Inhaltsverzeichnis

Verwendete Tools:

- Rechtschreibprüfung24, Mentor Duden
- ChatGPT, Zotero
- plantuml

WICHTIGE LINKS UND QUELLEN

Arduino:

- https://docs.arduino.cc/learn/programming/memory-guide/
- https://docs.arduino.cc/learn/programming/eeprom-guide/
- https://docs.arduino.cc/learn/starting-guide/whats-arduino/
- https://docs.arduino.cc/learn/starting-guide/software-libraries/
- https://docs.arduino.cc/learn/starting-guide/cores/
- https://docs.arduino.cc/learn/communication/uart/
- https://docs.arduino.cc/learn/communication/spi/
- https://docs.arduino.cc/libraries/ethernet/
- https://docs.arduino.cc/learn/communication/wire/
- https://docs.arduino.cc/learn/programming/reference/
- https://content.arduino.cc/assets/

AVR:

- https://www.microchip.com/en-us/product/atmega2560
- https://www.microchip.com/en-us/application-notes/an1497
- https://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf

RISC:

- https://www.stromasys.com/resources/decoding-risc-vs-cisc-architecture/
- https://linus5.blogspot.com/2016/07/risc-processor.html

FreeRTOS:

- https://www.freertos.org/Documentation/02-Kernel/04-API-references/
- https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/00-Developer-docs
- https://www.freertos.org/Documentation/02-Kernel/01-About-the-FreeRTOS-kernel/01-FreeRTOS-kernel

- https://www.freertos.org/Documentation/01-FreeRTOS-quick-start/01-Beginners-guide/00-Overview

Arduino_FreeRTOS:

- https://github.com/feilipu/Arduino_FreeRTOS_Library
- https://docs.arduino.cc/libraries/freertos/
- https://github.com/Floessie/frt
- https://oer-informatik.gitlab.io/mcu/arduino-esp/images/arduino-compile-prozess/build-prozess.png

Ethernetshield:

- https://www.mouser.com/catalog/specsheets/a000056_datasheet.pdf

Sensoren/Aktoren

- https://cdn-learn.adafruit.com/downloads/pdf/adafruit-mcp9601.pdf
- https://www.mouser.com/datasheet/2/758/DHT11-Technical-Data-Sheet-Translated-Version-1143054.pdf
- https://www.mouser.at/datasheet/2/734/MLX90614-Datasheet-Melexis-953298.pdf

Kommunikation:

- https://www.analog.com/en/resources/analog-dialogue/articles/introduction-to-spi-interface.html
- https://www.ti.com/lit/an/sbaa565/sbaa565.pdf?ts=1746559710707&ref_url=https%253A%252F%252Fwww.google.com%252F
- https://docs.arduino.cc/libraries/ethernet/

VAT:

- https://www.vatgroup.com/downloads

Informationen Dipomarbeit:

- https://www.diplomarbeiten-bbs.at/durchfuehrung/gliederung-der-diplomarbeit-und-formale-vorgaben
-

C/C++:

- https://devdocs.io/cpp/
- https://en.cppreference.com/w/
- https://cplusplus.com/doc/
- https://docs.arduino.cc/arduino-cloud/guides/arduino-c/
- https://docs.arduino.cc/language-reference/

Python:

- https://docs.python.org/3.9/

Docker:

- https://docs.docker.com/
- https://docs.docker.com/engine/install/raspberry-pi-os/

Github:

- https://docs.github.com/en
- https://git-scm.com/doc

Linux:

- https://docs.kernel.org/
- https://www.raspberrypi.com/documentation/

Libraryübersicht – 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
- **Wire** – Kommunikation mit I2C-Geräten
- **Ethernet** – Netzwerkkommunikation
- **ErriezMemoryUsage** – Überwachung des Speichers
- **FreeRTOS** – Betriebssystem für Multitasking, Zeitsynchronisierung etc.
- **Frt.h** – Objektorientierter Wrapper für FreeRTOS
- **MCP9601** – Library, um Temperatursensor auszuwerten

3. Entscheidungsgrundlage & Architekturprinzipien

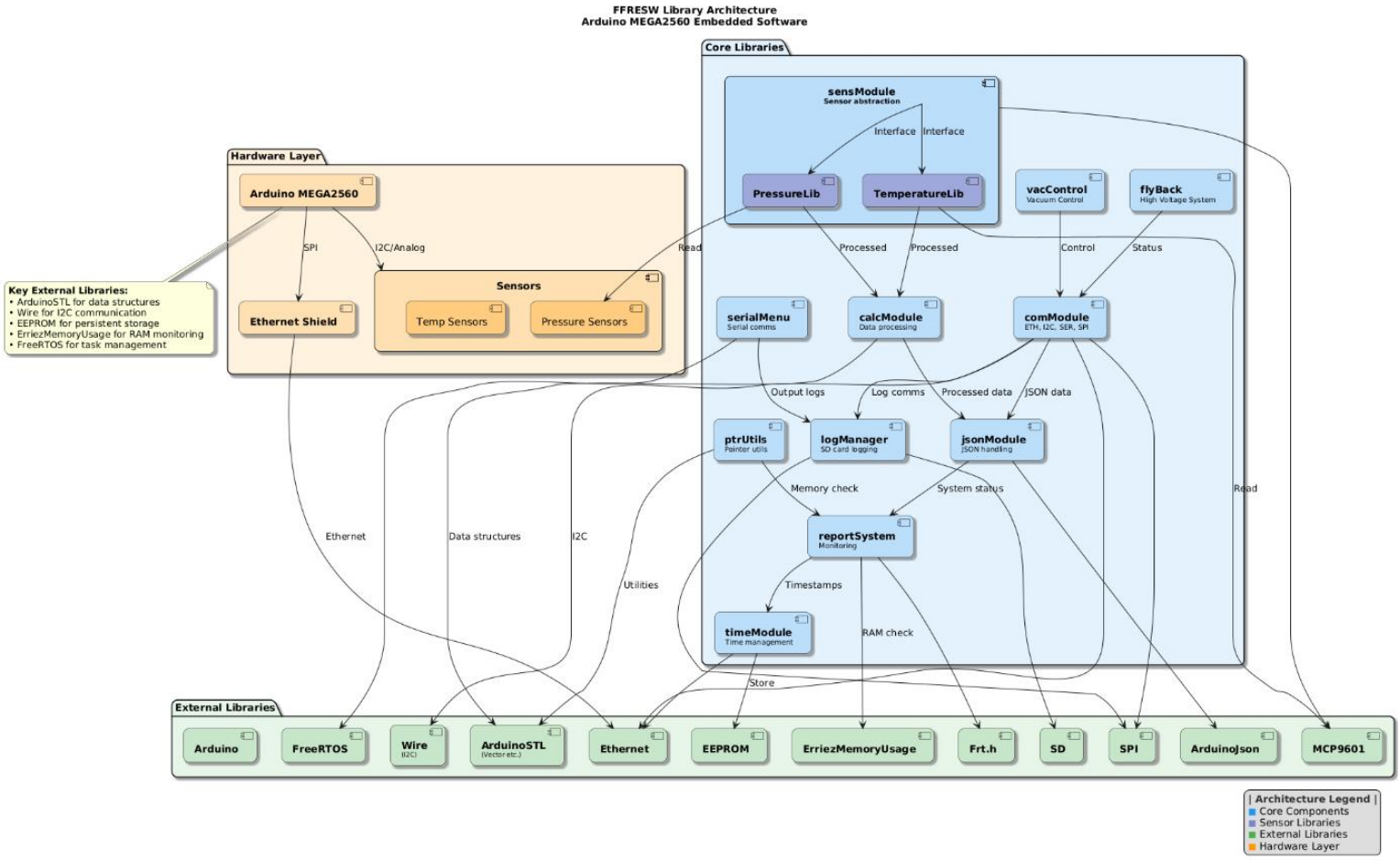
Die Entscheidung für C++ basiert auf den technischen Gegebenheiten des Arduino MEGA2560 und der nativen Unterstützung von .ino-Dateien. Das objektorientierte Design erleichtert:

- 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

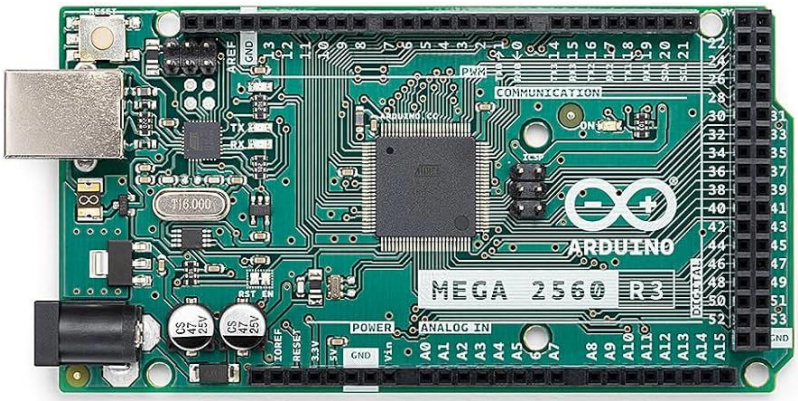


Hardwareübersicht FFRESW

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.



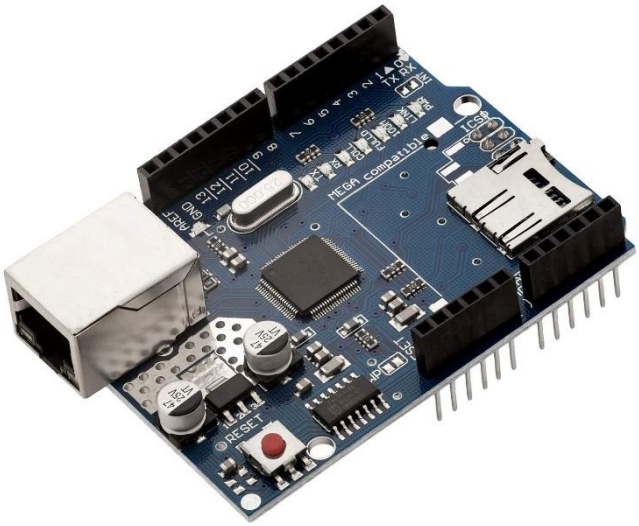
QUELLE: https://www.amazon.de/Arduino-Mega-2560-R3-Microcontroller/dp/B0046AMGW0

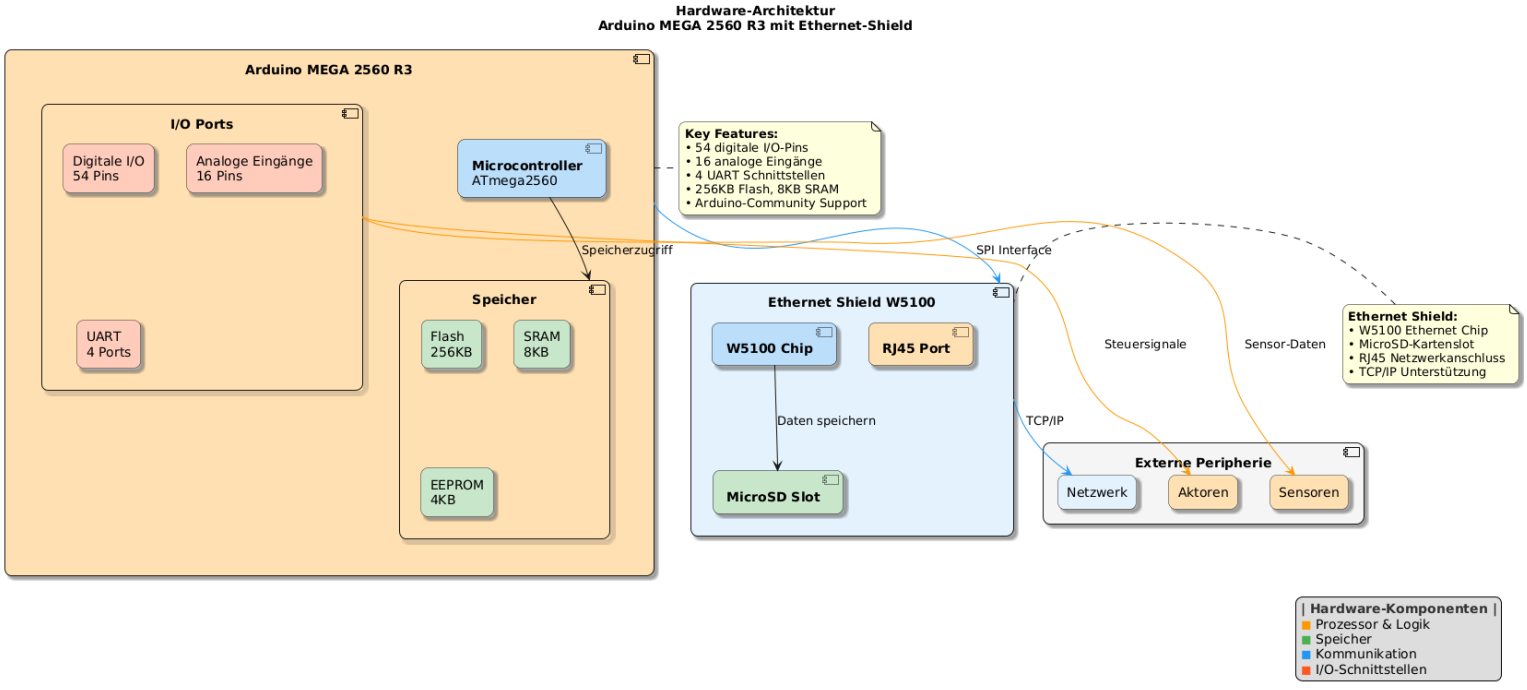
Ethernetshield W5100 von AZ Delivery

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.

Welche Architektur verwenden wir, welche Limitationen haben sich ergeben?

Der Mikrocontroller **Arduino Mega 2560 R3** basiert auf dem **ATmega2560**-Prozessor, der zur **AVR-Mikrocontrollerfamilie** von Atmel (heute Microchip Technology) gehört. Der ATmega2560 verwendet eine sogenannte **RISC-Architektur** (Reduced Instruction Set Computer), zu Deutsch: „Rechner mit reduziertem Befehlssatz“.

Bei einer RISC-Architektur handelt es sich um ein Designprinzip für Prozessoren, das auf einen **einfachen und effizienten Befehlssatz** setzt. Im Gegensatz zur **CISC-Architektur** (Complex Instruction Set Computer), die viele komplexe Maschinenbefehle mit teilweise mehreren Taktzyklen ausführt, verfolgt RISC das Ziel, **jeden Befehl möglichst schnell und in nur einem Taktzyklus** abzuarbeiten. Dadurch ist die Hardwarestruktur einfacher, was unter anderem zu einem geringeren Energieverbrauch und höherer Effizienz führen kann.

Vorteile der RISC-Architektur:

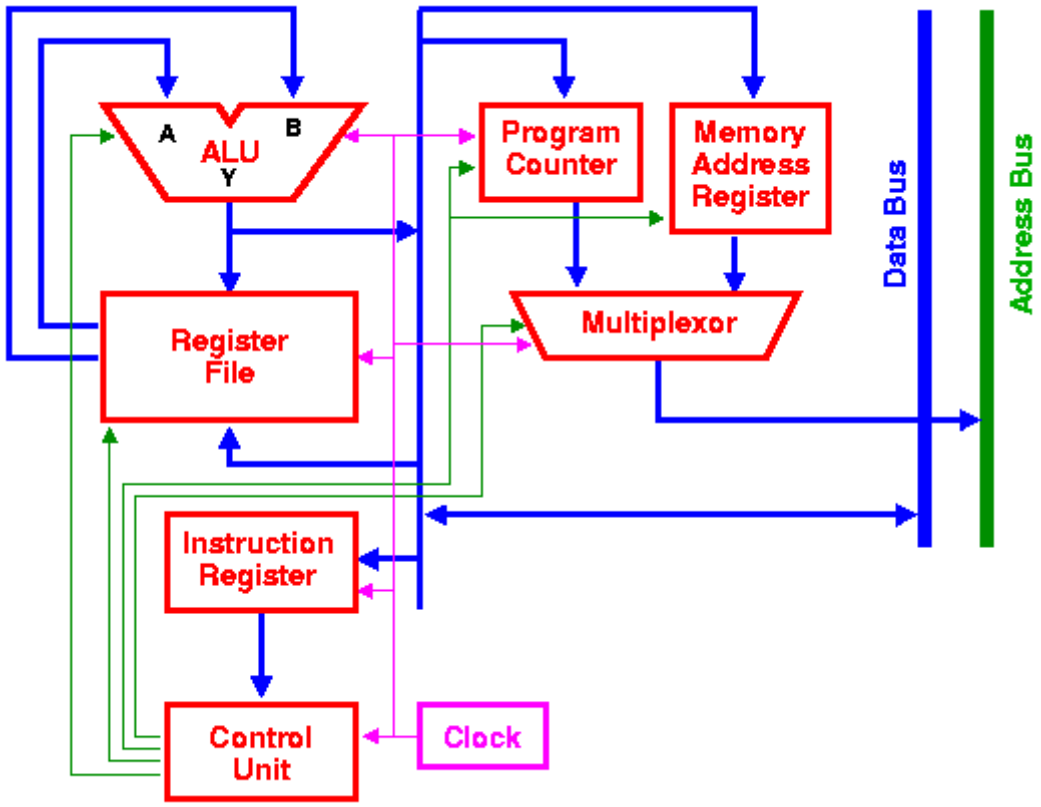
- **Schnellere Befehlsausführung**: Die meisten Befehle benötigen nur einen Taktzyklus.
- **Einfacherer Hardwareaufbau**: Der reduzierte Befehlssatz ermöglicht eine effizientere Prozessorentwicklung.
- **Geringerer Energieverbrauch**: Ideal für eingebettete Systeme und mobile Anwendungen.

- **Bessere Optimierungsmöglichkeiten**: Der Compiler kann den Code leichter und effizienter umsetzen.

Nachteile der RISC-Architektur:

- **Mehr Befehle notwendig**: Komplexere Aufgaben erfordern mehrere einfache Befehle, was den Codeumfang erhöhen kann.
- **Größerer Speicherbedarf für Programme**: Da mehr Einzelbefehle notwendig sind, kann das Programm insgesamt mehr Speicherplatz benötigen.
- **Komplexität beim Compilerdesign**: Die Intelligenz muss teilweise vom Prozessor auf die Software (Compiler) verlagert werden.

Der Arduino Mega 2560 R3 profitiert durch die RISC-Architektur des ATmega2560 von hoher Effizienz, einfacher Programmierung und guter Eignung für Echtzeitanwendungen. Trotz des größeren Codeumfangs bei komplexeren Aufgaben ist die RISC-Architektur besonders für Mikrocontroller-Systeme wie Arduino ideal geeignet, da sie Geschwindigkeit und Energieeffizienz optimal miteinander vereint.



QUELLE: https://linus5.blogspot.com/2016/07/risc-processor.html

Der ATMEGA2560 ist ein **8-Bit** Prozessor, mit Speicheradressen oder andere Dateneinheiten, die maximal 8 Bit (1 Oktett) breit sind. Ebenso basieren **8-Bit-CPU- und ALU-Architekturen** auf Registern, Adressbussen oder Datenbussen dieser Größe.

Harvard oder Von Neumann?

Der **Arduino Mega 2560**, basierend auf dem **ATmega2560-Mikrocontroller**, verwendet die sogenannte **modifizierte Harvard-Architektur**. Diese Architekturform bietet **getrennte Speicher- und Datenbusse** für **Programmcode (Instruktionen)** und **Nutzdaten**, was mehrere wichtige Vorteile gegenüber der klassischen Von-Neumann-Architektur mit sich bringt.

In der Harvard-Architektur sind **Programmspeicher (Flash)** und **Datenspeicher (SRAM/EEPROM)** physikalisch getrennt. Das bedeutet, dass **Instruktionen und Daten gleichzeitig und unabhängig voneinander** gelesen oder geschrieben werden können. Der ATmega2560 nutzt intern **separate Busse**, um Instruktionen aus dem Flash-Speicher und Daten aus dem SRAM zu laden. Dies verbessert die **Ausführungsgeschwindigkeit**, da kein Engpass durch geteilte Speicherzugriffe entsteht, wie es bei der Von-Neumann-Architektur der Fall wäre.

Ein weiterer typischer Aspekt der Harvard-Architektur ist, dass **Programmspeicher nicht direkt als Datenspeicher genutzt** werden kann – der Zugriff auf Programminhalte als Daten muss explizit über spezielle Funktionen (z. B. pgm_read_byte in C/C++) erfolgen. Dadurch ist das System sicherer und weniger anfällig für unbeabsichtigte Speicherzugriffe.

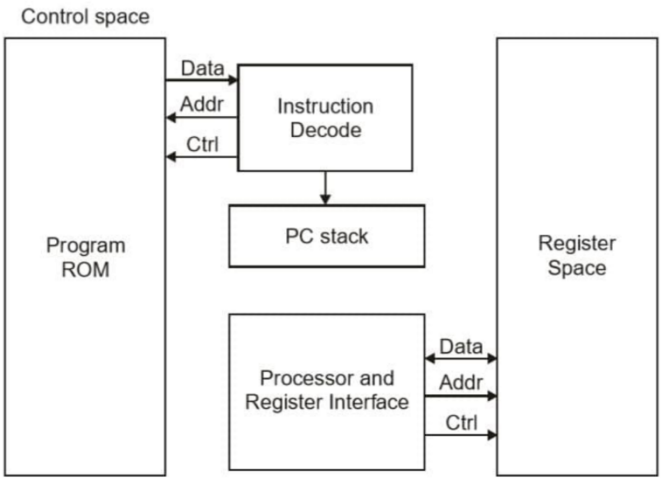
In der klassischen Harvard-Architektur war es zudem üblich, dass Programme **manuell vom Benutzer geladen werden mussten**, da der Prozessor selbst nicht über die Fähigkeit verfügte, ein Programm aus dem Speicher zu starten. Moderne Mikrocontroller wie der ATmega2560 hingegen verfügen über integrierte **Bootloader**, die es ermöglichen, Programme automatisch aus dem Flash-Speicher zu starten – ein praktisches Merkmal, das die Benutzerfreundlichkeit deutlich erhöht.

Vorteile der (modifizierten) Harvard-Architektur im Arduino Mega 2560:

- **Paralleler Zugriff** auf Instruktionen und Daten → schnellere Programmausführung.
- **Klare Trennung von Code und Daten** → höhere Systemsicherheit.
- **Effizienter Speicherzugriff** durch spezialisierte Busarchitektur.

Nachteil:

- **Komplexere Speicherverwaltung**, da der Zugriff auf Programm- und Datenspeicher unterschiedlich gehandhabt werden muss.



QUELLE:https://fabacademy.org/2024/labs/sanjivani/students/akash-mhais/Week%206%20Group%20Assignment.html

Agiles Arbeiten, Teamwork und Kollaboration was haben wir wie angewendet, welche Vorteile und Nachteile haben sich ergeben?

