // open nice to have

**VAT Komponenten**

* HAS REQUESTS GET SET (ENDPOINT) //DONE
* Fully Implement GET; SET Compound (1,2,3) //not done
* Testing mit HAS // DONE 90%

**DUTYCYCLE Implementation Flyback**

* Zuzüglich unserer Frequenzeinstellung sollen wir auch den DutyCycle einstellen //DONE
* Test mit HAS //DONE

**Treiber austesten**

* Nicht möglich, Probleme mit STU klären

# **05.04.2025**

* Tryout mit Pfeiffer Vaccum Gerät um Drücke zu überprüfen.
  + Falsche einstellungen in VAT Software, diese durch Parameter in UI konfiguriert
    - Franic fragt bei Rolf nach, Probleme mit mapping/skalierung
* Aufbau und tryout von vacControl, sowohl als Code-Review mit Franic
* Bugfixes in ReportSystem und bugfixes von Compound getter/setter

# **17.04.2025**

* Verbesserung Logging und Printing, Konzept des LogManagers, Vorbereitung für Integration
* Hilfe für Franic mit VacControl, Endpoints, Debugging, Fehlersuche
* Versuchsaufbau mit beiden Controllern
  + Nach langen Einsätzen und unter starker Belastung steigt die Temperatur des EthernetShields Controller auf knapp 60°, daher verwenden wir Kühlkörper.
    - Nach Dauerbetrieb von 09:00 bis 18:00
  + (Passive Kühlung) um dem Problem entgegenzuwirken
  + Mit Fluke TiS20 Messung von beiden EthernetShield’s
* Optimierung der Speichernutztung unter Verwendung von Flashstrings um SRAM zu schonen, Optimierung des SerialMenu, analyze mittels:
  + Bessere loglevel, weniger spam in der seriellen Konsole

Ein Bild, das Text, Screenshot, Electric Blue (Farbe) enthält.

KI-generierte Inhalte können fehlerhaft sein.Arduino\_MEGA2560\_W5100\_EthernetShield Elegoo\_MEGA2560\_W5100\_EthernetShield

Ein Bild, das Text, Screenshot, Electric Blue (Farbe), Majorelle Blue enthält.

KI-generierte Inhalte können fehlerhaft sein.

# **19.04.2025**

* Ausarbeitung des Header Files und cpp files für den LogManager, erste Tests mit verschiedenen SD-Karten und Formaten
  + SD-Karte mit 32GB FAT32
  + SD-Karte mit 2GB FAT16
* Fehlersuche bei LogManager Libraries, probleme mit pinbelegung und SPI (gleichzeitiges benutzten von Ethernet und SD-Card nicht möglich)
  + Switchen zwischen den Teilnehmern im Bus, verwendung von pinMode und richtigen timings…
* Anbringen von passiver Kühlung auf IC für EthernetShield

Ein Bild, das Elektronik, Elektrisches Bauelement, Elektronisches Bauteil, passives Bauelement enthält.

KI-generierte Inhalte können fehlerhaft sein.

# **25.04.2025**

* Nach Franic’s Anruf bei VAT, talk mit Embedded Entwickler richitge Range für Drucksensor eingestellt.
  + API Endpoints Funktion des VacControls getestet
  + Task für VacControl getestet
  + Code optimiereung
* Weitere Test von MicroSD Karten für Logging

# **26.04.2025**

* Testaufbau mit Franic, testen neue Range von Sensor.
  + PI- Regler Einstellungen vorgenommen um Pressure Control mode zu optimieren.
    - Anstatt Downstream Mode wurde nun der Upstream Mode verwendet und 2 x Learning Mode ausgeführt um Daten im CSV zu sammeln.
    - Eventuelle Absprache oder Austausch mit VAT bezüglich generierter Daten und Verhalen.
* Finalisierung des VacControl und kleiner Bugfixes, Codereview
* Anpassung des FlyBackVacControlTask
  + erhöhung der max. Erlaubten Stacksize um overflow zu verhindern
* StackMonitorTask Anpassungen für max. Grenzwerte des Stacks
* Anpassungen API-Endpoints, sowie aufräumen von alten Debug Messages usw.
* Bugfix und einige Tryouts im logManager, Probleme mit SPI Zugriff bzw setzten der States der betroffenen Ports

# **10.05.2025**

* Alle librarys aufgeräumt, besonders fokus auf Temp library
* MCP9601 angepasst, change device id to 0x41 in baseclass of adafruit
* Troubleshooting versuchsaufbau, integration neue Hardware, SRAM usage verkleinert
* Einge Graphen Anpassen und UML’s sowie Dokumentation
  + runAwk.bat neu ausführen
  + Test Endpoint Shellscripts anpassen
  + Dokumentation anpassen
  + <https://forums.adafruit.com/viewtopic.php?t=183174>

# **14.05.2025**

* Updaten der CI Github Action von FFRESW Repository
  + Compilieren
  + Statische Analyse des Codes
  + Binary Analyse der Elf Binaries
  + Buildprozess und Deployment als Buildartefakt
  + Troubleshooting

# **17.05.2025**

* Versuchsaufbau mit Arduino-Extendershield und Schaltbare Netzteile.

Erkentniss:

* BC635 – IC 4Ampere
* B337 – IC 800mA

Kurze größere Einschaltströme vorhanden, aber nur sehr kurz daher kaum bis keine Wärmeentwicklung.

Auf Polung achten bei Netzteilsteuerung, sollte nicht Funktionieren nochmals Polung überprüfen und ausprobieren -> In Zukunft Kabel markieren.

