Textbaustein: Methodische Herleitung der C++17 Entscheidung

4.1 Problemstellung: Die Grenzen von ANSI-C in sicherheitskritischen Systemen

In der klassischen Embedded-Entwicklung (C90/C99) werden Konfigurationen und Konstanten häufig über Präprozessor-Makros (#define) gelöst. Dies führt zu zwei signifikanten Risiken im Kontext der Funktionalen Sicherheit (ISO 26262):

1. **Mangelnde Typsicherheit:** Makros sind reine Textersetzungen ohne semantische Prüfung durch den Compiler.
2. **Späte Fehlererkennung:** Fehler in der Logik oder Dimensionierung treten oft erst zur Laufzeit auf (Runtime), statt zur Kompilierzeit (Compile-time).

4.2 Lösungsansatz: „Shift-Left“ durch C++17

Das Projekt ersetzt unsichere C-Konstrukte durch typstarke C++17-Mechanismen. Das Ziel ist die **Verschiebung der Fehlererkennung** von der Hardware-Integrationsphase (teuer, gefährlich) in die statische Analysephase (günstig, sicher).

Wir wenden das Prinzip der *Zero-Overhead Abstraction* an: Die Abstraktion kostet keine CPU-Zyklen zur Laufzeit, da sie bereits vom Compiler aufgelöst wird.

4.3 Analyse der Sprachmittel (Vergleich)

Die folgende Tabelle stellt die gewählten C++17-Konstrukte ihren C-Äquivalenten gegenüber und bewertet sie hinsichtlich *Safety* (Betriebssicherheit) und *Security* (Angriffssicherheit).

Kriterium	**Legacy C Ansatz**	**C++17 Lösung (Projektstandard)**	**Safety-Gewinn**
Konstanten	#define TIMEOUT 100	static constexpr uint32_t TimeoutMs = 100;	**Hoch:** Strenge Typbindung. Der Compiler verhindert, dass TimeoutMs versehentlich als float oder char interpretiert wird.
Arrays / Puffer	int buf[10]; *(Verfällt zu Pointer, Länge geht verloren)*	std::array<int, 10> buf;	**Sehr Hoch:** Die Länge ist Teil des Typs. Funktionen wie .at() (sofern aktiv) oder

			Iteratoren verhindern Buffer-Overflows.
Optionale Werte	int* val (kann NULL sein) oder Magic Numbers (-1)	std::optional<int> val	**Mittel:** Erzwingt explizite Prüfung (val.has_value()) vor dem Zugriff. Vermeidet Dereferenzierung von Null-Pointern.
Hardware-Prüfung	Laufzeit-Check: if (sizeof(Reg) != 4) Error();	Kompilierzeit-Check: static_assert(sizeof (Reg) == 4);	**Exzellent:** Der Build bricht ab, wenn die Hardware-Struktur nicht passt. Es wird kein fehlerhafter Code generiert.

4.4 Konsequenz für die Ressourcennutzung

Trotz der höheren Abstraktionsebene steigt der Ressourcenverbrauch auf dem ESP32-S3 nicht signifikant.

- **Flash-Speicher:** Durch constexpr werden Berechnungen (z. B. Register-Bitmasken) bereits zur Kompilierzeit ausgeführt. Im Binary landet nur die fertige Konstante, identisch zu einem manuell optimierten C-Code.
- **RAM:** std::array und std::optional liegen auf dem Stack und benötigen keine dynamische Speicherverwaltung (Heap). Dies eliminiert das Risiko von Speicherfragmentierung, ein kritisches Kriterium für Langzeitstabilität.

4.5 Fazit zur Sprachwahl

Die Entscheidung für C++17 folgt der **AUTOSAR C++14 Guidelines** (aktualisiert). Sie ermöglicht eine defensive Programmierung, bei der der Compiler als erstes statisches Analyse-Tool fungiert. Dies reduziert die Wahrscheinlichkeit von *Undefined Behavior* (UB) drastisch gegenüber klassischem C.