Durch meine Erfahrungen in der Softwareentwicklung war von Anfang an klar: **Um effizient und im Team zu programmieren – und das auf iterative Weise – benötigen wir einen agilen Workflow**, der es uns erlaubt, **schnell und flexibel auf Veränderungen zu reagieren**. Besonders in einem interdisziplinären Projekt, in dem sowohl Hardware- als auch Softwareentwicklung ineinandergreifen, ist eine klare Struktur und gute Zusammenarbeit entscheidend. Um dies zu ermöglichen, haben wir **verschiedene Tools und Methoden eingesetzt**, die uns bei der Organisation und Durchführung der Arbeit unterstützt haben.

Für die **Dokumentation, Zusammenarbeit und das Teilen von Informationen** haben wir **Notion** verwendet.
Notion ist ein vielseitiges, cloudbasiertes Tool für Wissensmanagement, Aufgabenverwaltung und Teamkollaboration. Es vereint Funktionen wie Notizbücher, Datenbanken, Aufgabenboards und Wikis in einer zentralen Plattform. In Notion haben wir **mehrere Arbeitsbereiche („Spaces“) eingerichtet**, darunter:

- **Hardwareentwicklung**
- **Softwareentwicklung**
- **Wissensdatenbank (Knowledge Base)**
- **Projektplanung**

Innerhalb dieser Spaces haben wir wiederum Inhalte strukturiert abgelegt, wie z. B. technische Dokumentationen, Teamabsprachen und Meetingnotizen. Außerdem haben wir **Verknüpfungen zu weiteren Cloud-Diensten** eingebunden, zum Beispiel:

- **Google Drive** für Tabellen und Präsentationen
- **OneDrive** zur gemeinsamen Bearbeitung von Word-Dokumenten (z. B. für Pflichtenhefte, Berichte oder Dokumentationen)

Für die **konkrete Aufgabenverwaltung im Projektverlauf**, insbesondere im Zusammenhang mit der Software- und Hardwareentwicklung, haben wir ein **GitHub Projektboard** verwendet. Dieses basierte auf dem **Kanban-Prinzip** und bot eine übersichtliche Einteilung in:

- **Backlog** (alle anstehenden Aufgaben)
- **To-Do** (geplante nächste Schritte)
- **In Progress** (aktuell bearbeitete Aufgaben)
- **In Review** (Aufgaben in der Prüfung)
- **Done** (abgeschlossene Aufgaben)

Diese Struktur ermöglichte uns eine **transparente Aufgabenverteilung und eine klare Nachverfolgbarkeit** des Projektfortschritts. Jeder im Team konnte jederzeit den aktuellen Stand einsehen und wusste, woran gerade gearbeitet wird oder welche Aufgaben noch offen sind.

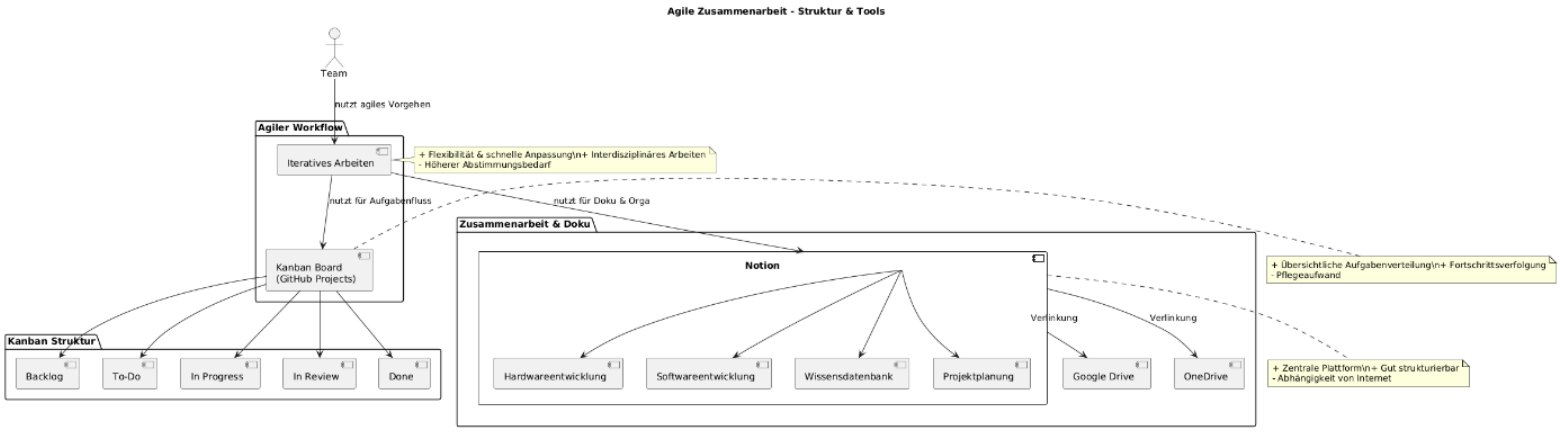
Durch diese **agile und digitale Organisation** war es **problemlos möglich, auch remote oder flexibel von zu Hause aus zu arbeiten**. Die Zusammenarbeit funktionierte sowohl asynchron als auch synchron einwandfrei – ob im Büro, im Homeoffice oder während Online-Meetings. Insgesamt hat dieser Workflow unsere Effizienz gesteigert, die Kommunikation verbessert und die Qualität unserer Ergebnisse positiv beeinflusst.

QUELLE: https://upload.wikimedia.org/wikipedia/commons/4/45/Notion_app_logo.png

QUELLE: https://icon-icons.com/de/symbol/github-logo/181401

QUELLE:https://de.m.wikipedia.org/wiki/Datei:Microsoft_Office_OneDrive_%282019%E2%80%93present%29.svg

QUELLE: https://de.wikipedia.org/wiki/Datei:Google_Drive_icon_(2020).svg



GitHub und Repositorys für was das ganze?

Um eine **Versionsverwaltung** zu gewährleisten und **effizient gemeinsam am Code arbeiten** zu können, haben wir uns für **GitHub** als Plattform entschieden. Da ich **privat und beruflich bereits umfassende Erfahrung mit GitHub** habe – insbesondere im Umgang mit Workflows, Konfigurationen und Integrationen – habe ich bewusst den **Vorrang gegenüber GitLab** gegeben.

Im Rahmen des Projekts habe ich **drei separate Repositories** eingerichtet, die jeweils unterschiedliche Aufgabenbereiche abdecken:

FFRESW – *Farnsworth Fusion Reactor Embedded Software*

GitHub-Link

Dieses Repository enthält die **gesamte Embedded-Software** unseres Projekts, inklusive:

- Quellcode in **C/C++**, **Python**, **Shell** und **Bash**
- Dokumentation mit **Doxygen**
- Hilfsskripte und Build-Tools

Es dient als zentrale Codebasis für alle Mikrocontroller-Anwendungen im Projekt.

FFRHAS – *Farnsworth Fusion Reactor Hardware Access Service*

GitHub-Link

Dieses Repository beinhaltet die Software, die auf dem **Raspberry Pi** innerhalb eines **Docker-Containers** läuft. Der Fokus liegt auf der **Visualisierung und Hardwareansteuerung**, unter Verwendung folgender Technologien:

- **JavaScript**, **Python**, **HTML**, **CSS**
- **GoJS**: Ein spezialisiertes JavaScript-Framework zur Erstellung interaktiver Diagramme, ideal für **SCADA-ähnliche Anwendungen** GoJS Framework

rpi-docker-setup

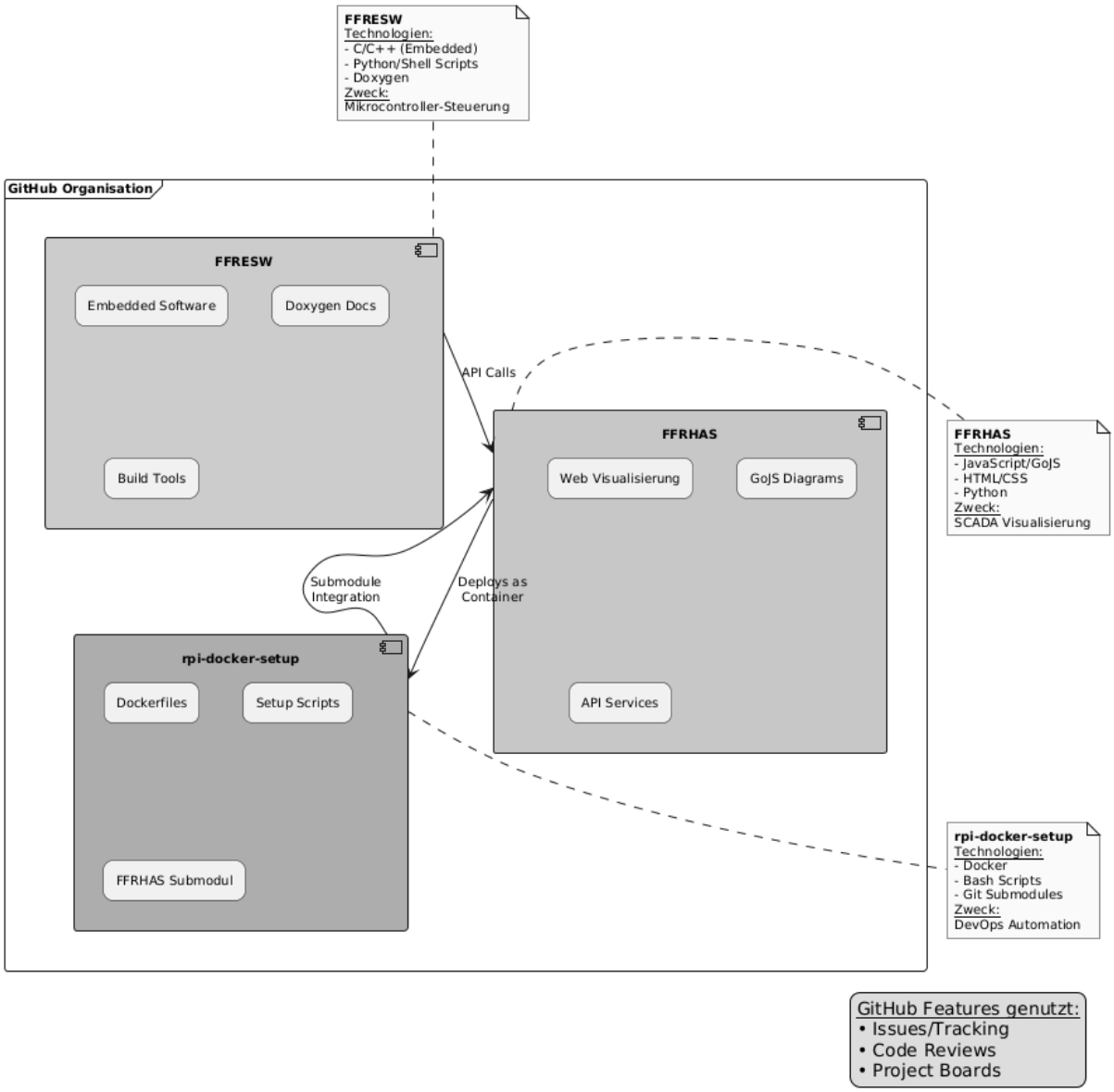
GitHub-Link

Dieses Repository enthält:

- **Shell-Skripte** für die Einrichtung und Verwaltung der Raspberry-Pi-Umgebung
- **Dockerfiles** zur Containerisierung der Services
- Das **FFRHAS-Repository als Git-Submodul**, um eine enge Kopplung und automatisierte Bereitstellung zu ermöglichen

Durch die klare Aufteilung der Repositories nach Zuständigkeiten konnten wir die Entwicklungsprozesse **modular, nachvollziehbar und gut wartbar** gestalten. GitHub diente dabei nicht nur als Versionsverwaltung, sondern auch als Plattform für **Issue-Tracking**, **Code-Review** und **kontinuierliche Integration**.

GitHub Repository Struktur für Farnsworth Fusion Reactor Projekt



Docker Dockerimages warum nicht baremetal?

In unserem Projekt verwenden wir **Docker-Images**, anstatt die gesamte Softwareumgebung manuell auf dem Raspberry Pi (Bare-Metal) zu installieren. Doch warum dieser zusätzliche Aufwand? Schließlich könnte man theoretisch auch alle Komponenten direkt auf dem Host-Betriebssystem installieren und konfigurieren.

Die Antwort liegt in den Anforderungen an **Wartbarkeit, Wiederholbarkeit und Agilität**.

Warum nicht Bare-Metal?

Wenn man Software direkt auf dem Host (z. B. Raspberry Pi OS) installiert – also „Bare-Metal“ –, läuft man schnell in typische Probleme:

- Unterschiedliche Paketversionen oder Bibliotheken auf verschiedenen Geräten
- Fehleranfällige manuelle Konfiguration
- Schwer nachvollziehbare Abhängigkeiten
- Keine automatische Isolierung zwischen Diensten

Das widerspricht unseren Zielen, da wir:

- **Agil arbeiten** wollen (schnelle Iterationen, Tests, Änderungen),
- **sichere und wiederholbare Umgebungen** benötigen,
- und eine **problemlose Zusammenarbeit im Team** ermöglichen müssen.

Was ist Docker?

Docker ist eine Plattform zur Erstellung, Ausführung und Verwaltung von sogenannten **Containern**. Ein Container ist eine **isolierte Umgebung**, die den benötigten Code, Abhängigkeiten und Konfigurationen enthält – ähnlich wie eine Mini-VM.

Vorteile von Docker im Überblick:

Vorteil	Bedeutung im Projektkontext
Reproduzierbarkeit	Jede Umgebung ist identisch, unabhängig vom Gerät
Leichtgewichtig & schnell	Startet in Sekunden, kein Overhead wie bei VMs
Portabilität	Ein Image funktioniert auf jedem Docker-fähigen Host
Isolation	Services stören sich nicht gegenseitig
Versionsverwaltung	Images und Konfigurationen sind versionierbar

Minimal und effizient

Die von uns eingesetzten Docker-Container basieren auf schlanken Images wie python:3.9-slim, das auf einem minimalen Debian-Buster-System aufbaut. Innerhalb dieser Container laufen alle projektspezifischen Komponenten wie die Flask-Webanwendung, Datenbank (MongoDB) und Visualisierung (Grafana). Durch die Trennung vom Host-System bleibt dieses schlank, stabil und frei von projektspezifischen Abhängigkeiten. Gleichzeitig sorgt Docker für eine konsistente, reproduzierbare und leicht wartbare Umgebung – unabhängig davon, auf welchem Raspberry Pi das System ausgeführt wird.

CI/CD Automatisierung, Github Actions und Deployment

Für zuverlässige, fehlerminimierte Binaries und Hex-Dateien habe ich eine automatisierte Pipeline mittels GitHub Actions implementiert. Diese gewährleistet **Qualitätssicherung und frühe Fehlererkennung**, indem sie bei jedem Push oder Pull Request ausgeführt wird.

Kernfunktionen der Pipeline:

1. **Kompilation & Statische Analyse**
   Mittels der **Arduino-CLI-Toolchain** wird der Code kompiliert, wobei Warnungen und Fehler direkt erkannt werden. Zusätzlich wird mit size.awk der Speicherverbrauch analysiert, um sicherzustellen, dass nach dem Flashen des Controllers ausreichend Ressourcen verfügbar sind.
2. **ELF-Binary-Analyse**
   Die Pipeline nutzt **Binutils**, um detaillierte Reports der generierten ELF-Dateien im Markdown-Format zu erstellen. Dies unterstützt:
   - **Fehlerdiagnose** (z. B. unerwartete Symbolbelegungen)
   - **Performance-Optimierung** (Sektionenanalyse)
   - **Plattformkompatibilität** (Header-Checks)
3. **Automatisiertes Artefakt-Management**
   Alle Build-Artefakte (Hex-Dateien, Binaries, Reports) werden als **GitHub Actions-Artefakte** bereitgestellt. Dies ermöglicht:
   - **Einfachen Zugriff** für alle Teammitglieder, auch ohne Entwicklungsumgebung
   - **Branch-unabhängiges Testen** (paralleles Arbeiten mit unterschiedlichen Mikrocontroller-Features)
   - **Reproduzierbarkeit** (jeder Build ist dokumentiert und archiviert)

Vorteile gegenüber manuellen Prozessen:

- **Unabhängigkeit von lokalen Entwicklungsrechnern**
  Keine Abhängigkeit von individuellen Laptop-Konfigurationen – die Pipeline läuft konsistent in einer kontrollierten Umgebung.
- **Transparente Qualitätskontrolle**
  Durch automatische Reports und Artefakt-Dokumentation wird die Code-Qualität objektiv nachvollziehbar.
- **Effizientes Teamwork**
  Hardware-Entwickler können jederzeit aktuelle Firmware-Versionen flashen, ohne die Software-Toolchain zu benötigen.

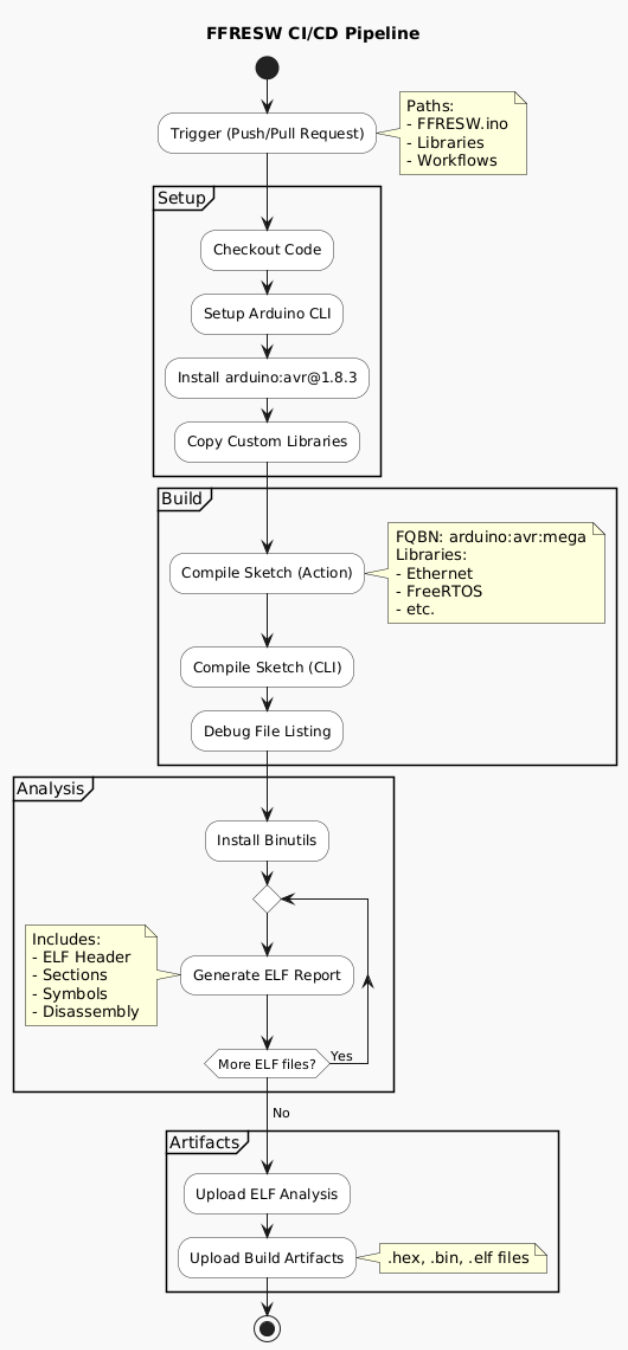
Diese Lösung **reduziert Risiken** durch manuelle Build-Fehler und **beschleunigt** iterative Entwicklungszyklen – besonders kritisch in interdisziplinären Projekten mit Hardware-Software-Schnittstellen.

Den Prozess von Push/Pull bis hin zu den generierten Build-Artefakten habe ich in diesem **UML-Flussdiagramm** dargestellt, um den Ablauf abstrakter zu visualisieren.

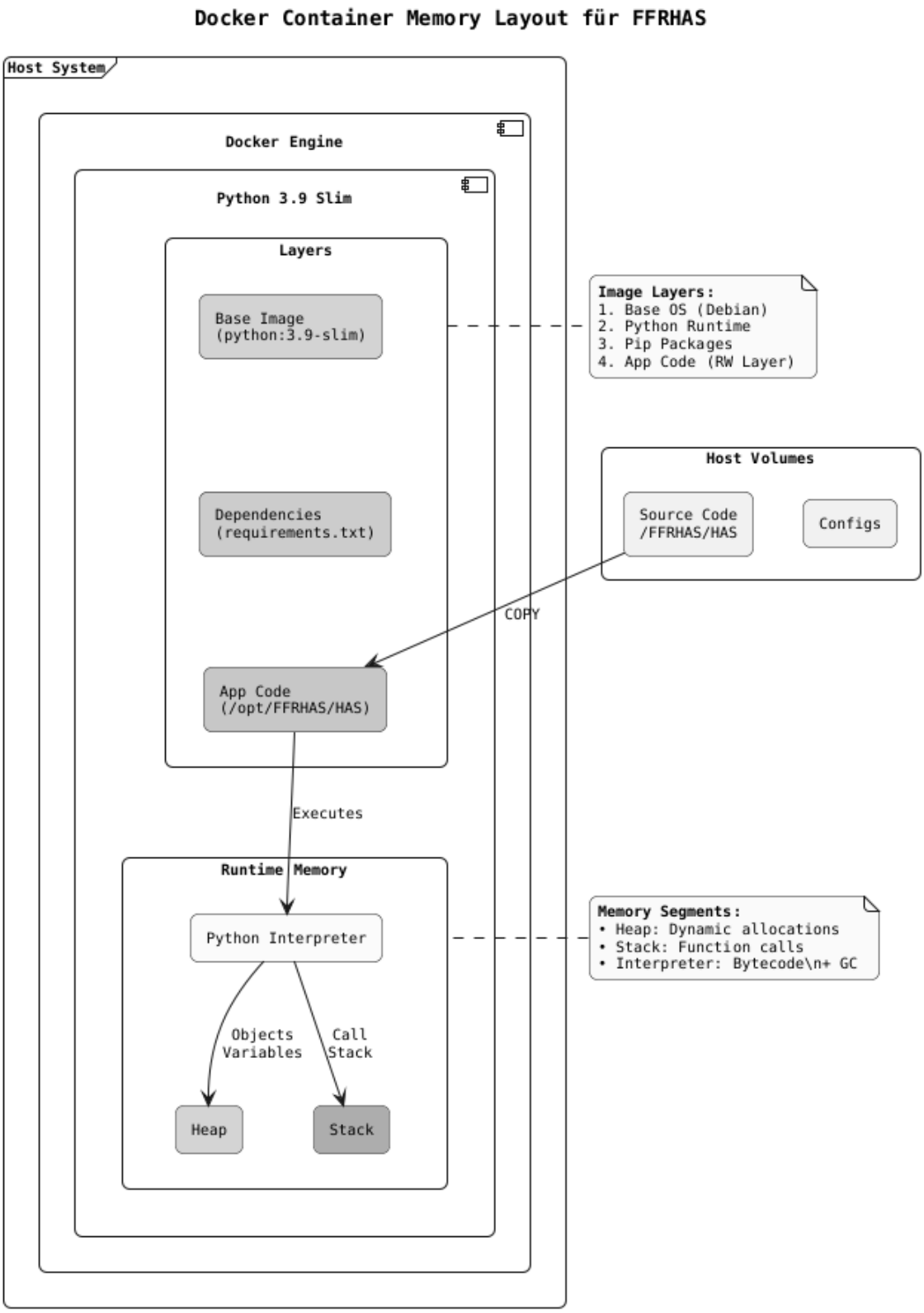
Die **Pipeline** (GitHub Action) wurde mittels einer **YAML-Datei** definiert. **YAML** (*Yet Another Markup Language*) ist eine Skriptsprache, die häufig im **DevOps-Kontext** verwendet wird, um Prozesse zu automatisieren und zu konfigurieren.