* Überarbeitung der Librarys für Temperture, SensorModule und I2C Communication
  + Einpflegung der neuen MCP9601 Board und anpassung sowie kontrolle der Sensoren
  + Pläne erstellt und Bei Felix in Auftrag gegeben für Gehäuse für Externes PCB für beide MCP9601
* Pipeline Runs überprüft und durch statische Analyse Warnings im Code gefunden und diese Behoben.

# **Architekturübersicht – FFRESW (Farnsworth Fusion Reactor Embedded Software)**

**Einführung**

FFRESW ist die Embedded Software des Farnsworth Fusionsreaktors. Sie basiert auf C++ und läuft auf einem Arduino MEGA2560. Aufgrund der Plattform wurde C++ gewählt, da diese mit .ino-Dateien und der Arduino-IDE kompatibel ist. Der objektorientierte Aufbau erleichtert Wartung, Erweiterbarkeit und Modularisierung.

Die Software ist in **Core Libraries** (eigene Entwicklungen) und **externe Libraries** unterteilt.

**1. Core Libraries (eigene Entwicklungen)**

**1.1 comModule**

Das comModule bietet Zugriff auf die verschiedenen Schnittstellen des Systems:

* **ETH** (Ethernet): Kommunikation mit dem HAS (Hardware Access Service) und dem VAT µC.
* **I2C**: Besonders wichtig zur Auswertung der Temperaturdaten.
* **SER** (Serielle Schnittstelle)
* **SPI**: Zentral für das Ethernet-Shield und den LogManager. Darüber wird gesteuert, wer wann senden darf.

Diese Library wird genutzt, um Daten zu senden, zu empfangen sowie Konfigurationen zu übertragen.

**1.2 calcModule**

Das calcModule enthält verschiedene Funktionen zur Datenverarbeitung:

* Umrechnung von Sensorwerten und Kammerdaten
* Finalisierung, Parsing und Bearbeitung von Rohdaten
* Berechnung abgeleiteter Werte

**1.3 sensModule**

Das sensModule stellt die Verbindung zu den Sensor-Libraries **Temperature** und **Pressure** her. Es übernimmt:

* Überwachung und Prüfung von Sensordaten
* Abstraktion zur einfachen Einbindung neuer Sensoren
* Einheitliches Interface zur Sensordatenerfassung

**1.4 jsonModule**

Ein integraler Bestandteil der Core Libraries, da die gesamte Kommunikation über JSON erfolgt:

* Gestaltung und Verwaltung von JSON-Objekten und -Strings
* Validierung der Datenstruktur
* Sicherheitsmechanismen zur Gewährleistung valider Daten

**1.5 logManager**

Ermöglicht das kontinuierliche Logging auf der SD-Karte im Ethernet-Shield:

* Speicherung strukturierter Logdaten
* Optimierungen zur Umgehung bekannter SPI-Probleme
* Interaktion mit dem serialMenu zur Protokollierung serieller Ausgaben

*Hinweis: [Hier können zukünftig Links oder Referenzen eingefügt werden]*

**1.6 serialMenu**

Bietet sichere und strukturierte Kommunikation über die serielle Schnittstelle:

* Thread-Sicherheit durch Mutexes (FreeRTOS)
* Steuerung der Ausgabe über boolesche Flags
* Möglichkeit zur direkten Speicherung von Nachrichten im LogManager

**1.7 reportSystem**

Umfasst zentrale Funktionen zur Systemüberwachung:

* Stack Guard und Stackoverflow-Detection
* Speicherüberwachung (RAM, EEPROM)
* Kontrolle aller Kommunikationswege
* Threshold-Kontrolle, Stackdump-Funktionalität
* Fehlerpersistenz im EEPROM

**1.8 timeModule**

Verwaltet die Systemzeit auf dem Arduino MEGA2560:

* Abruf der aktuellen UTC-Zeit via Ethernet (als JSON)
* Lokale Speicherung in einer DateTimeStruct
* Eigenständige Inkrementierung und Formatierung
* Nutzung in allen gesendeten JSON-Daten

**1.9 ptrUtils**

Eine Header-only-Library zur Verwaltung von Pointern:

* Überprüfung und Validierung
* Speicher- und Zugriffskontrolle

**1.10 vacControl**

*Beschreibung folgt.*

**1.11 flyBack**

*Beschreibung folgt.*

**2. Externe Libraries**

Folgende Libraries wurden importiert, um Funktionalität, Stabilität und Kompatibilität zu erweitern:

* **ArduinoJson** – Verarbeitung von JSON-Daten
* **Arduino STL** – Nutzung von Standard-Template-Elementen (z. B. vector, string)
* **SD** – Zugriff auf die SD-Karte
* **SPI** – Kommunikation mit SPI-Geräten
* **Ethernet** – Netzwerkkommunikation
* **ErriezMemoryUsage** – Überwachung des Speichers
* **FreeRTOS** – Betriebssystem für Multitasking, Zeitsynchronisierung etc.
* **Frt.h** – Objektorientierter Wrapper für FreeRTOS
* **MAX31855** – Library, um Temperatursensor auszuwerten
* **MAX6675** – Library, um Temperatursensor auszuwerten

**3. Entscheidungsgrundlage & Architekturprinzipien**

Die Entscheidung für C++ basiert auf den technischen Gegebenheiten des Arduino MEGA2560 und der nativen Unterstützung von .ino-Dateien. Das objektorientierte Design erleichtert:

* Die modulare Erweiterbarkeit
* Die klare Trennung der Funktionalitäten
* Die langfristige Wartung und Dokumentation

**4. Erweiterbarkeit**

Diese Struktur ist so konzipiert, dass sie leicht ergänzt und gepflegt werden kann:

* Neue Module können analog zu den bestehenden Sections hinzugefügt werden
* Platzhalter für vacControl und flyBack ermöglichen spätere Ergänzungen
* Alle Module können mit Quellcode-Links, Diagrammen oder Doxygen-Kommentaren ergänzt werden

Beschreibung der Anwendung der Libraries:

Ich habe mich für C++ entschieden, da wir aufgrund des Arduino MEGA2560 bereits mit C++/.ino arbeiten. Objektorientierter Ansatz beim Aufbau.