Wichtige Eigenschaften von YAML:

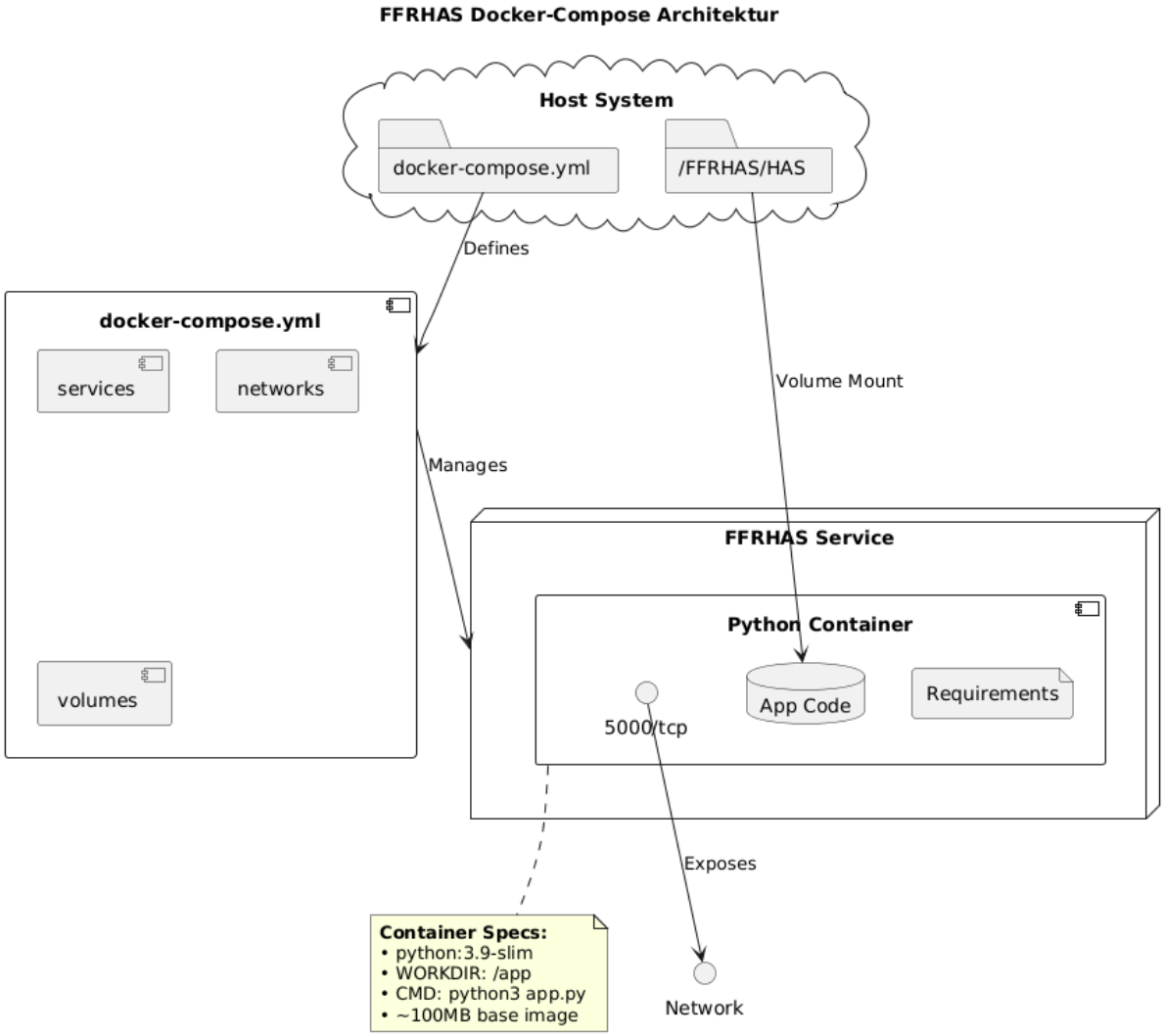
- **Striktes Superset von JSON** – ermöglicht komplexere Strukturen bei klarer Syntax.
- **Einfach zu schreiben und lesen** – durch Einrückungen und übersichtliche Formatierung.
- **Weltweit verbreitet** – Standard für CI/CD-Konfigurationen (z. B. GitHub Actions, GitLab CI).



Docker Container Memory Layout für HAS



FFRHAS Docker-Compose Architektur



FlowDiagramme/DoxyGen/UML

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.

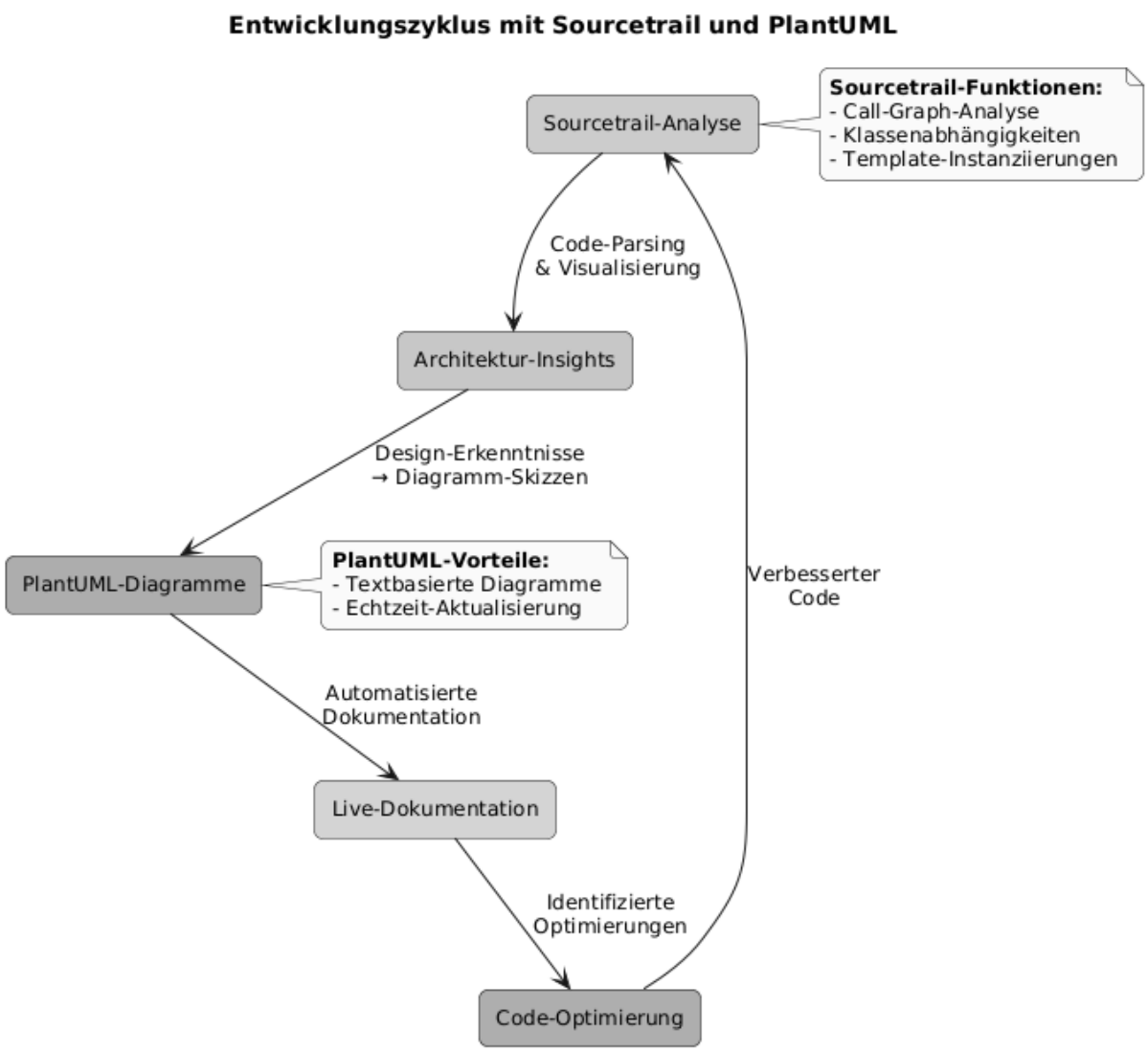
Darüber hinaus haben wir uns entschieden, die mit Sourcetrail generierten Diagramme direkt in unsere Dokumentation zu übernehmen, da sie sowohl **qualitativ hochwertig** sind als auch **einfach aktualisiert und angepasst** werden können, wenn sich der Code weiterentwickelt.

Für die Darstellung im UML-Format habe ich mich für **PlantUML** entschieden. Der wesentliche Vorteil dieses Tools liegt darin, dass UML-Diagramme nicht per Drag-and-Drop erstellt werden müssen, sondern über eine leicht verständliche **Skriptsprache** beschrieben werden. Dadurch lassen sich Diagramme effizienter und konsistenter erstellen – insbesondere bei komplexeren Modellen oder wiederkehrenden Strukturen. Zudem bietet die textbasierte Herangehensweise eine deutlich höhere **Kontrolle** über Layout, Inhalte und Formatierung. Änderungen können schnell nachvollzogen, versioniert und bei Bedarf automatisiert erzeugt werden. Das spart nicht nur Zeit, sondern unterstützt auch eine präzise und reproduzierbare Dokumentation der Softwarearchitektur.

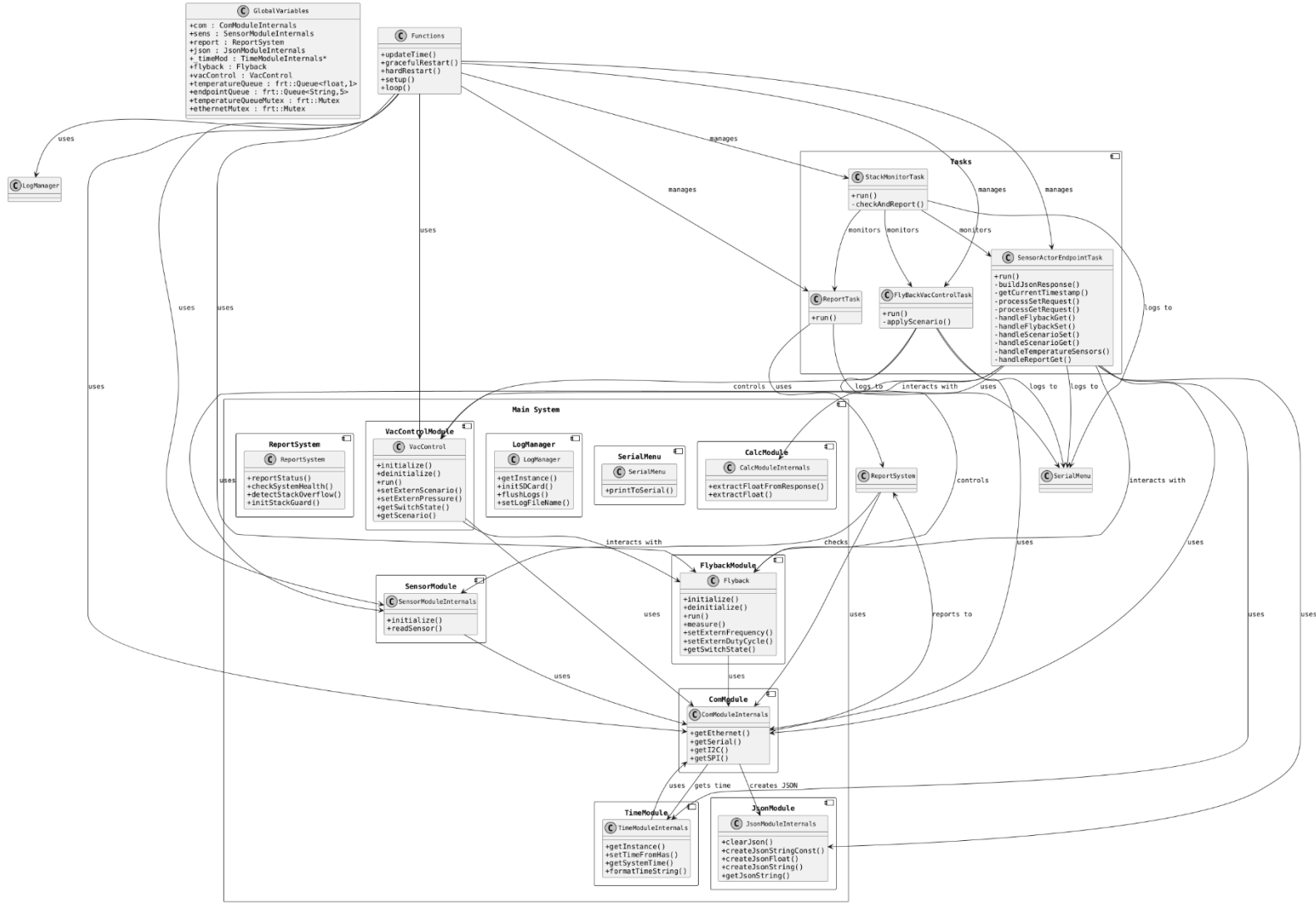
Leider wurde das ursprüngliche Projekt von **Coati Software** eingestellt, ist aber mittlerweile durch einen Fork aktiv weitergeführt worden:

- 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.


Entwicklungszyklus mit Sourcetrail und PlantUML
Sourcetrail-Analyse
Sourcetrail-Funktionen:
- Call-Graph-Analyse
- Klassenabhängigkeiten
- Template-Instanziierungen
Code-Parsing
& Visualisierung
Architektur-Insights
Design-Erkenntnisse
→ Diagramm-Skizzen
PlantUML-Diagramme
PlantUML-Vorteile:
- Textbasierte Diagramme
- Echtzeit-Aktualisierung
Verbesserter
Code
Automatisierte
Dokumentation
Live-Dokumentation
Identifizierte
Optimierungen
Code-Optimierung

Architekturübersicht FFRESW



1. **Systemarchitektur-Übersicht**
   Das System ist ein eingebettetes Steuerungssystem für industrielle Anwendungen mit folgenden Hauptkomponenten:

- Kommunikationsmodule (Ethernet, I2C, SPI, Serial)
- Sensormodule für Temperatur- und Druckmessung
- Hochspannungssteuerung (Flyback) und Vakuumkontrolle (VacControl)
- Echtzeit-Reporting und Systemüberwachung
- Aufgabenverwaltung durch RTOS-Tasks

2. **Kernprozesse im Betrieb**

A. **Initialisierungsphase (Setup)**

- Hardwarekomponenten werden initialisiert (Sensoren, Flyback, VacControl)
- Kommunikationsschnittstellen werden gestartet (Ethernet mit fester IP, I2C, SPI)
- Logging-System wird auf SD-Karte eingerichtet
- Systemzeit wird vom HAS (Host Automation System) abgefragt
- Alle Tasks werden gestartet mit unterschiedlichen Prioritäten

B. **Echtzeit-Task-Verarbeitung**

- **ReportTask**: Sendet alle 5 Sekunden Systemstatus und führt Health-Checks durch
  - Prüft Ethernet, SPI, I2C, Temperatursensoren, Drucksensoren
  - Erkennt Stack-Overflows
  - Loggt kritische Zustände über SerialMenu
- **FlyBackVacControlTask**: Regelungsschleife (1s Zyklus)

    - Steuert Hochspannungserzeugung (Frequenz/Duty-Cycle)
    - Verwaltet Vakuum-Szenarien (5 vordefinierte Betriebsmodi)
    - Kopplung mit externen Druckwerten über Ethernet
- **SensorActorEndpointTask**: Bearbeitet eingehende Kommandos
    - Verarbeitet HTTP-Endpoints (get/set Operationen)
    - Handhabt JSON-Antwortgenerierung mit Zeitstempeln
    - Unterstützt Szenariensteuerung und Sensordatenabfrage
- **StackMonitorTask**: Überwacht Task-Ressourcen
    - Prüft Stack-Auslastung aller Tasks (80% Threshold)
    - Warnmeldungen bei kritischer Auslastung

3. **Datenflüsse**

- Sensordaten → JSON-Modul → Ethernet (HTTP Responses)
- HAS-Kommandos → Ethernet → Szenariensteuerung → Aktoren
- Systemzustände → LogManager (SD-Karte) und Serial-Ausgabe

4. **Kritische Systeminteraktionen**

- **Vakuumsteuerung**:
- Ethernet (Druckwerte) → VacControl → Flyback (HV-Anpassung)
- **Notfallroutinen**:
    - Graceful Restart: Geordnetes Herunterfahren aller Komponenten
    - Hard Restart: Watchdog-ausgelöster Reset

5. **Sicherheitsmechanismen**

- Mutexe für gemeinsame Ressourcen (Temperaturdaten, Ethernet)
- Stack-Guard zur Überlaufserkennung
- Zyklische Health-Checks aller Subsysteme

6. **Typischer Betriebszyklus**
7. Zeitabgleich mit HAS
8. parallele Ausführung der Tasks
9. kontinuierliche Überwachung durch StackMonitor
10. bedarfsgesteuerte Aktorregelung
11. zyklische Systemdiagnose

Das System zeigt eine typische RTOS-Architektur mit:

- strikter Aufgabentrennung
- prioritätsbasierter Abarbeitung
- hardwarenaher Echtzeitsteuerung
- umfassendem Fehlermanagement

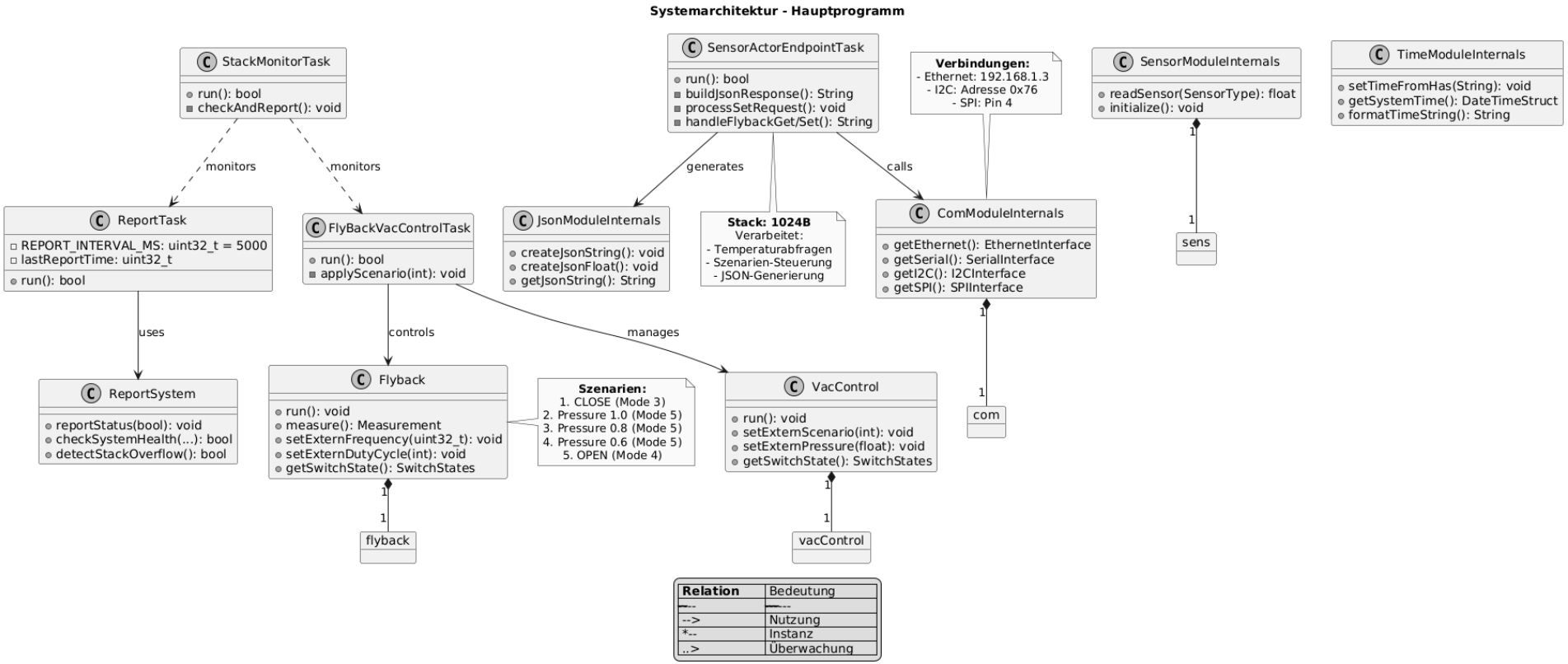
Die dargestellten Abhängigkeiten verdeutlichen die komplexe Interaktion zwischen:

- Echtzeitsteuerung (Flyback/VacControl)
- Kommunikationsschnittstellen
- Überwachungssubsystemen
- zentralem Task-Management

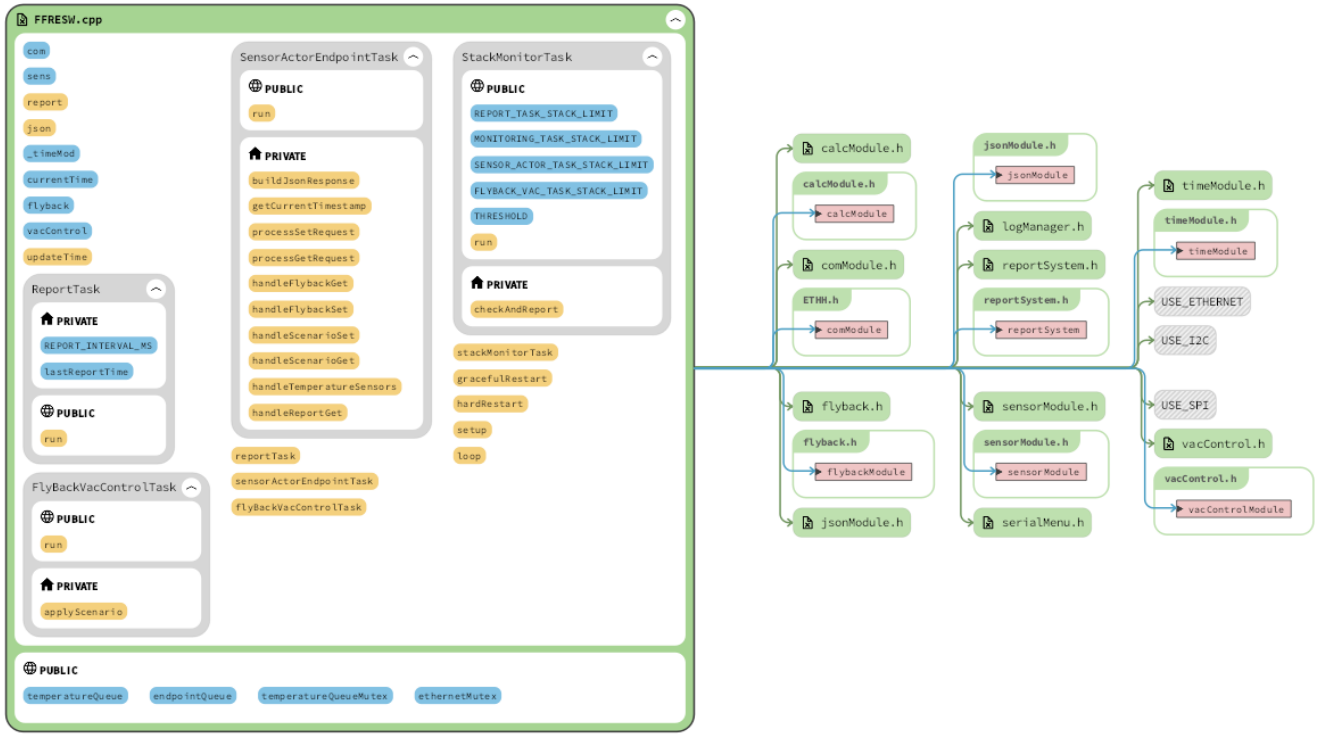
Jede Komponente ist so entworfen, dass sie:

- unabhängig getestet werden kann
- klar definierte Schnittstellen hat
- bei Fehlern das Gesamtsystem nicht blockiert

VEREINFACHTE ANSICHT:



FFRESW.ino

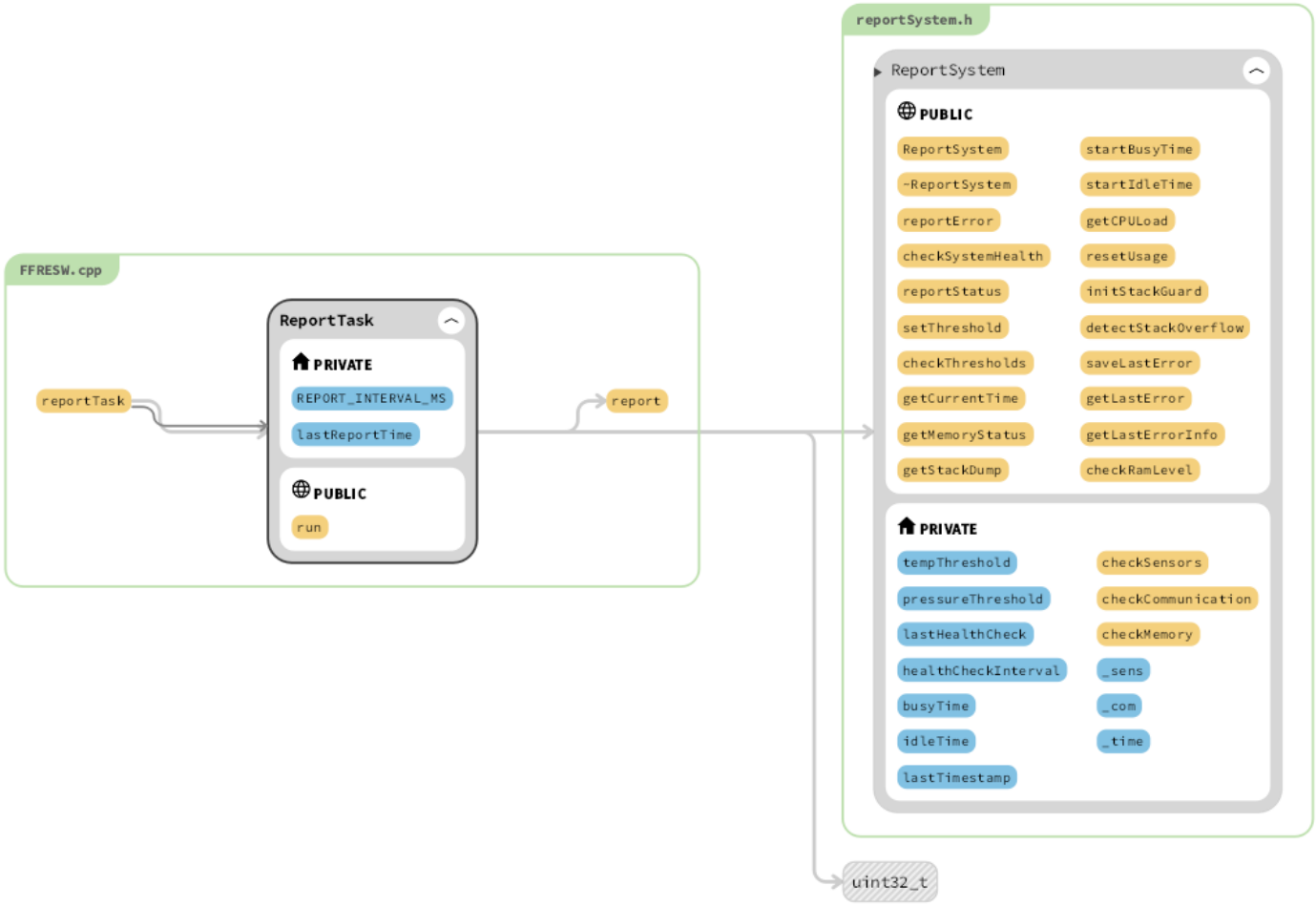


In diesem Abschnitt betrachten wir die Hauptdatei unseres Projekts, **FFRESW.ino**. Bei Arduino-Projekten ist die Dateiendung .ino standardisiert für sogenannte *Sketches*. Warum diese Endung verwendet wird und wie die Arduino-IDE sie verarbeitet, wurde bereits im entsprechenden Abschnitt erläutert: [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 mehrere parallellaufende Tasks ermöglicht.

ReportTask

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

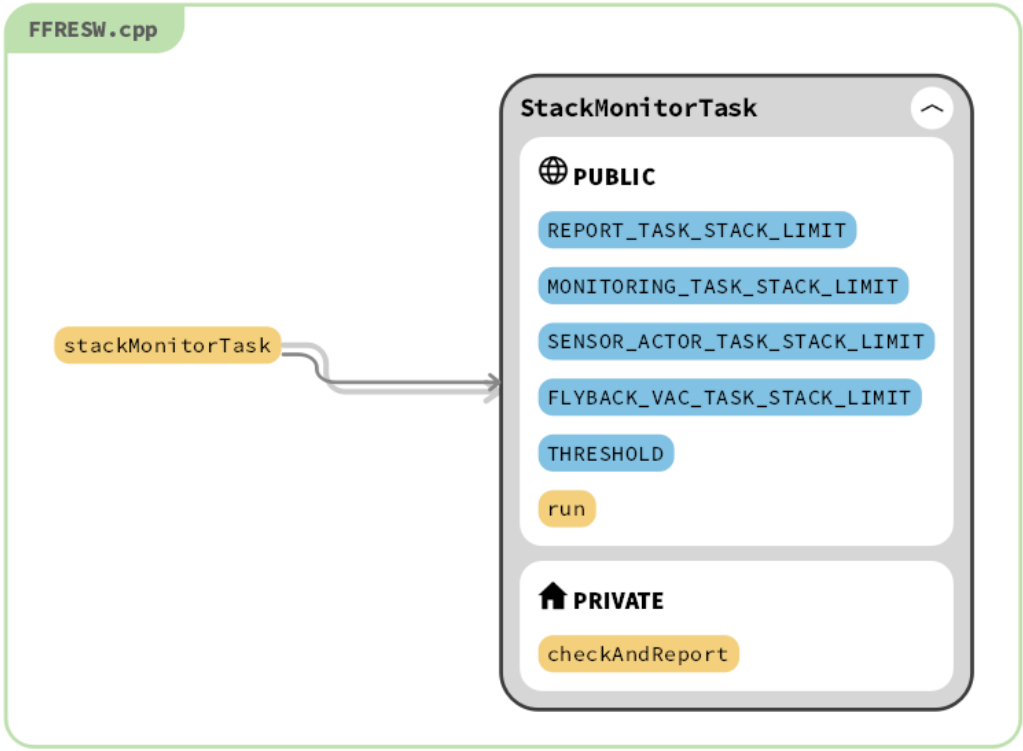


```cpp
/// @brief Class-Task to report system health and status periodically \class ReportTask
class ReportTask final : public frt::Task<ReportTask, 512> // TODO Check usage on stack was previously 256
{
private:
    uint32_t REPORT_INTERVAL_MS = 5000;
    uint32_t lastReportTime = 0;

public:
    bool run()
    {
        uint32_t currentTime = millis();

        if (currentTime - lastReportTime >= REPORT_INTERVAL_MS)
        {
            lastReportTime = currentTime;

            report.reportStatus(true);

            bool healthCheck = report.checkSystemHealth(3000, true, true, true, false, false);
            if (!healthCheck)
            {
                SerialMenu::printToSerial(SerialMenu::OutputLevel::CRITICAL, F("System health check failed!"));
            }
        }
        yield();
        return true;
    }
};
```

// BILD UPDATEN!!!

StackMonitorTask

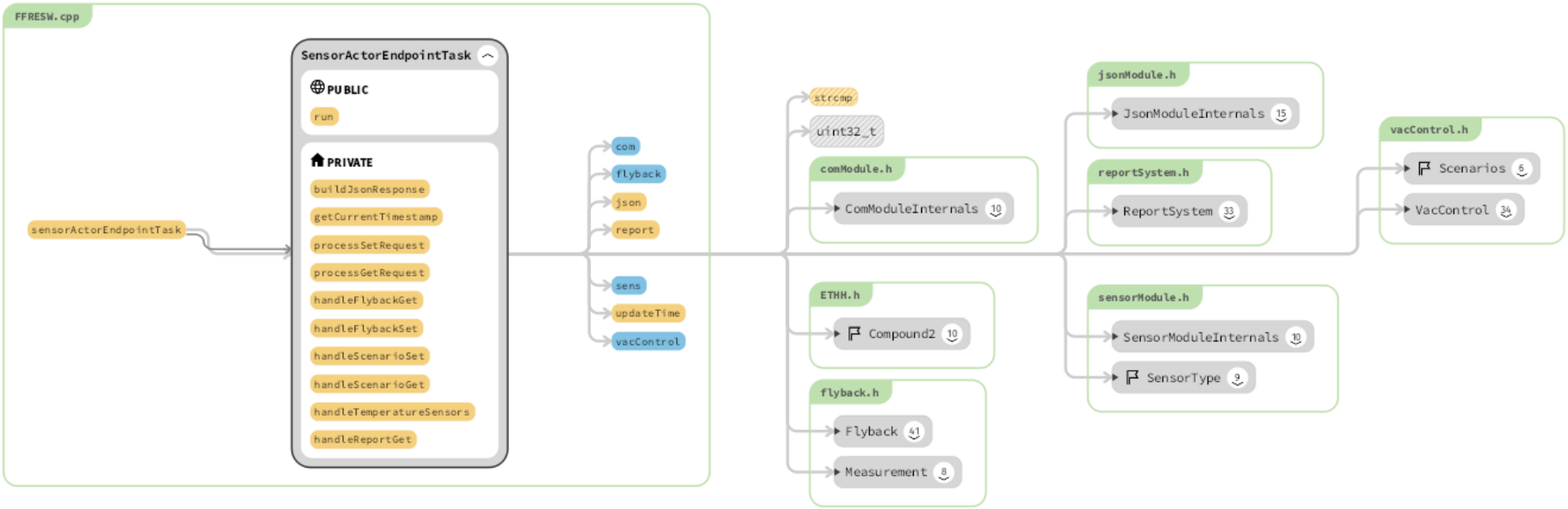
Diese Task beobachtet die Stacknutzung aller anderen Tasks. Ziel ist es, Stacküberläufe (Stackoverflow) frühzeitig zu erkennen. Wenn sich ein Stack der definierten Grenze nähert, erfolgt eine Warnung. Eine automatische Reaktion (z. B. Neustart der betroffenen Tasks), sicheres abschalten oder eine neue Priorisierung.



```cpp
/// @brief Implementation of theStackMonitorTask Class, Handles the Stacks of all running tasks. \class StackMonitorTask
class StackMonitorTask final : public frt::Task<StackMonitorTask, 256>
{
public:
    static const unsigned int REPORT_TASK_STACK_LIMIT = 512;
    static const unsigned int MONITORING_TASK_STACK_LIMIT = 128;
    static const unsigned int SENSOR_ACTOR_TASK_STACK_LIMIT = 1024;
    static const unsigned int FLYBACK_VAC_TASK_STACK_LIMIT = 512;

    static const float THRESHOLD = 0.8f;
    static const float ERR_THRESHOLD = 0.9f;

    bool run()
    {
        checkAndReport("reportTask", reportTask.getUsedStackSize(), REPORT_TASK_STACK_LIMIT);
        checkAndReport("sensorActorEndpointTask", sensorActorEndpointTask.getUsedStackSize(), SENSOR_ACTOR_TASK_STACK_LIMIT);
        checkAndReport("flyBackVacControlTask", flyBackVacControlTask.getUsedStackSize(), FLYBACK_VAC_TASK_STACK_LIMIT);

        msleep(1000);
        reportTask.post();
        return true;
    }

private:
    void checkAndReport(const String& taskName, unsigned int used, unsigned int limit)
    {
        if (used >= static_cast<unsigned int>(limit * THRESHOLD))
        {
            SerialMenu::printToSerial(SerialMenu::OutputLevel::WARNING, "Stack usage high for " + taskName + ": " + String(used) + " / " + String(limit));
        }
        else if (used >= static_cast<unsigned int>(limit * ERR_THRESHOLD))
        {
            gracefulRestart();
        }
    }
};
StackMonitorTask stackMonitorTask;
```

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.

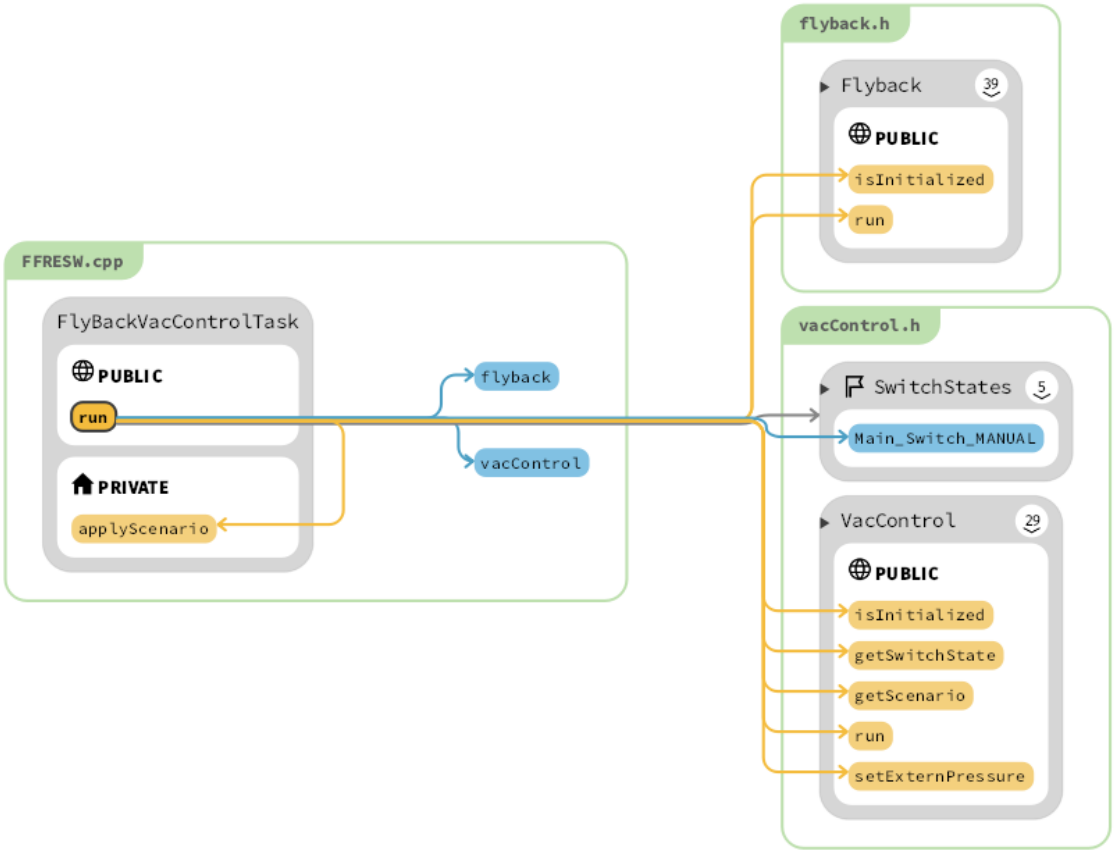


// BILD UPDATEN

TODO: NOCH BILDER EINFÜGEN ODER NICHT?

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.

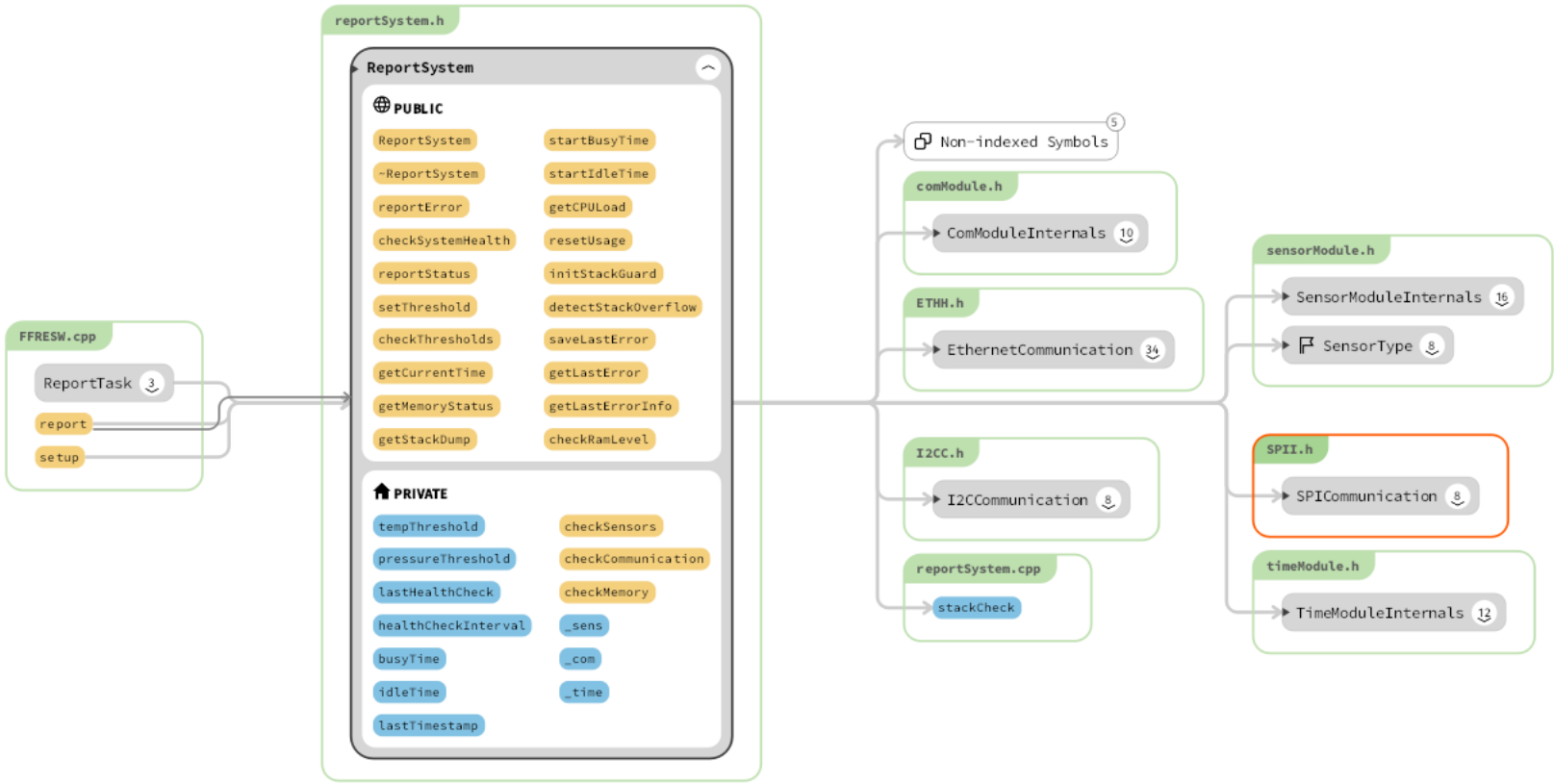


// Bilder Updaten

```cpp
/// @brief Implementation of the FlyBackTask class, control and use HV/VAC module \class FlyBackVacControlTask
class FlyBackVacControlTask final : public frt::Task<FlyBackVacControlTask, 512>
{
public:
    bool run()
    {
        if (flyback.isInitialized())
        {
            flyback.run();
            msleep(1000);
        }


        if (vacControl.isInitialized())
        {
            vacControl.run();

            vacControlModule::SwitchStates swState = vacControl.getSwitchState();
            if (swState == vacControlModule::SwitchStates::Main_Switch_MANUAL)
            {
                String response;

                {
                    ethernetMutex.lock();
                    response = com.getEthernet().getParameter(Compound2::ACTUAL_PRESSURE);
                    ethernetMutex.unlock();
                }

                float rawValue = CalcModuleInternals::extractFloatFromResponse(response, Type::Pressure);
                vacControl.setExternPressure(rawValue);

                int scenario = vacControl.getScenario();
                applyScenario(scenario);

                reportTask.post();
            }

            msleep(1000);
        }

        yield();
        return true;
    }
```

ReportSystem

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.

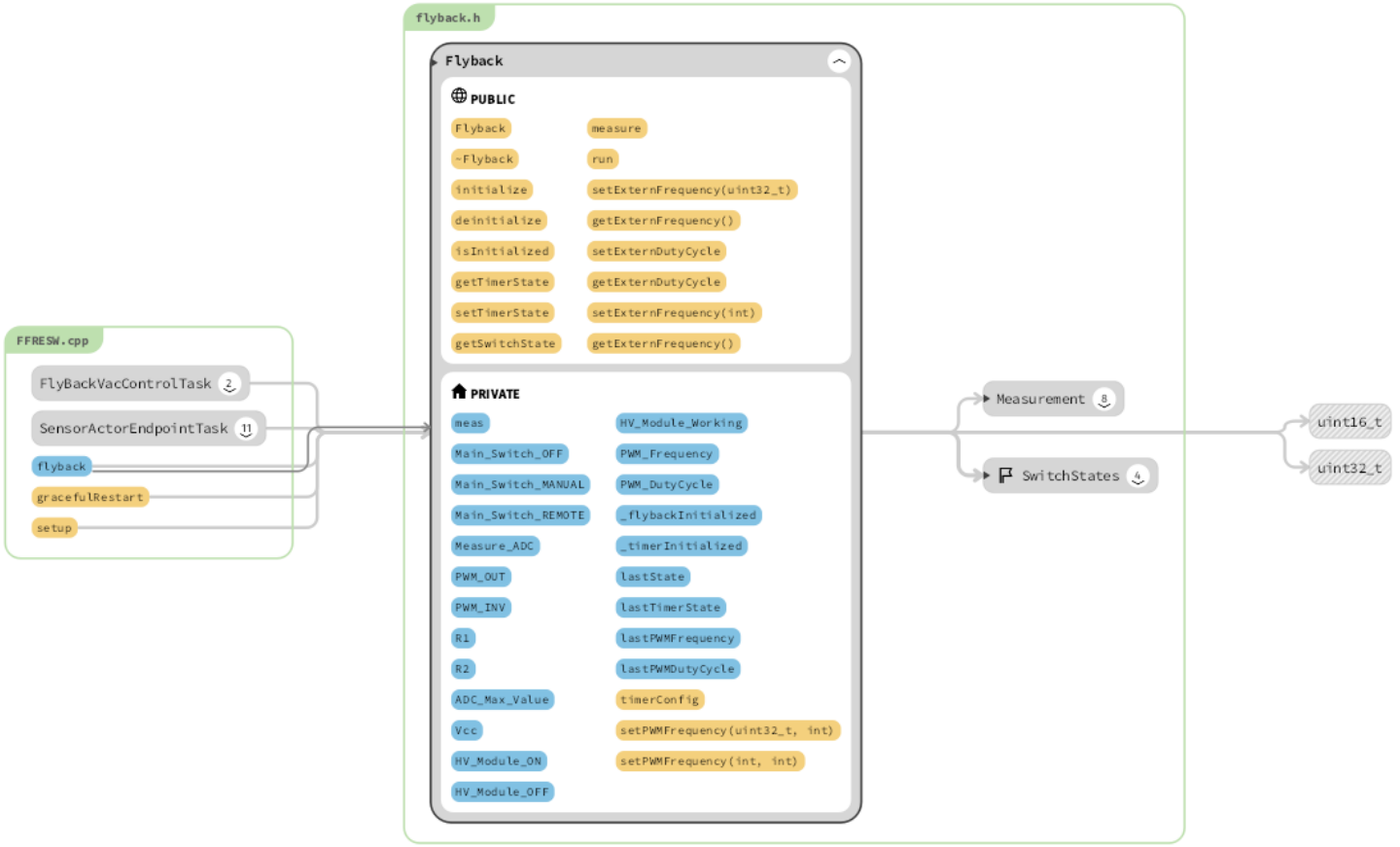
Darüber hinaus übernimmt das System eine zentrale Rolle beim **Monitoring von Ressourcen**: Funktionen wie getMemoryStatus, checkRamLevel, getStackDump oder detectStackOverflow ermöglichen eine präzise Überwachung des internen Speichers, des SRAMs und der Stack-Nutzung. Schwellenwerte können über setThreshold 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.

```cpp
/// @brief Class-Task to report system health and status periodically \class ReportTask
class ReportTask final : public frt::Task<ReportTask, 512>
{
private:
    uint32_t REPORT_INTERVAL_MS = 5000;

public:
    bool run()
    {
        if (wait(REPORT_INTERVAL_MS))
        {
            report.reportStatus(true);

            bool healthCheck = report.checkSystemHealth(3000, true, true, true, false, false);
            if (!healthCheck)
            {
                SerialMenu::printToSerial(SerialMenu::OutputLevel::CRITICAL, F("System health check failed!"));
            }
        }
        else
        {
            yield();
        }

        return true;
    }
};
ReportTask reportTask;
```

Flyback Library

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.

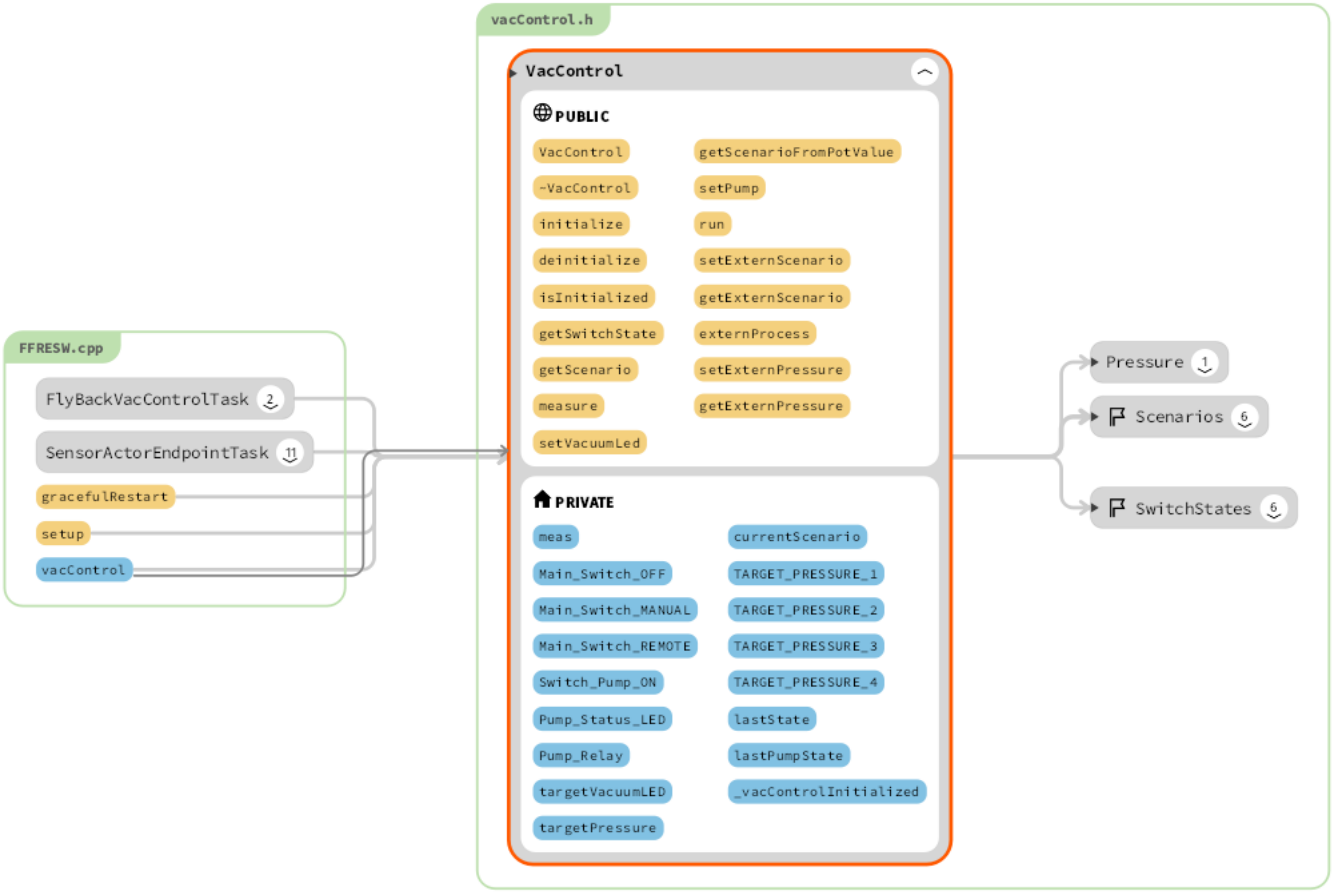
Die Integration der Flyback Library erfolgt an zwei verschiedenen Stellen innerhalb des Task-Modells der Embedded Software:

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.

vacControl Library

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**:

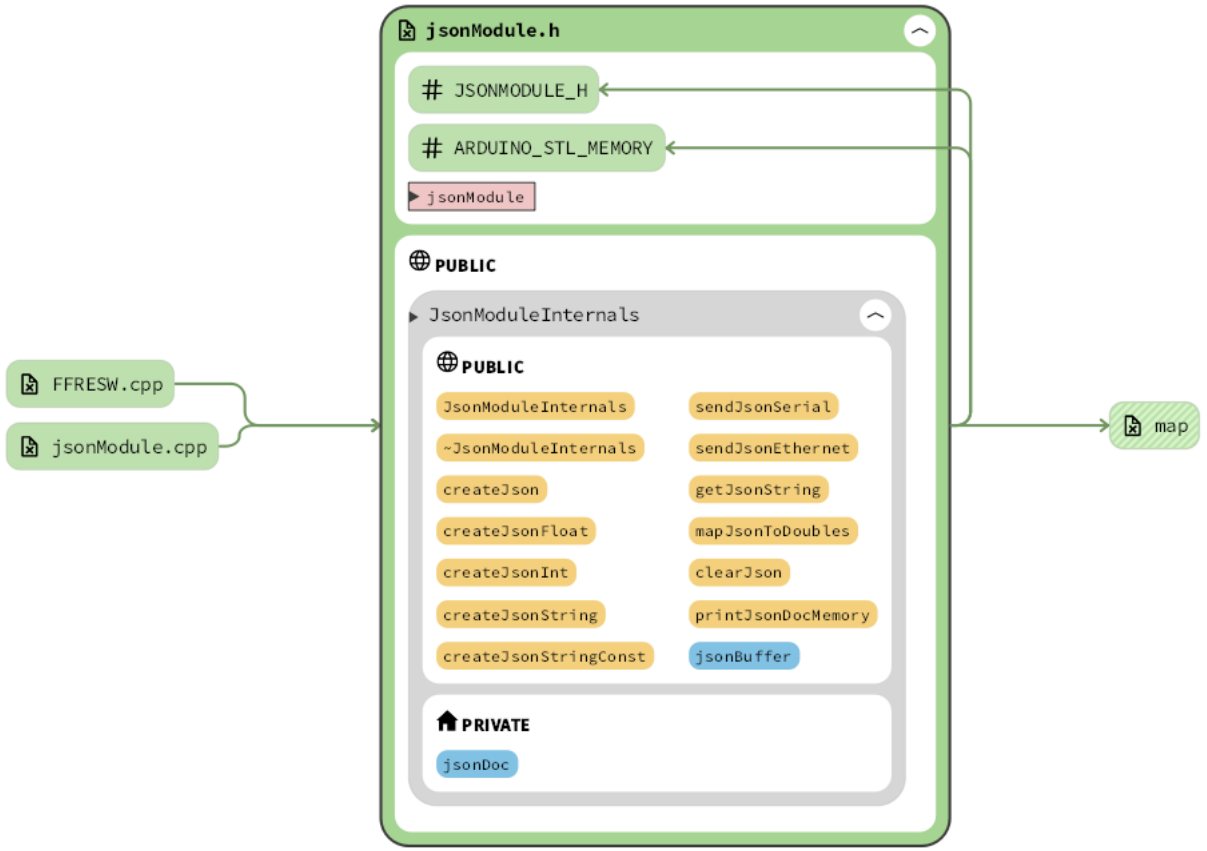
1. **SensorActorEndpointTask**: Dieser Task ermöglicht eine **direkte Steuerung über die API**. Wird ein neues Szenario oder ein Steuerbefehl von außen empfangen (z. B. über eine PC-Software oder ein HMI), reagiert vacControl entsprechend und passt die Vakuumsteuerung an die neuen Anforderungen an.
2. **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.

jsonModule

Das **jsonModule** dient als zentrale Komponente für die **Erstellung, Verwaltung und Übertragung von JSON-Daten** innerhalb unserer Embedded Software. Über die bereitgestellten Methoden der Klasse JsonModuleInternals lassen sich strukturierte JSON-Objekte dynamisch aufbauen, welche anschließend zur **Datenbereitstellung** oder zum **Datenaustausch mit dem HAS (Home Automation System)** verwendet werden. BILD ÄNDERN UPDATEN!!!!



Die Bibliothek unterstützt dabei die Erstellung unterschiedlicher Datentypen – darunter **Integer**, **Float**, **Strings** sowie **konstante Strings** – mittels Methoden wie createJsonInt, createJsonFloat, createJsonString und createJsonStringConst. Die erzeugten JSON-Dokumente können anschließend über sendJsonSerial oder sendJsonEthernet an andere Systemkomponenten oder externe Systeme gesendet werden.

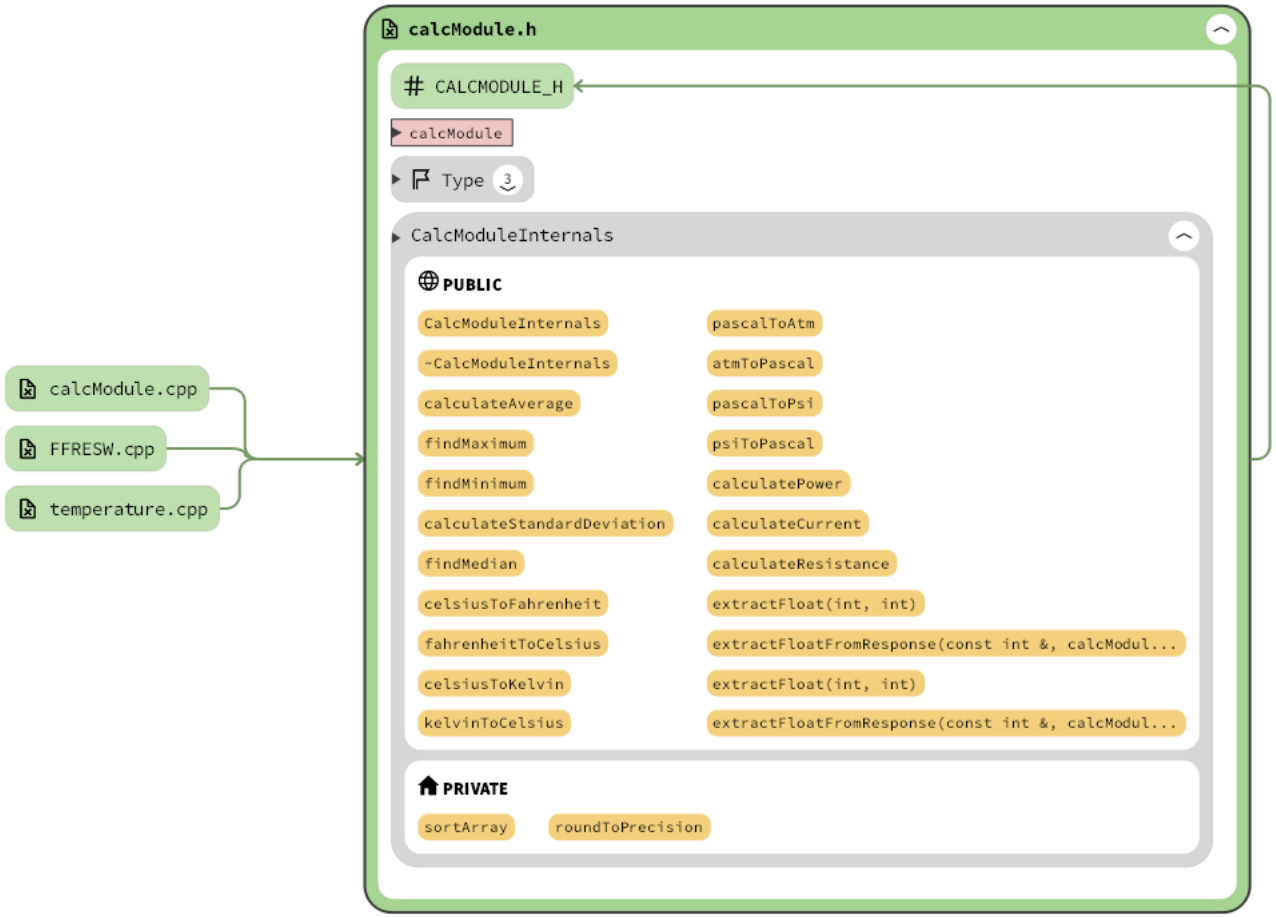
Zur Weiterverarbeitung stellt das Modul zusätzliche Funktionen bereit, etwa um JSON-Daten in Strings zu konvertieren (getJsonString), in Double-Werte zu mappen (mapJsonToDoubles) oder die Dokumente zu analysieren und zu debuggen (printJsonDocMemory). Über clearJson lassen sich JSON-Daten gezielt zurücksetzen.

Eingebunden wird das Modul in mehreren Dateien, u. a. FFRESW.cpp und jsonModule.cpp, und greift intern auf einen privaten jsonDoc zur Speicherverwaltung zurück.

Insgesamt stellt das jsonModule eine **flexible und wiederverwendbare Lösung zur strukturierten Kommunikation** innerhalb des Systems und mit externen Partnern wie dem HAS dar.

calcModule

Das **calcModule** stellt eine zentrale Sammlung mathematischer und physikalischer Berechnungsfunktionen innerhalb der Embedded Software dar. Es wird in verschiedenen Modulen eingebunden, unter anderem in FFRESW.cpp, temperature.cpp und calcModule.cpp, und ist dafür zuständig, **grundlegende Auswertungen, Konvertierungen und physikalische Umrechnungen** effizient und standardisiert durchzuführen. BILD ÄNDERN UPDATEN!!!!



Die Klasse CalcModuleInternals bietet eine Vielzahl an **öffentlichen Methoden**, die zur Verarbeitung von Messdaten, zur Umrechnung von physikalischen Einheiten sowie zur Durchführung statistischer Auswertungen genutzt werden. Dazu gehören unter anderem:

- **Statistische Funktionen** wie calculateAverage, findMaximum, findMinimum, calculateStandardDeviation und findMedian, welche zur Auswertung von Sensordaten und anderen Messreihen eingesetzt werden.
- **Physikalische Umrechnungen** wie celsiusToFahrenheit, fahrenheitToCelsius, celsiusToKelvin, kelvinToCelsius sowie Druckeinheiten-Konvertierungen (pascalToAtm, atmToPascal, pascalToPsi, psiToPascal), die für die normierte Darstellung von Messwerten sorgen.
- **Technische Berechnungen** wie calculatePower, calculateCurrent und calculateResistance, die im Kontext von elektrischen und energetischen Systemen eingesetzt werden.

Zusätzlich bietet das Modul Funktionen zur **Datenextraktion**, wie extractFloat(int, int) oder extractFloatFromResponse(...), welche es ermöglichen, gezielt Messwerte aus komplexeren Datenstrukturen zu isolieren und weiterzuverarbeiten.

Im Hintergrund unterstützen private Hilfsfunktionen wie sortArray und roundToPrecision die Genauigkeit und Effizienz der Berechnungen.

Insgesamt dient das calcModule als **universelle Recheneinheit**, die in verschiedensten Systemkomponenten eingebunden ist, um Rohdaten in verwertbare Informationen umzuwandeln und physikalische Größen konsistent zu behandeln – essenziell für die sichere und nachvollziehbare Steuerung technischer Prozesse.

EthernetCommunication-Modul

Das **EthernetCommunication-Modul** ist die zentrale Schnittstelle für die Netzwerkkommunikation über Ethernet innerhalb unserer Embedded Software. Es ermöglicht die bidirektionale Übertragung von Daten zwischen dem System und externen Komponenten oder Benutzerschnittstellen über TCP/IP-basierte Verbindungen.

Hauptfunktionen:

Die Klasse EthernetCommunication stellt eine breite Palette an **öffentlichen Methoden** bereit, die sämtliche Aspekte der Kommunikation abdecken:

- **Initialisierung & Verbindungsaufbau**:
  - beginEthernet() initialisiert die Ethernet-Schnittstelle.
  - handleEthernetClient() und getClient() kümmern sich um die Verwaltung von eingehenden Verbindungen.
  - isInitialized() stellt sicher, dass die Schnittstelle korrekt gestartet wurde.
- **Datenübertragung**:
  - sendEthernetData() und receiveEthernetData() bilden die Grundlage für den Datentransfer.
  - sendJsonResponse() erlaubt das gezielte Versenden von JSON-Antworten, z. B. als Reaktion auf Anfragen des HAS oder anderer Clients.
  - Über getSendDataFlag() und setSendDataFlag() kann der Sendezustand kontrolliert werden.
- **Endpunkt-Handling**:
  - Methoden wie getRequestedEndpoint() und getSpecificEndpoint() helfen beim Parsen eingehender Datenpakete und deren Routing an die richtigen internen Komponenten.
- **Compound-Management**:
  Eine zentrale Besonderheit ist das **Compound-System**, das strukturierte Datenpakete kapselt. Methoden wie setCompound, getCompound, getParsedCompound, setCompoundInternal und parseCompoundResponse erlauben es, diese Datenstrukturen zu erstellen, zu interpretieren und in verschiedene Module (z. B. Compound1, Compound2, Compound3) weiterzuleiten.
- **Kommando- und Parametersteuerung**:
  - Über sendCommand(), setParameter() und getParameter() kann gezielt auf Steuerbefehle und Konfigurationswerte zugegriffen werden, um z. B. Module im System remote zu konfigurieren oder Statuswerte abzufragen.

Interne Verwaltung:

Intern nutzt das Modul unter anderem folgende private Variablen:

- server und client: zur Verwaltung der Netzwerkverbindung.
- ethernetInitialized: Statusflag zur Prüfung der Initialisierung.
- sendDataFlag: Signalisiert, ob neue Daten zur Übertragung vorliegen.
- floatToIEEE754 und parseResponse: Methoden zur Konvertierung und Auswertung von Rohdaten.

Ethernet Frame

The implemented Ethernet standard used by the controller is IEEE 802.3. This frame has the following format:

1 2 3 4 5 6	7 8 9 10 11 12	13 14	15 16 17	variable (43 to 1497bytes)	n-3 n-2 n-1 n
Destination	Source	L	LLC	Data	FCS

Destination	Destination Address
Source	Source Address
L	Length
LLC	The Logical Link Control Header, described in the IEEE 802.2 Specification
Data	The Data consists of upper layer headers such as TCP/IP or IPX and then the actual user data
FCS	Frame Check Sequence

Der gezeigte Ethernet-Frame basiert auf dem IEEE 802.3-Standard und ist wie folgt aufgebaut:

1. **Destination und Source (je 6 Bytes)**: Diese Felder enthalten die MAC-Adressen des Zielgeräts (Destination) und des Absenders (Source).
2. **Length (2 Bytes)**: Gibt die Länge des nachfolgenden Datenbereichs an.
3. **LLC (Logical Link Control, variabel)**: Dieser Header nach IEEE 802.2 ermöglicht die Steuerung der Datenübertragung zwischen Geräten.
4. **Data (43 bis 1497 Bytes)**: Enthält die Nutzdaten sowie Protokoll-Header höherer Schichten (z. B. TCP/IP oder IPX).
5. **FCS (Frame Check Sequence, 4 Bytes)**: Eine Prüfsumme zur Erkennung von Übertragungsfehlern im Frame.

Der Frame ist klar strukturiert, wobei die festen Header-Felder (Destination, Source, Length) von variablen Daten und einer Fehlerprüfung (FCS) abgeschlossen werden.

Service	Command	Response
GET	p:|**service**|**parameter ID**|**index**	p:|**error**|**service**|**parameter ID**|**index**|**value**
SET	p:|**service**|**parameter ID**|**index**|**value**	p:|**error**|**service**|**parameter ID**|**index**|**value**

Part	Description	Format	Digits
service	Service code (see 6.4.4 Services)	Hex	2
parameter ID	uint32 identity of a parameter (see 6.4.5 Parameter ID)	Hex	8
error	Error code (see 6.4.6 Error Codes)	Hex	2
index	Array index. If parameter is not an array use 00	Hex	2
value	Set or response value	Dec/Hex*	variable

Die Tabelle beschreibt die **GET-** und **SET-Parameter**-Funktionen für die Kommunikation mit dem VAT-Mikrocontroller. Hier die Funktionsweise im Detail:

GET-Parameter (Abfragen eines Werts)

- **Kommando (p:)**
  - service: Hex-Code des Dienstes (2-stellig).
  - parameter ID: Hex-Code der Parameter-ID (8-stellig).
  - index: Array-Index (2-stellig, 00 falls kein Array).
- **Antwort (p:)**
  - error: Hex-Fehlercode (2-stellig).
  - service, parameter ID, index: Echo der Anfrage.
  - value: Abgerufener Wert (variable Länge).