Das **comModule** bietet Zugriff auf die verschiedenen Schnittstellen auf unserem System, vor allem das Ethernet-Modul, mit dem wir als eSW mit dem HAS (Hardware Access Service) und dem VAT µC kommunizieren. Diese Library wird verwendet, um Daten zu senden und zu empfangen sowie Konfigurationen zu versenden oder zu erhalten. Durch das **comModule** arbeiten wir auch mit dem I2C-Modul, was besonders wichtig für die Auswertung der Daten aus dem Temperatursensor ist. **SPI** ist sowohl für den **logManager** als auch das Ethernet-Shield wichtig, da wir dort die Teilnehmer tracken und steuern, wer wann senden darf.

Das **calcModule** bietet verschiedenste Funktionen, um wichtige Werte der Vakuumkammer oder Daten von Sensoren umzurechnen, zu finalisieren, zu parsen oder zu bearbeiten und umzuwandeln.

Das **sensModule** bietet Zugriff auf die Libraries **Temperature** und **Pressure**, welche im **sensModule** überwacht und geprüft werden. Des Weiteren dient die Abstraktion dazu, ein einfaches Interface zu schaffen und schnell neue Sensoren einzubinden, sollte dies notwendig sein.

Das **jsonModule** ist ein integraler Bestandteil der Core-Libraries, da wir das JSON-Format verwenden, um Daten zu versenden oder zu empfangen. Diese Library übernimmt die wichtigsten Grundaufgaben in der Gestaltung der JSON-Strings/Objekte und hat selbst einige Sicherheitsmechanismen, um valide JSONs zu garantieren.

Der **logManager** bietet die Möglichkeit, Logfiles zu erstellen und diese kontinuierlich auf der verwendeten SD-Karte im Ethernet-Shield zu speichern. Diese Library beinhaltet einige Tweaks, um Probleme mit dem SPI-Bus zu umgehen und einen guten Workflow zu garantieren.

[HIER LINKS EINFÜGEN]  
Des Weiteren sind einige Funktionen des **logManager** Bestandteil des **serialMenus**, was es uns erlaubt, Serial-Messages direkt in einem strukturierten Format abzuspeichern.

Das **serialMenu** bietet einige Funktionen, um sicheres Drucken via Serial zu garantieren. Hier haben wir Wert auf Sicherheit gelegt; die Funktionen verwenden Mutexes, um Probleme beim Drucken mit mehreren Tasks innerhalb des RTOS zu verhindern. Des Weiteren erfolgt die Speicherung der Messages durch das Setzen eines Boolean-Flags, um den **logManager** zu aktivieren.

Das **reportSystem** hat verschiedene Funktionen, um die Integrität des Systems zu gewährleisten, vom **StackGuard** und **StackOverflow**-Detection bis hin zur Speicherüberwachung von allen relevanten Speichern. Des Weiteren werden alle Kommunikationswege überwacht und kontrolliert sowie alle anderen Sensoren. Es erfolgt eine Kontrolle von Thresholds, das Abrufen von Stack-Dumps und das Speichern von Fehlern im EEPROM.

Das **timeModule** verwaltet die Zeit auf dem Arduino MEGA2560. Im Setup holen wir uns mit dem Ethernet-Modul den JSON-String mit der aktuellen Zeit in UTC. Danach speichern wir diesen lokal auf dem Arduino in einem DateTime-Struct, inkrementieren die Zeit selbstständig, formatieren diese und verwenden sie dann in unseren erstellten JSON-Strings, die wir auch versenden.

Die **ptrUtils** (Pointer Utilities) ist eine Header-only-Library, die wir verwenden, um Pointer zu überprüfen, zu überwachen und zu managen.

Flyback -> // NOCH SCHREIBEN -> FRANIC!!!!!!!!!!!!!!!!!!!!!!

Die **Flyback Library** steuert das Hochspannungsmodul (HV-Modul), indem sie Frequenz und Duty Cycle anpasst, um die Ausgangsleistung zu regulieren. Sie stellt auch Funktionen zum Auslesen von Systemwerten wie Ausgangsspannung und Stromfluss zur Verfügung.

vacControl -> // NOCH SCHREIBEN -> FRANIC!!!!!!!!!!!!!!!!!!!!!!

Die **vacControl Library** steuert und überwacht das Vakuumsystem, inklusive Ventilsteuerung und Druckabfragen. Sie wird in zwei Tasks genutzt:

**Hardwareübersicht FFRESW**

**Arduino MEGA 2560 R3**

Für das Projekt wurde der **Arduino MEGA 2560 R3** gewählt. Dieser Mikrocontroller zeichnet sich durch eine hohe Anzahl an digitalen und analogen Ports aus, die eine flexible Anbindung verschiedener Peripheriegeräte ermöglichen. Mit 54 digitalen I/O-Pins, 16 analogen Eingängen und 4 UARTs bietet der MEGA 2560 eine exzellente Basis für die Erweiterung des Systems und den Anschluss zahlreicher Sensoren und Aktoren. Die Leistungsfähigkeit dieses Mikrocontrollers ist für den Preis hervorragend und stellt sicher, dass auch komplexe Aufgaben in Echtzeit verarbeitet werden können.