Beispiel:

p: 01 00000042 00 → Fragt Parameter-ID 0x42 (Dienst 0x01, Index 0) ab.

Antwort: p: 00 01 00000042 00 1234 → Wert "1234" bei Erfolg (Fehlercode 0x00).

SET-Parameter (Schreiben eines Werts)

- **Kommando (p:)**
    - Wie GET, aber mit zusätzlichem value-Feld (neuer Wert).
- **Antwort (p:)**
    - Bestätigt die Änderung oder meldet einen Fehler (error). Enthält ebenfalls den geänderten Wert (value).

Beispiel:

p: 02 000000FF 01 ABCD → Setzt Parameter-ID 0xFF (Dienst 0x02, Index 1) auf "ABCD".

Antwort: p: 00 02 000000FF 01 ABCD → Erfolg (Fehlercode 0x00).

Schlüsselfelder

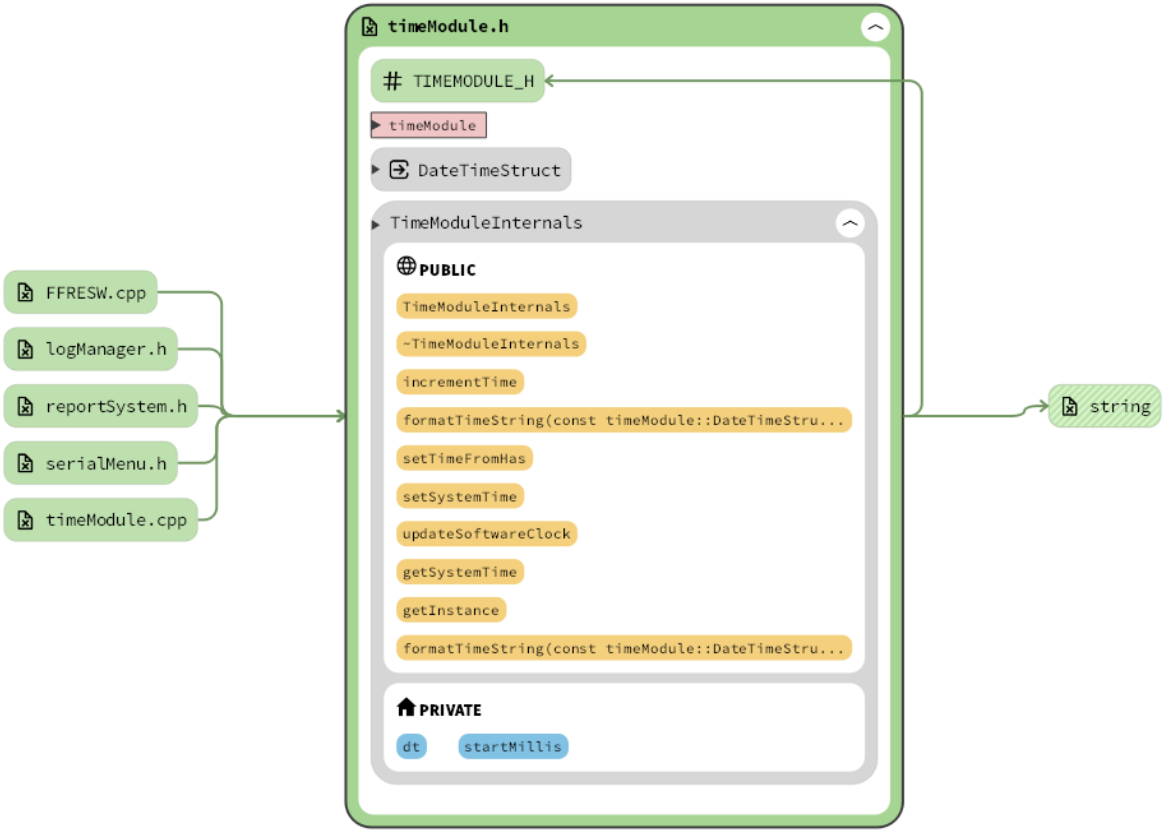
- **Fehlercodes**: 00 = Erfolg, andere Werte signalisieren Probleme (siehe Error Codes).
- **Formate**:
    - service, parameter ID, error: Hexadezimal.
    - index: Dezimal/Hex (je nach Kontext).
    - value: Variable Länge (abhängig vom Parameter).

Zweck:

- **GET** dient dem Auslesen von Konfigurationen oder Statuswerten.
- **SET** ermöglicht das Ändern von Parametern, z. B. in Steuerungssystemen.

timeModule

Das **timeModule** ist ein zentrales Modul in der Embedded-Software, das für die **Zeitverwaltung** zuständig ist. Es synchronisiert, speichert und verteilt die Systemzeit für verschiedene Anwendungsfälle wie Logging, JSON-Kommunikation und persistente Protokolle.



Kernfunktionen

1. Zeitabfrage & Synchronisierung (setTimeFromHas())

- **Request an Hardware Access Service (HAS):**
    - Das Modul sendet eine Anfrage an einen **HAS-Endpoint**, um die aktuelle **UTC-Zeit** zu erhalten.
    - Die Antwort kommt im **JSON-Format**.
- **Verarbeitung:**
    - Die empfangene Zeit wird geparst und in die interne **DateTimeStruct** geschrieben.
    - Falls nötig, wird die Zeit in die lokale Zeitzone umgerechnet.

2. Zeitverwendung in der Embedded-Software

- **Logging:**
    - Zeitstempel für **Echtzeit-Logs** werden aus dem DateTimeStruct generiert.
- **JSON-Kommunikation:**
    - Beim Versenden von JSON-Daten (z. B. Sensordaten) wird die aktuelle Zeit als timestamp eingefügt.
- **Persistente Logs (LogManager):**
    - Die Zeit wird in **dauerhaften Protokolldateien** gespeichert, z. B. für spätere Analysen.

3. Weitere Methoden

Methode	Beschreibung
getSystemTime()	Gibt die aktuelle Zeit aus dem DateTimeStruct zurück.
updateSoftwareClock()	Aktualisiert die Software-Uhr (da kein Hardware-RTC vorhanden).
formatTimeString()	Wandelt die Zeit in ein lesbares Format.
IncrementTime()	Erlaubt manuelle Zeitkorrekturen.

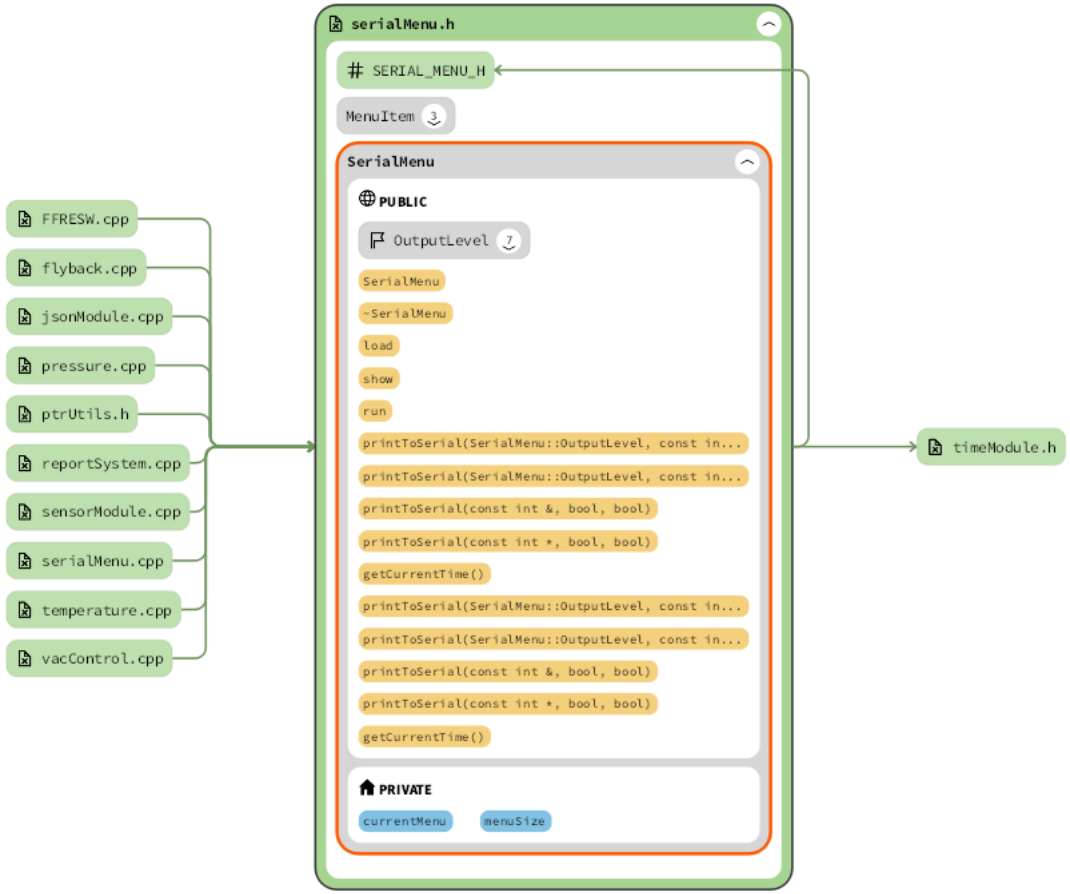
Zusammenfassend

- **Das Modul holt die Zeit vom HAS (Hardware Access Service) und verwaltet sie zentral.**
- **Die Zeit wird für Logging, JSON-Kommunikation und persistente Protokolle verwendet.**
- **Es stellt sicher, dass alle Systemkomponenten eine einheitliche Zeitreferenz haben.**

Das Modul ist essenziell für **Zeitsynchronisation, Debugging und Datenkonsistenz** in der Embedded-Software.

SerialMenu

Das **SerialMenu** ermöglicht in der Embedded-Software eine vereinfachte Handhabung des seriellen Debuggings, indem es alle Ausgaben standardisiert formatiert und zusätzlich die Möglichkeit bietet, diese über den **LogManager** persistent auf der **SD-Karte des Ethernet-Moduls** zu speichern.



```cpp
void SerialMenu::printToSerial(OutputLevel level, const String& message, bool newLine, bool logMessage)
{
    frt::Mutex serialMutex;
    serialMutex.lock();

    String output;
    if (level != OutputLevel::PLAIN)
    {
        output.reserve(message.length() + 32);
        output += '[';
        output += getCurrentTime();
        output += "] ";

        switch(level)
        {
            case OutputLevel::DEBUG:    output += "[DEBUG] "; break;
            case OutputLevel::INFO:     output += "[INFO] "; break;
            case OutputLevel::WARNING:  output += "[WARNING] "; break;
            case OutputLevel::ERROR:    output += "[ERROR] "; break;
            case OutputLevel::CRITICAL: output += "[CRITICAL] "; break;
            case OutputLevel::STATUS:   output += "[STATUS] "; break;
            case OutputLevel::PLAIN:    break; // No prefix
        }
        output += message;
    }
    else
    {
        output = message;
    }

    if (newLine)
    {
        Serial.println(output);
    }
    else
    {
        Serial.print(output);
    }

    if (logMessage)
    {
        LogManager* logger = LogManager::getInstance();

        if (logger && logger->isSDCardInitialized())
        {
            if (level == OutputLevel::WARNING ||
                level == OutputLevel::ERROR ||
                level == OutputLevel::CRITICAL)
            {
                String toLog = output;
                if (newLine) toLog += "\n";
                logger->writeToLogFile(toLog);
            }
        }
    }

    serialMutex.unlock();
}
```

rbeit

Die Methode **printToSerial** ist eine zentrale Komponente für die threadsichere Ausgabe von Nachrichten in unserem RTOS-System. Da sie von verschiedenen Tasks parallel aufgerufen werden kann, wird der Zugriff durch einen **frt::Mutex serialMutex** geschützt. Dieser Mechanismus stellt sicher, dass die seriellen Ausgaben (Serial.print) auch bei gleichzeitigen Aufrufen korrekt und ohne Race Conditions verarbeitet werden.

Der Mutex wird zu Beginn der Methode mit **lock()** aktiviert und am Ende mit **unlock()** wieder freigegeben. Diese Synchronisation ist kritisch, da die Methode systemweit für Debugging, Fehlerprotokollierung und Statusmeldungen verwendet wird. Durch den Mutex garantieren wir, dass Nachrichten vollständig und ununterbrochen ausgegeben werden – selbst bei hoher Systemlast oder parallelen Zugriffen.

Zusätzlich zur Threadsicherheit bietet die Methode flexible Steuerungsoptionen:

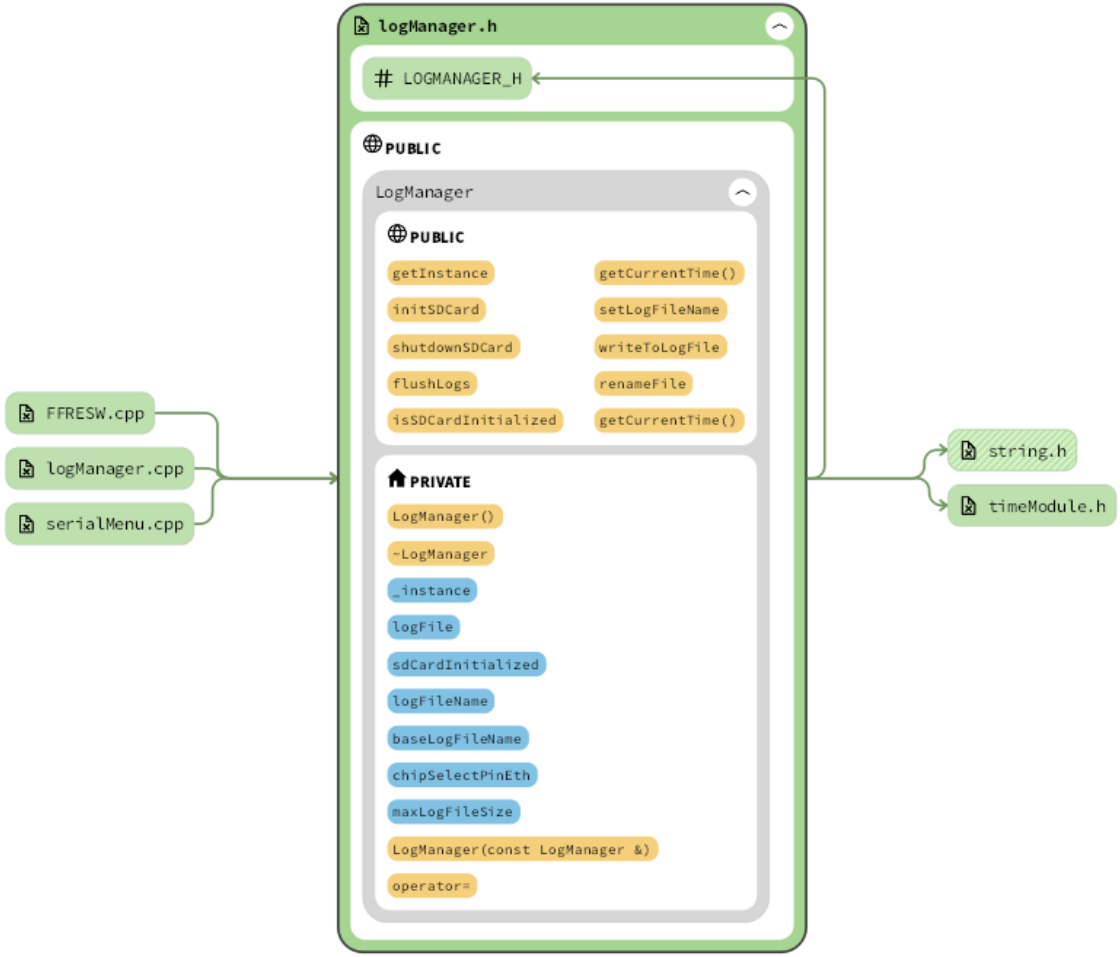
- **Log-Level** (z.B. DEBUG, ERROR) für strukturierte Ausgaben
- **Zeitstempel** für jede Nachricht
- **Optionales Logging** auf SD-Karte für kritische Meldungen

Diese Kombination aus Schutzmechanismen und Funktionalität macht printToSerial zu einer robusten Lösung für unsere Echtzeit-Anforderungen, bei denen sowohl Zuverlässigkeit als auch Performance essenziell sind.

BILD ÄNDERN UPDATEN!!!!

LogManager

Der **LogManager** ist also ein ein zentrales Singelton-Modul in der Embedded-Software, das für die persistente Protokollierung von Systemereignissen, Debug-Informationen und Benutzerinteraktionen verantwortlich ist. Er arbeitet eng mit dem SerialMenu zusammen, um eine umfassende Logging-Lösung zu bieten.



Hauptverantwortlichkeiten:

1. **Zentrale Logging-Infrastruktur**:
    - Verwaltung aller Systemlogs in einer einheitlichen Struktur
    - Bereitstellung einer Schnittstelle für andere Module (wie SerialMenu) zum Schreiben von Log-Einträgen
    - Garantie der Thread-Sicherheit durch Singleton-Implementierung
2. **Persistente Speicherung**:
    - Verwaltung der SD-Karte auf dem Ethernet-Modul (Initialisierung/Deinitialisierung)
    - Rotierende Log-Dateien mit Größenbeschränkung (maxLogFileSize)
    - Automatische Namensverwaltung (baseLogFileName, logFileName)
    - Zuverlässiges Schreiben und Flushen der Daten (writeToLogFile, flushLogs)
3. **Zeitbezogene Funktionen**:
    - Automatische Zeitstempelung aller Log-Einträge (getCurrentTime())
    - Synchronisation mit dem Systemzeitgeber
4. **Integration mit SerialMenu**:
    - Ermöglicht die persistente Speicherung aller seriellen Debug-Ausgaben
    - Standardisiert das Log-Format über alle Ausgabekanäle hinweg
    - Bietet eine einheitliche Schnittstelle für Debug- und Betriebslogs

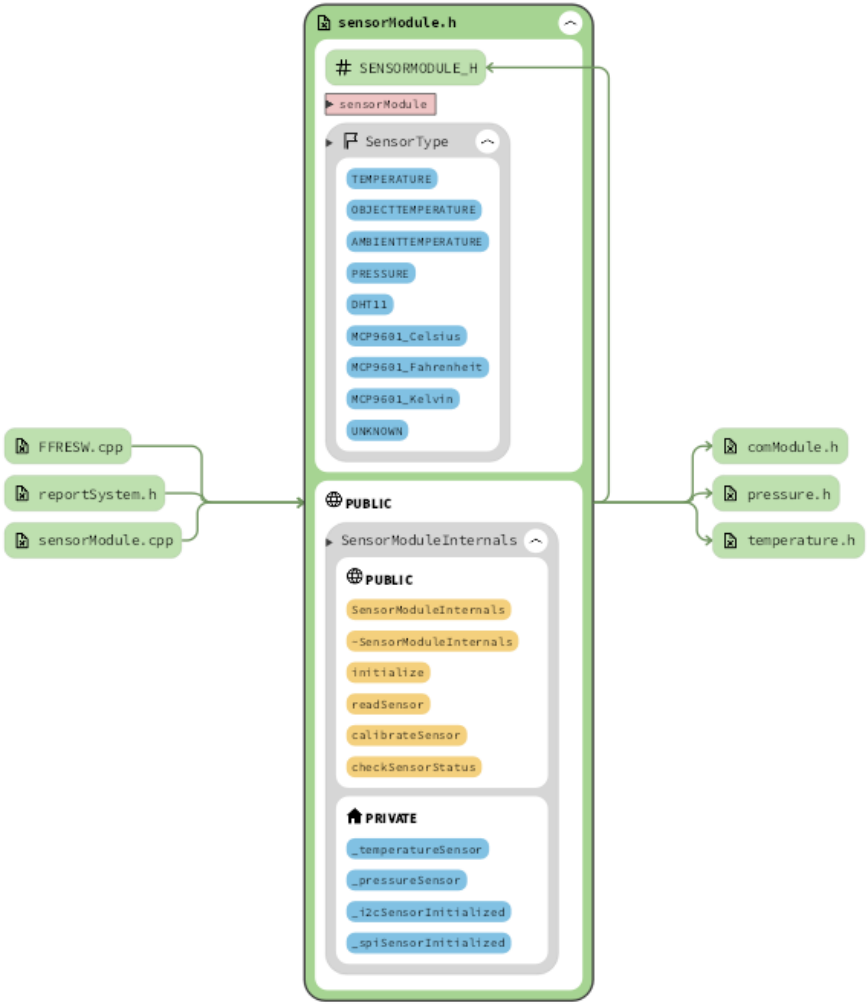
Technische Besonderheiten:

- **Hardware-Integration**:
    - Spezielle Behandlung des Chip-Select-Pins (chipselectPinEth) für parallelen Betrieb von SD-Karte und Ethernet
    - Robustes Fehlerhandling bei SD-Karten-Operationen
- **Performance-Optimierungen**:
    - Pufferung von Log-Nachrichten für effizientes Schreiben
    - Kontrolliertes Flushen der Daten zur Vermeidung von Datenverlust
- **Erweiterte Dateiverwaltung**:
    - Automatische Dateirotation bei Erreichen der Maximalgröße
    - Umbenennungsfunktionen (renamefile) für bessere Logorganisation

Der LogManager bildet damit das Rückgrat des Debugging- und Monitoring-Systems, indem er die Lücke zwischen temporären seriellen Ausgaben und dauerhafter Protokollierung schließt. Seine enge Integration mit dem SerialMenu ermöglicht Entwicklern ein nahtloses Debugging-Erlebnis, bei dem alle relevanten Informationen sowohl sofort sichtbar als auch später auswertbar sind.

SensorModule

Das **SensorModule** ist eine Kernkomponente für die Sensorintegration in Embedded-Systemen, das als abstrahierende Wrapper-Klasse verschiedene Sensortypen und Messprotokolle vereinheitlicht.



Kernfunktionalitäten

1. **Multi-Sensor Unterstützung**:
    - Integriert unterschiedliche Sensorchips:
        - MCP9601 (Temperatur in Celsius/Fahrenheit/Kelvin)
        - DHT11
    - Unterstützte Messgrößen:
        - TEMPERATURE (Generische Temperatur)
        - OBJECTTEMPERATURE (Oberflächentemperatur)
        - AMBIENTTEMPERATURE (Umgebungstemperatur)
        - PRESSURE (Druckmessung)

    Vereinheitlichte Schnittstelle:
    - Öffentliche Methoden:
        - initialize(): Sensorinitialisierung
        - readsensor(): Einheitliche Datenabfrage
        - CalibrateSensor(): Kalibrierungsroutine
        - CheckSensorstatus(): Diagnosefunktion

2. **Protokollabstraktion**:
    - Unterstützt verschiedene Kommunikationsprotokolle:
        - I2C
        - SPI
    - Kapselt chipspezifische Implementierungen

Technische Merkmale

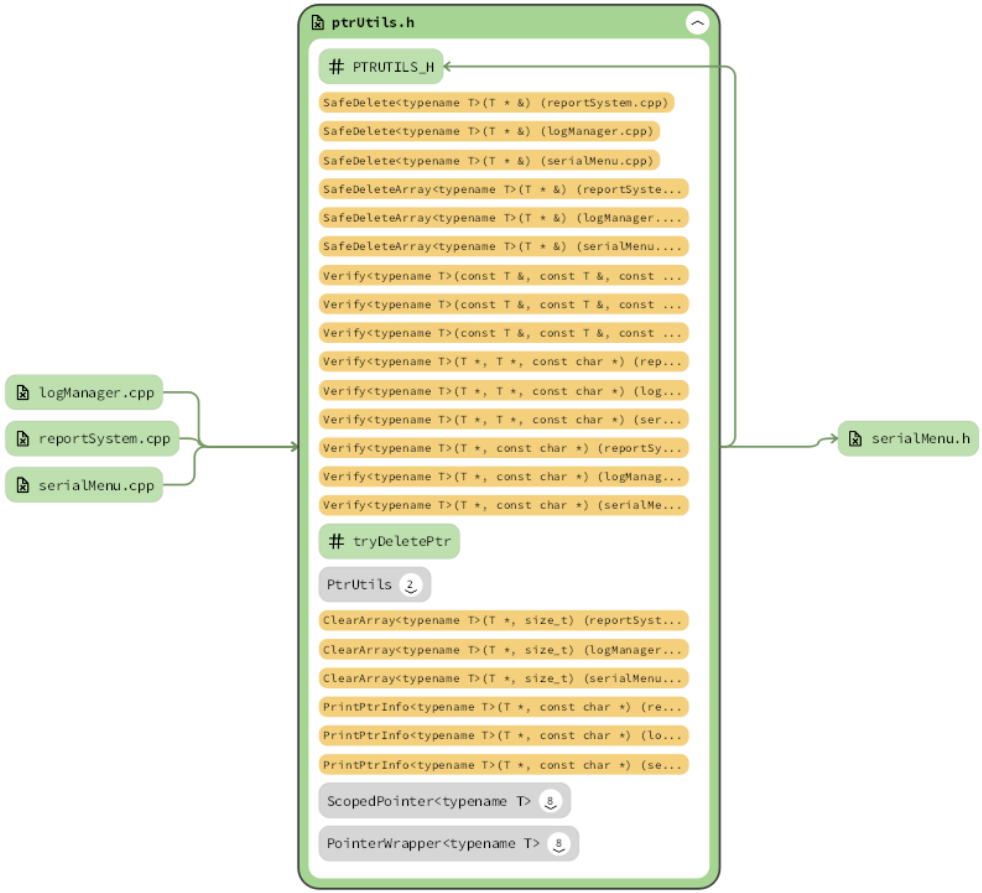
- **Initialisierungsmanagement**:
    - Zustandstracking
    - Hardware-spezifische Konfiguration
- **Chipunterstützung**:
    - MCP-Serie Temperatursensoren mit verschiedenen Ausgabeformaten
    - Erweiterbarkeit für unbekannte Sensoren (UNKNOW)
- **Diagnoseintegration**:
    - Anbindung an ReportSystem (reportSystem.h)
    - Eigenes Implementierungsmodul (sensorModule.cpp)

Das Modul abstrahiert die Komplexität der Sensorhardware und bietet eine konsistente Schnittstelle für:

- Temperaturmessungen in verschiedenen Einheiten
- Objekt- vs. Umgebungstemperatur
- Druckmessungen
- Sensorstatusüberwachung

ptrUtils

Die **ptrUtils**-Bibliothek ist eine spezialisierte Utility-Sammlung für die sichere Handhabung von Pointern in C++-Projekten, die insbesondere in Embedded- und Sicherheitskritischen Systemen eingesetzt wird.



Hauptfunktionen

1. **Sicheres Pointer-Management**:
    - SafeDelete<T>(T*&):
        - Führt eine NULL-check-gesteuerte Löschung durch
        - Setzt den Pointer automatisch auf nullptr nach dem Löschen
        - Wird in Core-Modulen (LogManager, SerialMenu, ReportSystem) verwendet
        - Verhindert doppeltes Löschen und Zugriffe auf gelöschte Objekte
2. **Pointer-Validierung**:
    - Verify<T>(const T&, const T&, ...):
        - Vergleichsfunktion für Pointer-Werte
        - Unterstützt variadische Argumente für erweiterte Checks
    - Verify<T>(T*, const char*):
        - Kombinierte NULL- und Gültigkeitsprüfung
        - Optionaler Debug-Identifier für Fehlermeldungen
3. **Erweiterte Pointer-Utilities**:
    - tryDeletePtr:
        - Exception-safe Löschmechanismus
    - ClearArray<T>(T*, size_t):
        - Sichere Array-Bereinigung mit Größenangabe
        - Wird für Pufferbereinigung in mehreren Modulen genutzt

- PrintPtrInfo<T>(T*, const char*):
    - Debugging-Hilfe mit Pointer-Informationen
    - Protokolliert Adressen und Zustände

4. **Wrapper-Klassen**:
    - ScopedPointer<T>:
        - RAII-Wrapper für automatische Speicherverwaltung
        - Ähnlich std::unique_ptr mit erweiterten Features
    - PointerWrapper<T>:
        - Generischer Pointer-Container mit Ownership-Management

Technische Besonderheiten

- **Header-only Design**:
    - Template-basierte Implementierung
    - Keine Link-Time-Abhängigkeiten
    - Einfache Integration in bestehende Projekte
- **Cross-Modul Einsatz**:
    - Wird konsistent in LogManager, SerialMenu und ReportSystem verwendet
    - Gewährleistet einheitliches Pointer-Handling systemweit
- **Debugging Support**:
    - Integrierte Diagnosefunktionen
    - Kontextsensitive Fehlermeldungen

    - Protokollierbare Pointer-Operationen
- **Sicherheitsfokus**:
    - Verhindert häufige Pointer-Fehler
    - Schutz gegen:
        - Memory Leaks
        - Dangling Pointer
        - Double Free
        - Nullpointer-Dereferenzierung

Die Bibliothek dient als fundamentale Sicherheitsschicht für die Speicherverwaltung im Projekt und reduziert dadurch systemkritische Fehlerquellen bei der manuellen Pointer-Handhabung.

Erklärung Zusammenspiel HAS, ESW und VAT

Das Gesamtsystem besteht aus drei zentralen Komponenten, die jeweils spezialisierte Aufgaben innerhalb eines vernetzten, industriellen Steuerungsumfelds übernehmen. Diese Komponenten kommunizieren über ein Ethernet-basiertes Kommunikationsnetzwerk miteinander und sind über verschiedene Schnittstellen logisch und funktional miteinander verknüpft.

1. Hardware Access Service (HAS)

Plattform: Raspberry Pi (Linux-basiert, Docker-gemanaged)

Der HAS fungiert als **zentrale Vermittlungs- und Kontrollinstanz** zwischen Benutzerschnittstelle, Datenhaltung und Embedded System. Er stellt die Verbindung zur Außenwelt (z. B. User Interface oder übergeordnete Systeme) her, empfängt Steuerbefehle und verarbeitet Rückmeldungen aus der Embedded-Hardware.

Wesentliche Merkmale:

- Läuft in einem **Docker-Container** unter Ubuntu, wodurch eine **portierbare und isolierte Systemumgebung** geschaffen wird.
- Die Containerumgebung ist über **Docker Compose** orchestriert und kann über Git versioniert verwaltet und automatisiert deployt werden.
- Die eigentliche Anwendung basiert auf dem **FFRHAS-Repository**, das als Git-Submodul eingebunden ist. Dies erlaubt **modulare Updates**, Wiederverwendbarkeit und saubere Trennung von Infrastruktur und Anwendungscode.
- **JSON-basierte Kommunikation** über TCP/IP-Sockets zur Embedded-Komponente (eSW).
- Speicherung aller ein- und ausgehenden Daten sowie Zustandsinformationen in einer **MongoDB**. Diese dokumentenorientierte Datenbank ermöglicht eine **flexible Speicherung strukturierter Daten** und eignet sich ideal für dynamische Systemzustände.

2. Embedded Software (eSW)

Plattform: Arduino Mega 2560 mit Ethernet Shield (W5100)

Die Embedded-Software auf dem Arduino ist ein **kontextsensitiver Protokollübersetzer und Echtzeit-Controller**. Sie nimmt strukturierte JSON-Kommandos vom HAS entgegen, konvertiert sie in das proprietäre Steuerprotokoll des VAT-Geräts und leitet sie über eine serielle Schnittstelle weiter. Die Antwortdaten durchlaufen denselben Weg in umgekehrter Richtung.

Kernfunktionen:

- Übersetzung von Nachrichten zwischen dem **standardisierten JSON-Format** (vom HAS) und dem **spezifischen Format des VAT-Mikrocontrollers**.
- Kommunikation über **TCP/IP via Ethernet Shield**, wobei das Shield über **SPI** angebunden ist.
- Verwendung von **FreeRTOS** zur Umsetzung eines **Multitasking-fähigen Firmware-Designs**:
    - Tasks für Kommunikation, Sensorabfrage, Datenpufferung und Watchdog-Funktionen
    - Objektorientierte Programmierung durch Einsatz der Bibliothek frt.h:
        - Bessere Kapselung, Wiederverwendbarkeit und Erweiterbarkeit
        - Nutzung von Queues, Timer-Callbacks und Prioritätsmanagement

Weitere Schnittstellen:

- **SPI**:
    - Für das Ethernet-Shield (W5100)
    - Optional: Zugriff auf MicroSD zur Datensicherung
- **I^2C/SPI**:
    - Sensorintegration: z. B. MCP9601 (Thermoelemente)
    - Ermöglicht die Erfassung zusätzlicher Umgebungsparameter in Echtzeit

3. VAT-Steuergerät

Plattform: proprietärer Mikrocontroller der Firma VAT (z. B. STM32-basiert)

Das VAT-Gerät steuert über eine industrielle Schnittstelle **präzise Ventile, Druckregelventile oder Vakuumkomponenten** und ist in vielen industriellen Anwendungen für **Hochpräzisionsregelung** im Einsatz.

Kommunikation und Protokoll:

- Die Kommunikation mit externen Komponenten erfolgt über **EtherCAT** – ein industrielles **Echtzeit-Ethernet-Protokoll**, das extrem geringe Latenzen, deterministisches Timing und Synchronisierung mehrerer Slave-Geräte erlaubt.
- Der Mikrocontroller versteht kein JSON – daher ist ein **Middleware-Ansatz durch die eSW notwendig**, um zwischen High-Level-Kommandos (z. B. „Setze Ventil 3 auf 65 %“) und dem Low-Level-Datenprotokoll der VAT-Schnittstelle zu übersetzen.

Kommunikationsarchitektur: Datenfluss und Zuständigkeiten

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

Toolchain

Für unser Embedded-Software-Projekt war von Anfang an klar: Die Möglichkeiten der offiziellen **Arduino IDE** – sowohl in ihrer klassischen als auch in der neueren Version (Arduino Pro IDE, später wieder eingestellt) – reichen nicht aus. Zwar ist die Arduino IDE ideal für kleinere Projekte und schnelle Prototypen, doch sie stößt bei wachsender Komplexität an technische und strukturelle Grenzen.

Warum eine eigene Toolchain notwendig war

Herausforderungen mit der Arduino IDE:

- **Mangelhafte Projektstrukturierung:** Große Codebasen mit mehreren Modulen oder Bibliotheken lassen sich kaum übersichtlich verwalten.
- **Fehlende Build-Konfiguration:** Keine feingranulare Kontrolle über Compiler-Flags, Build-Skripte oder Linker-Einstellungen.
- **Kein integriertes Debugging:** Debugging beschränkt sich meist auf Serial.println() – keine native Unterstützung für Breakpoints, Watchpoints oder Live-Inspektion.
- **Schwache Integration von Versionskontrolle:** Nur rudimentäre Unterstützung für Git oder andere VCS-Systeme.
- **Nicht geeignet für Continuous Integration/Deployment (CI/CD):** Automatisiertes Testen und Bauen ist ohne zusätzliche Skripte kaum möglich.

Daher fiel die Entscheidung: **Eine eigene, erweiterbare Toolchain muss her**, um unseren professionellen Anforderungen gerecht zu werden – vergleichbar mit klassischen Softwareprojekten.

IDE-Auswahl

1. Programino (kommerziell) – getestet, aber verworfen

Programino ist eine auf Arduino zugeschnittene kommerzielle IDE mit erweiterten Features wie Codevervollständigung, Syntaxhervorhebung und Tools zur Codeanalyse. Jedoch ergaben sich in der Praxis folgende Probleme:

- **Instabilität:** In Tests kam es zu mehreren Abstürzen und Fehlverhalten bei Projektimporten.
- **Eingeschränkter Support:** Die Reaktionszeiten auf Supportanfragen waren lang; Community-Support war kaum vorhanden.
- **Lizenzmodell:** Die Lizenz war kostenpflichtig und nicht flexibel genug für teamweite Nutzung.

Trotz guter Ansätze keine tragfähige Lösung für ein professionelles Setup.

https://programino.com

2. Sloeber – die überzeugende Alternative

Die Wahl fiel schließlich auf **Sloeber**, eine auf **Eclipse CDT** (C/C++ Development Tooling) basierende Open-Source-IDE speziell für Arduino. Vorteile:

- **Vollständige C/C++-Toolchain:** Projekte sind als klassische C++-Projekte organisiert, mit Header-Verwaltung, Pfadkonfiguration und Makefile-Support.
- **Modularität:** Einbindung von Bibliotheken, mehreren Targets, Custom Board-Definitionen etc. problemlos möglich.
- **Konfigurierbare Build-Einstellungen:** Compiler- und Linker-Flags lassen sich gezielt anpassen.
- **Gute Integration von externen Tools:** z. B. avrdude, avr-size, make, oder CI-Skripte.
- **Debugging-Support:** In Kombination mit Debuggern (z. B. Atmel ICE, J-Link) ist echtes Source-Level-Debugging möglich.
- **Kompatibilität:** Volle Unterstützung der Arduino-Kernel und Bibliotheken durch Integration der offiziellen Arduino-Core-Dateien.

https://eclipse.baeyens.it

Fazit: Trotz kleiner Eigenheiten der Eclipse-Umgebung bietet Sloeber eine robuste Plattform für strukturierte, skalierbare Embedded-Projekte mit Arduino-Technologie.

Compiler und Build-System

Compiler: avr-gcc

Verwendet wurde **avr-gcc in Version 7.3.0-atmel3.6.1-arduino7**, ein speziell für Atmel/AVR-Mikrocontroller optimierter GCC-Fork. Er ist das Standardwerkzeug für die Arduino-AVR-Plattform. Vorteile:

- **Standardkonformität:** Unterstützt C89, C99, C++11 und mehr.
- **Optimierungsoptionen:** Flags wie -Os, -O2, -ffunction-sections, -fdata-sections verbessern Codegröße und Performance.
- **Multiplattform:** Ermöglicht Cross-Kompilierung für alle gängigen AVR-Chips.

Erzeugte Datei: .elf

Statt nur eine .hex-Datei zu betrachten, wird hier bewusst mit der **ELF-Datei** (Executable and Linkable Format) gearbeitet. Vorteile:

- Enthält Debug-Symbole (für Source-Debugging)
- Symbolische Namen von Funktionen und Variablen
- Speichersegmente (.text, .data, .bss) klar abgegrenzt

Diese Datei ist zentrale Basis für weitere statische Analysen.

Analyse- und Diagnosetools

1. avr-objdump

- Disassemblierung der ELF-Datei
- Anzeige des erzeugten Assemblercodes
- Kontrolle, ob Inlining und Optimierungen korrekt angewendet wurden
- Rückverfolgbarkeit der generierten Binärdaten zu Funktionen

2. avr-nm

- Listet alle Symbole mit Adresse, Größe und Typ (Funktion, Variable, Konstante)
- Analyse von RAM- und Flashbelegung
- Ermittlung von Speichernutzern (z. B. große globale Arrays)

3. awk-Skripte zur Automatisierung

- Verarbeitung der Ausgaben von avr-size, avr-nm u. a.
- Beispiel: size.awk analysiert die Speicherverteilung (.text, .data, .bss)
- Ermöglicht automatische Auswertung über Zeit (z. B. Vergleich vor/nach Optimierungen)
- Hilfreich für Trends wie: *„Wie wächst der Speicherverbrauch im Projekt?“*

Professionelle Toolchain mit Arduino-Technologie

Durch die Kombination aus:

- einer flexiblen und mächtigen IDE (Sloeber),
- einem standardkonformen Compiler (avr-gcc),
- tiefgreifenden Analysewerkzeugen (objdump, nm, awk),
- und optionalen Debugger-Tools

… entstand eine **hochgradig kontrollierbare Entwicklungsumgebung**, die deutlich über das hinausgeht, was mit der Arduino-IDE möglich ist.

Diese Toolchain erlaubt:

- **Reproduzierbare Builds**
- **Feingranulare Optimierungen**
- **Transparente Speicheranalysen**
- **Langfristige Skalierbarkeit des Codes**

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

.ino-Dateien sind spezielle Quellcodedateien, die im Arduino-Ökosystem verwendet werden. Sie stellen den zentralen Bestandteil eines sogenannten „Sketches“ dar – so wird ein Arduino-Programm genannt. Diese Dateiendung leitet sich vom ursprünglichen Namen der Arduino-Software ab: *"Arduino Integrated Development Environment"*, intern früher auch als *Wiring* bezeichnet (daher auch die alte .pde-Endung, die bei „Processing“ verwendet wurde).

Die **.ino-Datei** enthält typischerweise den eigentlichen Anwendungscode, den der Nutzer schreibt, einschließlich der zwei wichtigsten Funktionen:

- setup(): Wird einmal beim Start des Programms ausgeführt.
- loop(): Wird anschließend dauerhaft wiederholt und bildet die Hauptprogrammschleife.

Diese einfache Struktur abstrahiert viele komplexe Aspekte der eingebetteten Programmierung und macht die Plattform besonders für Einsteiger zugänglich.

Warum verwenden eine .ino-Datei als unser „main.cpp“ (statt z. B. .cpp oder .c)?

Der Hauptgrund liegt in der Benutzerfreundlichkeit. Die Arduino-IDE wurde entwickelt, um den Einstieg in die Mikrocontrollerprogrammierung zu erleichtern. Deshalb bietet sie eine vereinfachte Umgebung, bei der viele technische Details (z. B. das Schreiben von Funktionsprototypen, manuelles Einbinden von Headern usw.) automatisch erledigt werden. Das Dateiformat .ino signalisiert der IDE: „Dieser Code muss vor der Kompilierung speziell behandelt werden.“

Build-Prozess eines Arduino-Sketches im Detail

1. Vorverarbeitung des Sketches

Bevor der eigentliche Kompilierungsvorgang beginnt, nimmt die Arduino-IDE mehrere automatische Anpassungen vor:

- **Zusammenführung aller .ino/.pde-Dateien:** Falls der Sketch-Ordner mehrere Dateien enthält, werden sie zu einer einzigen Datei zusammengefügt – in einer festen Reihenfolge.
- **Umwandlung in eine .cpp-Datei:** Da der Compiler C++ erwartet, wird der Sketch als C++-Quelldatei weiterverarbeitet.

- **Einfügen von #include <Arduino.h>:** Dadurch werden grundlegende Arduino-Funktionen wie digitalWrite(), delay() etc. verfügbar gemacht.
- **Generierung von Funktionsprototypen:** Die IDE analysiert alle Funktionen und erstellt automatisch die erforderlichen Vorab-Deklarationen, damit Funktionen auch aufgerufen werden können, bevor sie definiert sind.
- **Einfügen von #line-Direktiven:** Diese helfen dem Compiler, bei Fehlermeldungen auf die korrekten Zeilennummern in der Original-.ino-Datei zu verweisen.

Hinweis: Dateien mit anderen Endungen wie .c, .cpp, .h werden nicht verändert, sondern wie im klassischen C/C++-Workflow behandelt.

2. Auflösung von Abhängigkeiten (Bibliotheken)

Der Compiler sucht nach allen Bibliotheken, die im Sketch via #include eingebunden sind. Dabei folgt die IDE einer bestimmten Suchreihenfolge:

- Zuerst im Board-spezifischen Core (z. B. für AVR, SAMD, ESP32)
- Dann in globalen Bibliotheksordnern
- Schließlich in benutzerspezifischen oder projektbezogenen Pfaden

Gibt es mehrere gleichnamige Bibliotheken, entscheidet ein Prioritätensystem basierend auf Kriterien wie Plattformkompatibilität und Pfad.

3. Kompatibilitätsprüfung von Bibliotheken

Ob eine Bibliothek mit dem aktuellen Board kompatibel ist, erkennt die IDE anhand der Datei library.properties:

- architectures=avr → nur für AVR geeignet
- architectures=* oder nicht gesetzt → für alle Boards geeignet (aber nicht unbedingt optimiert)

Die Angabe hilft der IDE, nur passende Bibliotheken zu laden und auf inkompatible Pakete zu verzichten.

4. Kompilierung

Die übersetzte .cpp-Datei wird mithilfe des Compilers avr-gcc (bei AVR-Boards) in mehreren Schritten verarbeitet:

- **Compilieren**: Quelltexte werden in Objektdateien (.o) übersetzt.
- **Linken**: Alle Objektdateien und Bibliotheken werden zu einer ausführbaren Datei zusammengefügt.
- **Optimieren**: Unnötige Teile werden verworfen (sog. Dead Code Elimination), um Speicher zu sparen.
- **Generierung der .hex-Datei**: Diese Datei enthält den finalen Maschinen-Code, der auf das Mikrocontroller-Board geladen wird.

Die IDE verwendet einen temporären Build-Ordner, der nach dem Upload gelöscht wird – außer bei aktivierter Ausgabedetailstufe.

5. Hochladen auf das Board

Der Upload erfolgt per USB oder seriell – je nach Boardtyp – über ein Programm wie **avrdude** (für AVR-Boards):

- Das Tool überträgt die .hex-Datei auf das Board.
- Dazu wird entweder der Bootloader des Mikrocontrollers verwendet oder ein externer Programmer.
- Bei aktivierter „verbose“-Ausgabe zeigt die IDE den genauen Befehl, die Baudrate und den Uploadstatus an.