Ein weiterer großer Vorteil des **Arduino MEGA 2560 R3** ist die umfangreiche Unterstützung durch die Arduino-Community. Es gibt unzählige Tutorials, Foren und Libraries, die eine schnelle Implementierung und Fehlerbehebung ermöglichen. Besonders wichtig war uns, dass jeder im Team die Embedded Software (eSW) verstehen und bei Bedarf auch selbstständig erweitern oder anpassen kann. Für Programmierer mit weniger Erfahrung bietet die breite Zugänglichkeit des Systems eine ideale Grundlage. Im Vergleich zu anderen Mikrocontrollern, wie beispielsweise den ST-Mikrocontrollern, die ebenfalls in Betracht gezogen wurden, hat der Arduino MEGA 2560 den Vorteil einer niedrigeren Einstiegshürde und einer weitaus größeren Entwicklergemeinschaft.

**Ethernetshield W5100 von AZ Delivery**

Für die Netzwerkkommunikation wurde das **Ethernetshield W5100** von **AZ Delivery** verwendet. Das Shield nutzt den **W5100 Ethernet-Chip**, der eine zuverlässige und leistungsstarke Möglichkeit bietet, das Arduino-Board mit einem Netzwerk zu verbinden. Besonders hervorzuheben ist die einfache Integration und Handhabung des Ethernet-Shields, das mit einer Standardbibliothek von Arduino direkt kompatibel ist. Dies erleichterte die Implementierung der Netzwerkkommunikation erheblich.

Das **Ethernetshield W5100** verfügt über einen eingebauten **MicroSD-Kartenleser**, der für die Speicherung von Logdateien und anderen Daten verwendet werden kann. Diese Funktion war für das Projekt besonders relevant, da wir eine Lösung benötigten, die sowohl die Kommunikation über Ethernet als auch die Speicherung von Daten auf einer SD-Karte ermöglicht. Das Shield unterstützt TCP/IP-Verbindungen und ermöglicht die schnelle Übertragung von Daten zwischen dem Arduino-Board und einem externen Server oder anderen Geräten im Netzwerk.

Ein weiterer Vorteil des **W5100 Chips** ist, dass er im Vergleich zu anderen Ethernet-Modulen eine höhere Stabilität und geringere Latenz bietet, was für unsere Anforderungen an die Netzwerkkommunikation von entscheidender Bedeutung war.

**Zusammenfassung der Entscheidung:**

Die Wahl des **Arduino MEGA 2560 R3** und des **Ethernetshield W5100** von **AZ Delivery** wurde maßgeblich durch die folgenden Faktoren beeinflusst:

* **Arduino MEGA 2560 R3**:
  + Große Anzahl an digitalen und analogen I/O-Pins
  + Kostengünstig und leistungsstark für das Preis-Leistungs-Verhältnis
  + Umfangreiche Arduino-Community und Libraries
  + Einfache Implementierung für Entwickler mit unterschiedlichem Erfahrungsstand
* **Ethernetshield W5100 von AZ Delivery**:
  + Integrierter W5100 Ethernet-Chip für stabile Netzwerkverbindungen
  + MicroSD-Kartenleser für Datenspeicherung
  + Einfache Integration mit bestehenden Arduino-Libraries

Durch diese Wahl konnten wir die Hardwareanforderungen effizient abdecken und gleichzeitig eine Entwicklungsumgebung wählen, die für das gesamte Team zugänglich und gut dokumentiert ist.

**Limitationen durch Hardware und bekannte SW Bottlenecks**

Durch die Verwendung des **Arduino MEGA2560** ergeben sich bestimmte Einschränkungen, insbesondere in der Kommunikation mit dem **VAT uC** zur Steuerung der Ventile. Die API bietet Endpunkte mit Getter- und Setter-Funktionen, um Werte zu senden oder abzurufen. Aufgrund von Hardware- und Softwarearchitektur-Limitationen war es jedoch nicht möglich, kontinuierliche Datenströme von den Sensoren des **VAT uC** bereitzustellen. Stattdessen arbeiten wir im **FFRESW**-Projekt über das **EthernetModule** mit den **Compound1**, **Compound2** und **Compound3**-Befehlen, was unsere Möglichkeiten in dieser Hinsicht etwas einschränkt.

Durch die Verbindung des **VAT uC** via Ethernet über Port 503 kann jedoch der **Hardware Access Service (HAS)** konstant Daten streamen. Diese Daten können dann in einer Datenbank gespeichert und über **Grafana** visualisiert werden. Dieser Datenstrom erfolgt allerdings nicht über den **Arduino MEGA2560**, sondern direkt über den **VAT uC**.

Zusätzlich ergeben sich durch die Verwendung von **FreeRTOS** und dem OOP-Wraper **frt.h** Speicher- und **SRAM**-Limitationen. Um möglichen Problemen aufgrund dieser Begrenzungen entgegenzuwirken, haben wir einen **StackMonitorTask** implementiert. Dieser überwacht die anderen Tasks und verhindert so potenzielle **Stackoverflow**-Fehler.

Des Weiteren habe ich mir die Tools der GNU Compiler Collection zu nutzen gemacht und diese verwendet, um etwaige große Speicherallokationen in dem ELF Binary festzustellen. Danach habe ich diese dann behoben/entfernt/optimiert.

**FlowDiagramme/DoxyGen**

Für die Analyse der Programmstruktur sowie die Erstellung von Flowcharts und Architekturdiagrammen wurde das Open-Source-Tool **Sourcetrail** verwendet. Sourcetrail ermöglicht es, komplexe Softwareprojekte in C/C++, aber auch in anderen Sprachen wie Java oder Python, effizient zu indexieren, zu scannen, zu parsen und grafisch darzustellen.

Durch die graphische Visualisierung von Funktionsaufrufen, Klassenhierarchien und Datenabhängigkeiten war es uns möglich, einen tiefen Einblick in den Aufbau und die Abläufe unseres Codes zu gewinnen. Insbesondere bei der Arbeit an einem umfangreichen Embedded-Projekt mit Echtzeit-Anforderungen erwies sich dies als großer Vorteil, da wir damit nicht nur das Zusammenspiel der einzelnen Module nachvollziehen konnten, sondern auch versteckte Designfehler, zyklische Abhängigkeiten oder unübersichtliche Strukturen identifizieren konnten.

Ein großer Mehrwert von Sourcetrail war die interaktive Darstellung des Control Flows (Programmfluss), welche uns ermöglichte, gezielt Problemstellen im Code zu analysieren und zu verbessern.

Darüber hinaus haben wir uns entschieden, die mit Sourcetrail generierten Diagramme direkt in unsere Dokumentation zu übernehmen, da sie sowohl **qualitativ hochwertig** sind als auch **einfach aktualisiert und angepasst** werden können, wenn sich der Code weiterentwickelt.

Leider wurde das ursprüngliche Projekt von **Coati Software** eingestellt, ist aber mittlerweile durch einen Fork aktiv weitergeführt worden:

* Original Repository: <https://github.com/CoatiSoftware/Sourcetrail>
* Aktuell gepflegter Fork: <https://github.com/petermost/Sourcetrail>

Dieser Fork stellt sicher, dass Sourcetrail weiterhin nutzbar und aktuell bleibt, was insbesondere für Open-Source-affine Entwicklergemeinschaften und Forschungsprojekte und für unser Projekt von großer Bedeutung ist.

Ein Bild, das Text, Screenshot, Software, Computersymbol enthält.

KI-generierte Inhalte können fehlerhaft sein.

In diesem Abschnitt betrachten wir die Hauptdatei unseres Projekts, **FFRESW.ino**. Bei Arduino-Projekten ist die Dateiendung .ino standardisiert für sogenannte *Sketches*. Warum diese Endung verwendet wird und wie die Arduino-IDE sie verarbeitet, wurde bereits im entsprechenden Abschnitt erläutert: [ABSATZ UND ABSCHNITT ZUR ERLÄUTERUNG VERLINKEN].

Die Datei FFRESW.ino stellt den Einstiegspunkt unserer Embedded-Software dar. Hier werden alle benötigten Module eingebunden und die initiale Systemstruktur aufgebaut. Das Projekt basiert auf **FreeRTOS**, genauer gesagt auf der Arduino-Portierung *Arduino\_FreeRTOS*, und verwendet zusätzlich die Bibliothek **frt.h**, welche eine objektorientierte Schnittstelle (OOP-Wrapper) für FreeRTOS bereitstellt. Dadurch wird eine saubere und strukturierte Umsetzung mehrerer parallel laufender Tasks ermöglicht.

In unserem System sind aktuell **vier Tasks** definiert:

1. **ReportTask**  
   Diese Task übernimmt die Systemüberwachung. Sie kontrolliert kontinuierlich den Zustand des Gesamtsystems, inklusive aller Kommunikationsprotokolle (z. B. I2C, UART), Sensoren, Aktoren und weiterer Peripherie. Sie dient als zentrale Stelle für Statusberichte und Systemdiagnosen und verwendet dafür das reportSystem.

Ein Bild, das Text, Screenshot, Diagramm, Software enthält.

KI-generierte Inhalte können fehlerhaft sein.

1. **StackMonitorTask**  
   Diese Task beobachtet die Stacknutzung aller anderen Tasks. Ziel ist es, Stacküberläufe (Stackoverflow) frühzeitig zu erkennen. Wenn sich ein Stack der definierten Grenze nähert, erfolgt eine Warnung. Eine automatische Reaktion (z. B. Neustart der betroffenen Task) ist vorgesehen, wurde aber zum aktuellen Stand noch **nicht implementiert**.

Ein Bild, das Text, Screenshot, Betriebssystem, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein.

1. **SensorActorEndpointTask**  
   Dieser Task verwaltet alle logischen Endpunkte des Systems, insbesondere die Kommunikation und Steuerung der angeschlossenen Sensoren und Aktoren. Er fungiert als Schnittstelle zwischen dem Benutzer bzw. dem übergeordneten System und der eigentlichen Embedded-Hardware. Dieser Task bietet also Endpoints zu allen Zentralen Stellen innerhalb des Systems, wie dem ReportSystem, dem Flyback, dem VacControl und dem SensorModule.

Ein Bild, das Text, Screenshot, Zahl, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

1. Ein Bild, das Text, Screenshot, Diagramm, Schrift enthält.

   KI-generierte Inhalte können fehlerhaft sein.**FlybackVacControlTask**  
   Zuständig für die Regelung des Flyback-Wandlers und des Vakuumsystems. Dieser Task übernimmt die Ansteuerung sowie das Timing für die Hochspannungserzeugung und Vakuumregelung, welche zentrale Bestandteile der Hardwarefunktionalität darstellen. Verwendet werden hierfür ebenfalls die Libraries für den Flyback und das VacControl System.