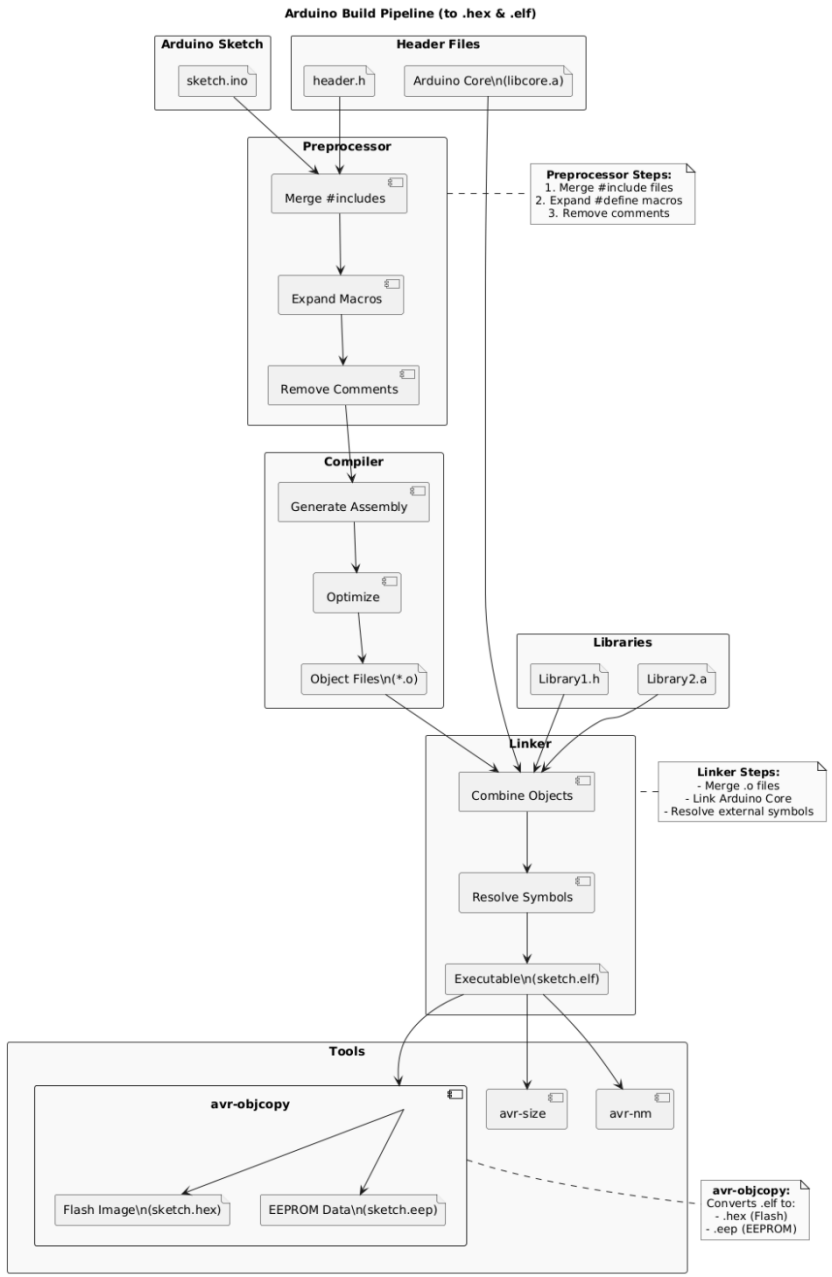
Fazit

Die Verwendung von .ino-Dateien ist ein Kernelement des Arduino-Ökosystems. Sie erleichtert den Zugang zur Mikrocontrollerprogrammierung durch automatische Vorverarbeitung, vereinfachte Syntax und integrierte Unterstützung für das Kompilieren und Hochladen. Für Fortgeschrittene erlaubt die Arduino-IDE dennoch die Einbindung klassischer .cpp, .c und .h-Dateien und kann auch komplexe Projekte mit mehreren Modulen handhaben.

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

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

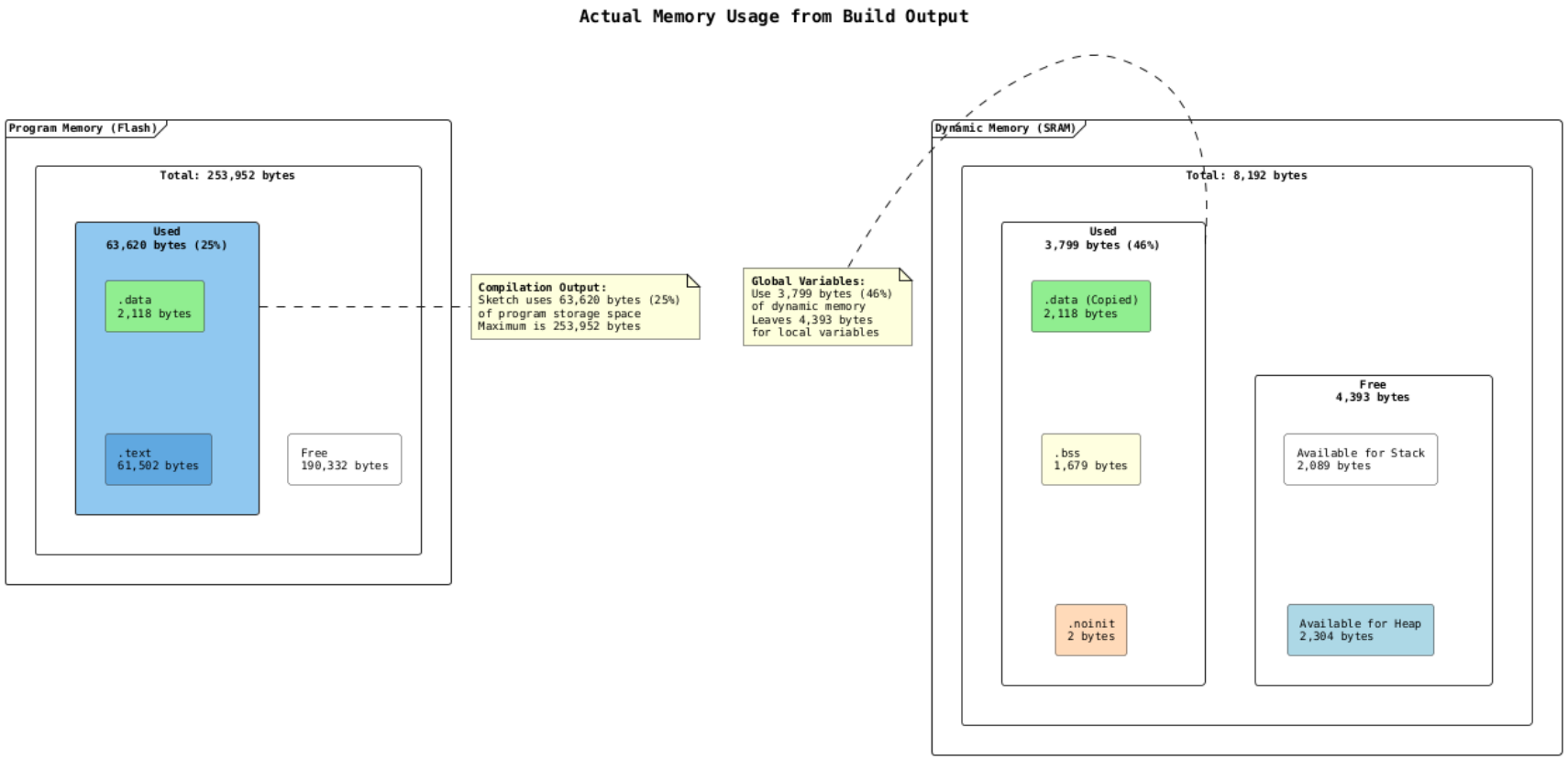
QUELLE-TOOL: http://www.plantuml.com/plantuml/



Arduino Build-Prozess (Von .ino zu .hex/.elf)

1. **Preprocessing**
   - Der Sketch (sketch.ino) wird mit allen #include-Headern (*.h) zusammengeführt.
   - Makros (#define) werden expandiert, Kommentare entfernt.
2. **Kompilierung**
   - Der preprozessierte Code wird vom **AVR-GCC-Compiler** (avr-g++) in Assembler und dann in **Objektdateien** (*.o) übersetzt.
   - Optimierungen werden angewendet (Größe/Geschwindigkeit).
3. **Linking**
   - Objektdateien werden mit der **Arduino-Core-Bibliothek** (libcore.a) und anderen Libraries verlinkt.
   - Der Linker (avr-gcc) erzeugt eine **.elf-Datei** (enthält Debug-Symbole und Programmcode).
4. **Toolchain**
   - **avr-objcopy** extrahiert aus der .elf-Datei:
     - **.hex**: Maschinencode für den Flash-Speicher (Hauptprogramm).
     - **.eep**: EEPROM-Daten (falls im Sketch verwendet).
   - **avr-size** analysiert Speichernutzung, **avr-nm** listet Symbole (Debugging).

Memory Layout und Usage vom Build Output

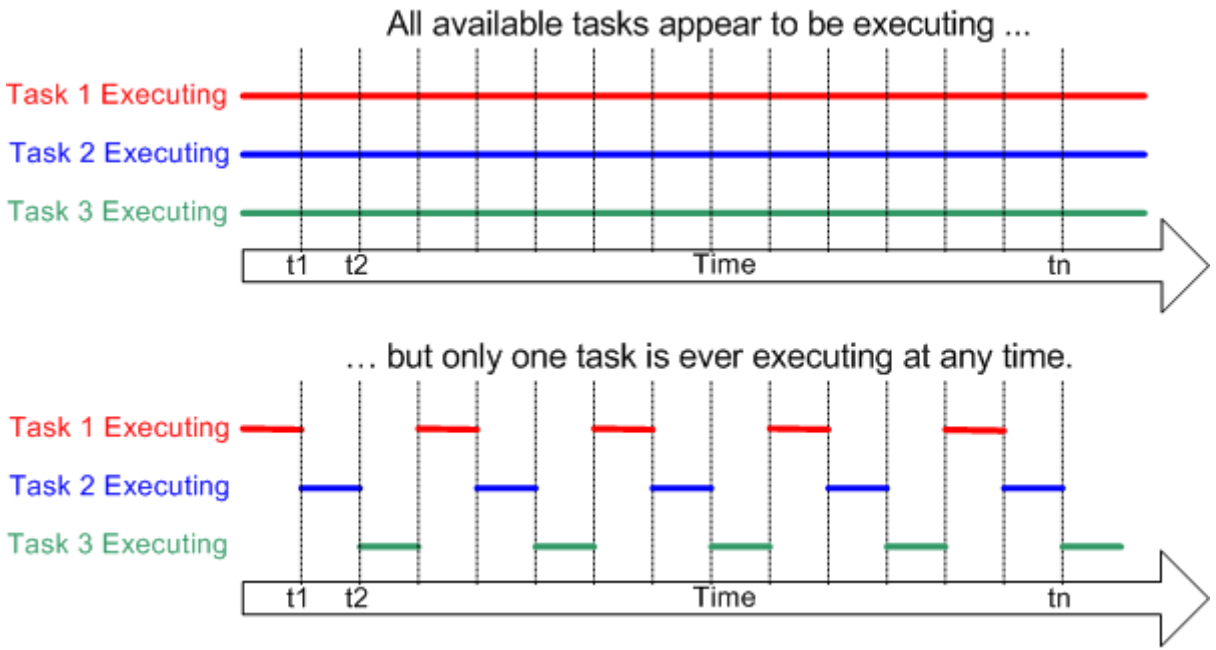


// WENN SICH SPEICHER ÄNDER VIELLEICHT NOCHMALS UPDATEN!

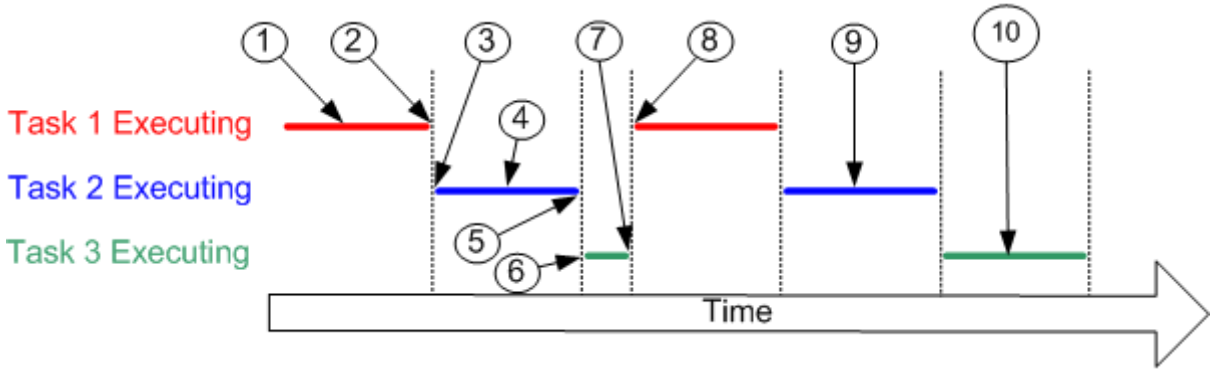
Was ist Arduino_FreeRTOS, frt.h und warum benutzten wir diese Libraries?

RTOS (Real-Time Operating System) ist ein kleines Betriebssystem für Mikrocontroller, das mehrere Aufgaben scheinbar gleichzeitig ablaufen lassen kann. Es verwaltet sogenannte *Tasks* und sorgt dafür, dass diese je nach Priorität und Zustand (laufend, wartend, blockiert) automatisch vom Scheduler abgewechselt werden. Das ermöglicht z. B.:

- Reaktion auf Sensoren und Eingaben in Echtzeit
- Gleichzeitiges Ausführen mehrerer Logiken (in unserem Fall Kommunikation, Sensordaten, Aktoren uvm.)
- Strukturierte, wartbare Programmierung durch Trennung in unabhängige Tasks



- **Darstellung**: Drei Tasks (Task 1, 2, 3) werden nacheinander ausgeführt
- **Charakteristika**:
  - Jeder Task läuft bis zur vollständigen Beendigung
  - Keine automatische Unterbrechung durch den Scheduler
  - Tasks müssen explizit "suspend" oder "yield" aufrufen, um CPU abzugeben
- **Typische Anwendung**:
  - Einfache Systeme ohne harte Echtzeitanforderungen
  - Bei Tasks mit kurzer, vorhersagbarer Ausführungszeit



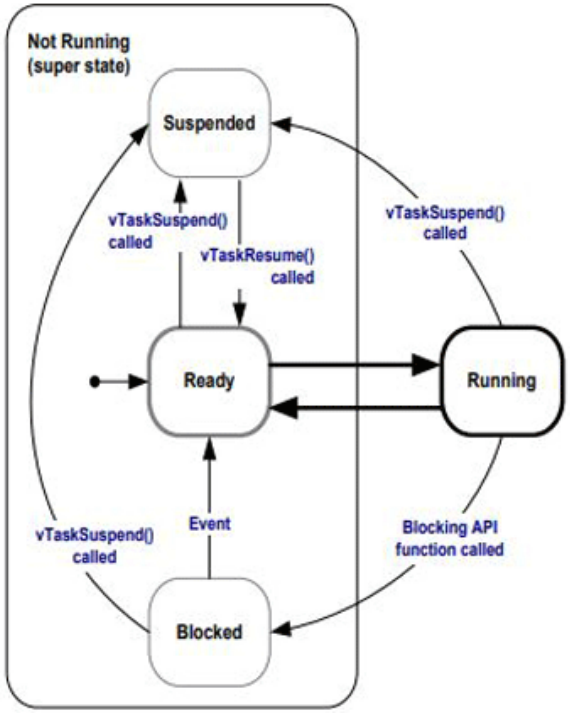
- **Scheinbare Parallelität**: Alle Tasks erscheinen gleichzeitig aktiv
- **Tatsächliche Ausführung**:
  - Nur ein Task läuft zu jedem Zeitpunkt (Single-Core)

QUELLE: https://wiki.seeedstudio.com/Software-FreeRTOS/

Arduino_FreeRTOS ist ein speziell angepasster Fork von Richard Barry's FreeRTOS für die 8-Bit ATmega-Mikrocontroller, wie sie in klassischen Arduino-Boards verwendet werden (z. B. Uno, Mega). Es ist vollständig in die Arduino-IDE integrierbar und erlaubt RTOS-Programmierung mit minimalem Konfigurationsaufwand. Merkmale:

- **Minimale Hardwareeingriffe:** Nutzt vorhandene Arduino-Timer und -Funktionen weitgehend unverändert
- **Volle IDE-Kompatibilität:** delay(), millis(), micros() bleiben funktionsfähig
- **Einfacher Einstieg:** Arduino-Sketches lassen sich durch Einbinden von Arduino_FreeRTOS.h direkt erweitern
- **Flexible Konfiguration:** Über FreeRTOSConfig.h können Optionen angepasst werden (z. B. Stack-Größe, Timer-Quellen)
- **Fehlersignalisierung über LED-Blinkcodes:** z. B. langsames Blinken bei Stacküberlauf

Der Fokus liegt auf Stabilität, nicht auf neueren Features wie Multiprocessing – deshalb wird Version 11.1.0 als LTS-Version gepflegt.



Diese Grafik zeigt die verschiedenen Zustände, die ein Task im FreeRTOS auf einem Arduino durchlaufen kann:

Zustandsübergänge:

1. **Not Running (Super State)**
   - Oberkategorie für alle inaktiven Zustände
2. **Suspended**
   - Task ist manuell angehalten (vTaskSuspend())
   - Wird durch vTaskResume() wieder aktiviert
   - Verbraucht keine CPU-Zeit
3. **Ready**
   - Task ist ausführbereit, wartet auf Zuteilung der CPU
   - Wird vom Scheduler zum Running-Zustand befördert
4. **Running**
   - Task ist aktuell auf der CPU aktiv
   - Einziger Zustand, wo Code tatsächlich ausgeführt wird
5. **Blocked**
   - Task wartet auf ein Ereignis (z.B. Delay, Semaphore)
   - Automatischer Übergang bei Aufruf blockierender Funktionen
   - Geht zurück zu Ready bei Ereigniseintritt

QUELLE(Bild above): https://circuitdigest.com/microcontroller-projects/arduino-freertos-tutorial1-creating-freertos-task-to-blink-led-in-arduino-uno

frt.h – Flössie's ready threading

frt.h ist eine moderne C++-Bibliothek, die eine objektorientierte Abstraktion über FreeRTOS bietet. Sie wurde ursprünglich für Arduino_FreeRTOS geschrieben, kann aber auch mit anderen FreeRTOS-Portierungen verwendet werden.

Vorteile von frt.h:

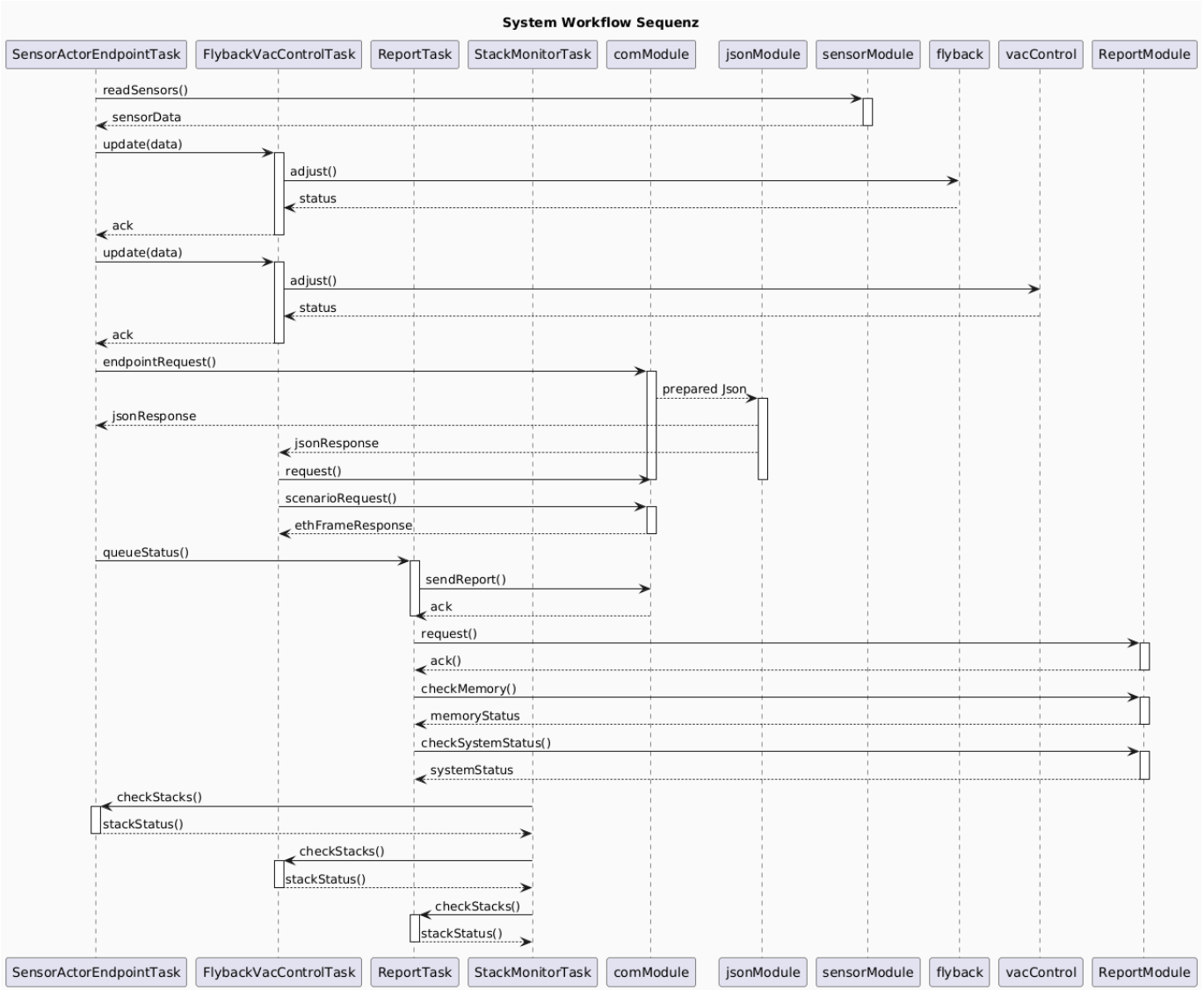
- **Objektorientiertes Design:** Aufgaben (Tasks), Mutexes, Queues, Semaphoren usw. als Klassen
- **Einfache Nutzung:** Statt direkt mit C-APIs zu arbeiten, schreibt man übersichtliche C++-Klassen mit Methoden wie run(), msleep(), post(), wait()
- **Statische Allokation bevorzugt:** Spart RAM und erhöht Vorhersagbarkeit
- **Unterstützung für Interrupt-Kommunikation:** Z. B. durch preparePostFromInterrupt(), finalizePostFromInterrupt()
- **Multitasking auf engstem Raum:** Auch kleine Controller können damit effektiv in mehrere Aufgaben gegliedert werden
- **Intelligenter Umgang mit Zeitscheiben (Ticks):** Methoden wie msleep(milliseconds, remainder) vermeiden Jitter

Die API ist bewusst schlank und auf das Wesentliche reduziert, mit Fokus auf Effizienz, Verständlichkeit und Sicherheit im Multitasking-Betrieb.

Zusammenspiel der Komponenten

Durch das Zusammenspiel von **Arduino_FreeRTOS** und **frt.h** wird die Verwendung eines Echtzeitbetriebssystems auf Arduino deutlich zugänglicher und robuster. Entwickler können ihre Projekte strukturierter und modularer aufbauen – mit echter Multitaskingfähigkeit, sauberer Trennung von Aufgaben und sicherer Synchronisation von Ressourcen (Daten, Sensoren, Aktoren). Das ist besonders nützlich bei unserer Anwendung mit diversen Tasks und vielen Schnittstellen.

Unser System Workflow Sequenz



RELEVANTE CODE SCHNIPSEL EINFÜGEN; VORALLEM FFRESW.ino ABER AUCH ZENTRALE LIBRARYS UND METHODEN/FUNKTIONEN!!!!