Durch die Verwendung von FreeRTOS mit OOP-Wrapper ergibt sich eine klare Trennung der Zuständigkeiten zwischen den Tasks sowie ein übersichtliches und wartbares Softwaresystem, das modular erweitert werden kann.

Ein Bild, das Text, Screenshot, Design enthält.

KI-generierte Inhalte können fehlerhaft sein.

Diese Flowchart zeigt das **ReportSystem**, welches eine zentrale Rolle in unserer Embedded Software einnimmt. Es dient als übergreifende Instanz zur Systemüberwachung und Fehlerberichterstattung. Wie im Diagramm ersichtlich, ist das ReportSystem mit einer Vielzahl an Modulen verbunden, darunter **Kommunikationsmodule** (z. B. Ethernet, SPI, I2C), **Sensormodule** und **Zeitmodule**. Diese Einbindung erfolgt über entsprechende Header-Dateien wie ETH.h, SPI1.h, I2C.h, sensorModule.h und timeModule.h.

Das ReportSystem nutzt verschiedene **Methoden** zur Überwachung der Funktionalität dieser Module. Fehlerzustände oder kritische Systemereignisse – wie etwa Kommunikationsfehler, Sensorausfälle oder Zeitabweichungen – werden durch dedizierte Funktionen erkannt (z. B. checkSensors, checkCommunication, checkSystemHealth) und über Methoden wie reportError oder reportStatus dokumentiert und weitergemeldet.

Darüber hinaus übernimmt das System eine zentrale Rolle beim **Monitoring von Ressourcen**: Funktionen wie getMemoryStatus, checkRamLevel, getStackDump oder detectStackOverflow ermöglichen eine präzise Überwachung des internen Speichers, des SRAMs und der Stack-Nutzung. Schwellenwerte können über setThreshold gesetzt und über checkThresholds überprüft werden. So wird sichergestellt, dass bei Überschreitung definierter Grenzen rechtzeitig gewarnt wird.

Im Fehlerfall wird nicht nur der Fehler selbst, sondern auch der genaue Zeitpunkt (startBusyTime, startIdleTime) sowie weitere relevante Systeminformationen (getLastErrorInfo, saveLastError) protokolliert. Dadurch bietet das ReportSystem eine robuste Grundlage für Fehlerdiagnose und Systemstabilität – von einem Stackoverflow bis hin zu fehlerhaften Sensordaten oder Kommunikationsproblemen.

BILD EINFÜGEN!!!!!!!!!!

Die **Flyback Library** dient der Ansteuerung des Hochspannungsmoduls (HV-Modul) innerhalb unserer Embedded Software. Sie ermöglicht eine gezielte Kontrolle zentraler Betriebsparameter wie der **Frequenz** und des **Duty Cycles**, um die Ausgangsleistung des Moduls präzise an verschiedene Anforderungen anzupassen.

Neben der reinen Steuerung stellt die Bibliothek auch eine Reihe von **Getter-Funktionen** zur Verfügung, über die sich aktuelle Systemwerte auslesen lassen. Dazu zählen unter anderem Informationen über die **Ausgangsspannung**, den **Stromfluss** und weitere relevante Messgrößen. Diese Funktionen bieten die Grundlage für eine kontinuierliche Überwachung und ermöglichen es, das System in Echtzeit zu bewerten und bei Bedarf zu regeln.

Die Integration der Flyback Library erfolgt an zwei verschiedenen Stellen innerhalb des Task-Modells der Embedded Software:

1. **SensorActorEndpointTask**: Dieser Task ermöglicht eine externe Steuerung der Flyback-Funktionalität über definierte Schnittstellen, insbesondere über die API. Er wird dann aktiv, wenn ein übergeordnetes System oder Benutzerinterface Steuerbefehle an das HV-Modul übermittelt.
2. **FlybackVacControlTask**: Dieser Task arbeitet autonom und übernimmt die Kontrolle über das HV-Modul eigenständig, wenn keine externe Steuerung aktiv ist. Über einen definierten Mechanismus – in der Regel über den **SwitchState** – erkennt der Task, ob er weiterhin zuständig ist oder ob die Steuerung durch einen anderen Task (z. B. den SensorActorEndpointTask) übernommen wurde.

Durch dieses Konzept wird sichergestellt, dass die Flyback-Steuerung sowohl **autonom** als auch **fernsteuerbar** betrieben werden kann – flexibel anpassbar an unterschiedliche Anwendungsszenarien und Betriebsmodi.

BILD EINFÜGEN!!!!!!!!!!

Die **vacControl Library** ist ein zentrales Modul zur Steuerung und Überwachung des **Vakuumsystems** innerhalb unserer Embedded Software. Sie ermöglicht es dem Benutzer, gezielte Steuerbefehle an das System zu senden und gleichzeitig aktuelle Betriebsdaten des Vakuumsystems auszulesen. Dazu zählen unter anderem Zustände von Ventilen, Druckwerte und Systemantworten, die für die Prozessüberwachung und -regelung von entscheidender Bedeutung sind.

Wie bereits bei der Flyback Library erfolgt auch bei vacControl die Einbindung in **zwei unterschiedliche Tasks**:

1. **SensorActorEndpointTask**: Dieser Task ermöglicht eine **direkte Steuerung über die API**. Wird ein neues Szenario oder ein Steuerbefehl von außen empfangen (z. B. über eine PC-Software oder ein HMI), reagiert vacControl entsprechend und passt die Vakuumsteuerung an die neuen Anforderungen an.
2. **FlybackVacControlTask**: In Abwesenheit externer Befehle arbeitet das vacControl-System **autonom**. Der FlybackVacControlTask übernimmt in diesem Fall die selbstständige Regelung des Vakuumsystems, unter anderem durch Ansteuerung des **VAT Mikrocontrollers**, welcher für die Ventilsteuerung zuständig ist.

Ein zentraler Mechanismus zur Koordination dieser beiden Betriebsmodi ist wiederum der **SwitchState**. Er sorgt dafür, dass klar definiert ist, welcher Task zu welchem Zeitpunkt die Kontrolle über das Vakuumsystem besitzt. So wird vermieden, dass konkurrierende Befehle gleichzeitig ausgeführt werden oder unklare Systemzustände entstehen.

Durch diese duale Architektur ist das vacControl-Modul in der Lage, sowohl **automatisiert** als auch **benutzergeführt** zu agieren – flexibel, robust und sicher im Betrieb. Es stellt somit eine wichtige Schnittstelle zwischen übergeordneter Steuerlogik und physikalischer Aktorik im Vakuumbereich dar.

**Erklärung Zusammenspiel HAS, ESW und VAT**

Die Systemarchitektur basiert auf mehreren vernetzten Komponenten, die über **Ethernet** miteinander kommunizieren:

* **Hardware Access Service (HAS)**: ein Raspberry Pi
* **Embedded Software (eSW)**: ein Arduino Mega 2560
* **VAT-Steuergerät**: ein Mikrocontroller der Firma VAT

Die Kommunikation erfolgt über **TCP/IP**-basierte Datenverbindungen. Dabei werden die Nachrichten in **JSON**-Format übertragen. Diese JSON-Objekte enthalten strukturierte Informationen über Steuerbefehle, Sensorwerte oder Systemstatus. Der HAS speichert diese Daten persistent in einer **MongoDB-Datenbank**.

Da der **VAT-Mikrocontroller** ein proprietäres oder anders strukturiertes Datenformat erwartet, übernimmt die eSW auf dem Arduino die Rolle eines Protokollübersetzers:  
Sie **konvertiert eingehende JSON-Daten in das vom VAT erwartete Format** und wandelt auch die Antworten vom VAT wieder zurück in JSON, sodass sie vom HAS korrekt interpretiert und gespeichert werden können.

**Schnittstellen & Protokolle**

Neben der Ethernet-Kommunikation über TCP/IP kommen weitere Schnittstellen zum Einsatz:

* **SPI (Serial Peripheral Interface)**:
  + Für das **Ethernet-Shield** am Arduino (W5100)
  + Für den Zugriff auf eine **MicroSD-Karte**
* **I²C oder SPI**:
  + Für den Anschluss verschiedener **Temperatursensoren** (z. B. MAX6675, MLX90614, DHT11)
* **EtherCAT**:
  + Wird auf dem VAT-Mikrocontroller verwendet, um mit **Drucksensoren** und **Ventilen** zu kommunizieren. EtherCAT ist ein industrielles Echtzeit-Ethernet-Protokoll mit hoher Genauigkeit und Synchronität, welches für solche Anwendungen ideal geeignet ist.

**Softwareinfrastruktur**

**Raspberry Pi (HAS)**

Auf dem Raspberry Pi läuft ein **Docker-Container**, basierend auf einem Ubuntu-Image, in dem sämtliche benötigten Software-Komponenten, Konfigurationsdateien und Services gekapselt sind. Die komplette Systemkonfiguration ist in einem **GitHub-Repository** versioniert. Der Hardware Access Service (HAS) ist dabei als **Git-Submodul** in dieses Repository eingebunden, um eine modulare und wartbare Struktur zu ermöglichen.

**Arduino Mega 2560 (eSW)**

Auf dem Arduino kommt ein **FreeRTOS** zum Einsatz, welches eine Echtzeit-Multitasking-Umgebung bereitstellt. Zur Vereinfachung und Strukturierung der Embedded-Software wird eine objektorientierte Bibliothek namens **frt.h** verwendet. Diese bietet eine klar strukturierte Klassenhierarchie für Tasks, Queues und andere RTOS-Funktionalitäten und erleichtert die Wartung und Erweiterung der Firmware erheblich.

Das Betriebssystem auf dem VAT uC ist zum jetzigen Zeitpunkt nicht bekannt. Muss noch herausfinden?????????

**Toolchain**

Da wir ein umfangreiches Embedded-Software-Projekt planten, reichten die Möglichkeiten der offiziellen Arduino IDE – weder in der alten noch in der neuen Version – für unsere Anforderungen aus. Besonders bei größeren Codebasen, mehreren Modulen und dem Wunsch nach feinerer Kontrolle über den Build-Prozess und Debugging stießen wir auf deutliche Grenzen. Deshalb habe ich mich entschieden, eine eigene, flexiblere Toolchain aufzubauen.

**IDE-Auswahl**

Im ersten Schritt testete ich die kommerzielle Arduino-Alternative **Programino**. Diese bietet eine für Arduino optimierte Oberfläche und einige nützliche Features, zeigte jedoch bei unseren Tests diverse Bugs. Zusätzlich waren Support und Lizenzmodell (Kostenpflichtigkeit) nicht überzeugend.

Daher habe ich mich bewusst gegen diese IDE entschieden.  
→ <https://programino.com>

Im zweiten Anlauf bin ich auf die Open-Source-IDE **Sloeber** gestoßen, welche auf **Eclipse CDT** basiert. Trotz einiger Eigenheiten der Eclipse-Umgebung überzeugte Sloeber durch eine bessere Integration in professionelle Entwicklungsprozesse, erweiterbare Projekteinstellungen und eine aktiv gepflegte Arduino-Integration.

Die kleinere Community bietet dennoch ausreichend Dokumentation und Hilfe bei Problemen.  
→ [https://eclipse.baeyens.it](https://eclipse.baeyens.it/index.shtml)

**Compiler, Build und Binary-Analyse**

Als Compiler kam **avr-gcc** in der Version 7.3.0-atmel3.6.1-arduino7 zum Einsatz – ein bewährter Compiler für AVR-Mikrocontroller. Dieser erlaubt es, standardisierte und optimierte ELF-Dateien (Executable and Linking Format) zu generieren. Die generierte ELF-Datei enthält neben dem eigentlichen Maschinencode auch Debug-Informationen, symbolische Verweise und Speicherzuweisungen – ideale Grundlagen für eine tiefere Analyse.

Um das erstellte Binary bzw. die ELF-Datei genauer zu untersuchen, nutze ich mehrere Tools aus der avr-binutils-Toolchain:

* **avr-objdump**  
  Mit diesem Tool kann man das ELF-File disassemblieren und den enthaltenen Maschinencode auf Assembler-Ebene sichtbar machen. Dies ist besonders hilfreich, um zu prüfen, welche Funktionen wohin gelinkt wurden und ob der Compiler erwartungsgemäß optimiert hat.
* **avr-nm**  
  Dieses Tool listet alle Symbole (Funktionen, Variablen) mit ihren Adressen auf. Damit lässt sich u. a. analysieren, wie der Speicher (Flash und RAM) belegt ist und ob bestimmte Funktionen oder globale Variablen zu viel Platz beanspruchen.
* **awk und size.awk (bzw. eigene Awk-Skripte)**  
  Diese Skripte dienen der automatisierten Auswertung der avr-nm- oder avr-size-Ausgaben. Ziel ist es, Speicherverteilung (z. B. .text, .data, .bss) präzise aufzulisten und ggf. Engpässe sichtbar zu machen. Mit diesen Tools lässt sich auch über Zeit hinweg die Speicherentwicklung im Projekt nachvollziehen.

Zusammengefasst erlaubt uns diese Toolchain – bestehend aus Sloeber, avr-gcc und den Analysewerkzeugen – ein professionelles, reproduzierbares und transparentes Embedded-Projektmanagement, das deutlich über das hinausgeht, was mit der Arduino IDE möglich wäre.

**WAS SIND „.INO“ files und warum verwenden wir diese?**

Beim Erstellen und Hochladen eines Sketches auf ein Arduino-Board durchläuft der Code mehrere Schritte. Dieser Prozess ist speziell für AVR-Architekturen beschrieben, gilt jedoch in ähnlicher Form auch für andere Plattformen.

**1. Vorverarbeitung des Sketches**

Bevor der Code kompiliert wird, führt die Arduino-IDE einige vorbereitende Schritte durch:

* Alle .ino- und .pde-Dateien im Sketch-Ordner werden zusammengeführt. Die Hauptdatei (gleicher Name wie der Ordner) wird zuerst verarbeitet.
* Die Datei wird in eine .cpp-Datei umgewandelt.
* Falls nicht vorhanden, wird #include <Arduino.h> hinzugefügt, um grundlegende Arduino-Funktionen verfügbar zu machen.
* Funktionsprototypen werden automatisch generiert, sofern sie nicht manuell vorhanden sind.
* #line-Direktiven werden eingefügt, um Fehlermeldungen auf die richtigen Zeilen im Originalcode zu beziehen.

Andere Dateitypen (.c, .h etc.) werden nicht automatisch verändert oder eingebunden.

**2. Auflösung von Abhängigkeiten**

Der Sketch wird nach eingebundenen Bibliotheken durchsucht. Die Arduino-IDE sucht diese zunächst in folgenden Verzeichnissen:

* Kernbibliothek und Board-spezifische Varianten
* Standard-Systemverzeichnisse
* Bereits aufgelöste Abhängigkeitspfade
* Installierte Bibliotheken im Benutzer- oder IDE-Verzeichnis

Falls mehrere Bibliotheken passen, entscheidet eine Prioritätenregel nach Kriterien wie Architekturkompatibilität, exakter Namensübereinstimmung oder Speicherort.

**3. Kompatibilität von Bibliotheken**

Eine Bibliothek gilt als **kompatibel**, wenn in ihrer library.properties-Datei die passende Architektur (avr, \*, oder leer) angegeben ist. Sie ist **optimiert**, wenn explizit nur avr eingetragen ist.

**4. Kompilierung**

Die eigentliche Kompilierung erfolgt in einem temporären Verzeichnis mit avr-gcc:

* Quellcode-Dateien (.cpp, .c, .S) und Bibliotheken werden zu Objektdateien (.o) kompiliert.
* Nur geänderte Dateien werden neu kompiliert (abhängig von Zeitstempeln und .d-Abhängigkeitsdateien).
* Alle Objektdateien werden gelinkt, unnötige Teile ausgelassen, um die Dateigröße zu minimieren.
* Ergebnis ist eine .hex-Datei – diese enthält den binären Code für das Arduino-Board.

**5. Hochladen auf das Board**

Der Upload erfolgt über ein plattformspezifisches Tool wie avrdude:

* Die .hex-Datei wird via USB oder serielle Schnittstelle (über den Bootloader oder Programmer) auf das Board übertragen.
* Bei aktiviertem „verbose mode“ wird der vollständige Befehlsausgabeprozess in der Konsole angezeigt.

QUELLE: <https://arduino.github.io/arduino-cli/1.2/sketch-build-process/>

QUELLE: <https://arduino.github.io/arduino-cli/1.2/platform-specification/>