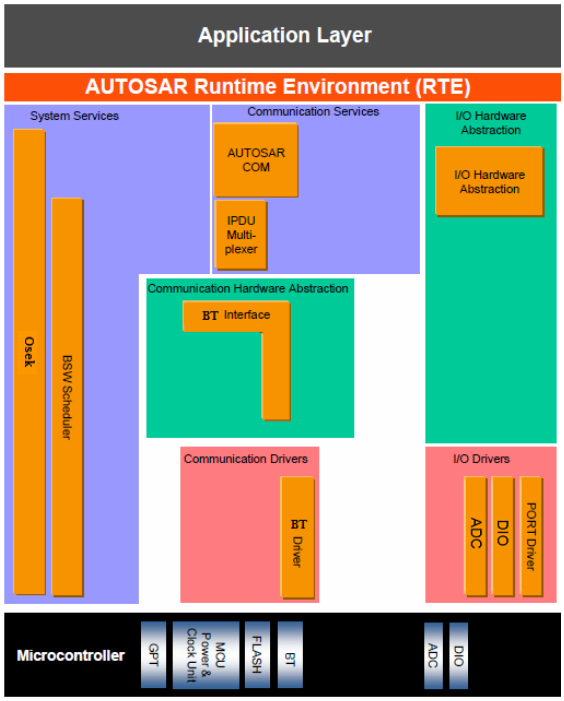
VFB Beschreibung

- Entscheidungen:
  - Umgesetzte Ports:
    - Trigger Port
      - Im Metamodell genannt: Event_Out/Event_In.
      - Möglichkeit auf Applikationsebene Events zu versenden.
      - Einstellmöglichkeit ob blockierend oder nicht blockierend.
    - Sender/Receiver Port
      - Im Metamodell genannt: Sender/Receiver.
      - Möglichkeit auf Applikationsebene Nachrichten zu versenden.
      - Auf Metaebene ist es möglich Multicast-Nachrichten zu versenden, allerdings wurde dies an keinem Beispiel getestet.
        →Codegenerierung ist allerdings vorhanden
      - Benutzt in Codegenerierung einen Trigger Port
    - Server/Client Port
      - Im Metamodell genannt: Server/Client.
      - Möglichkeit auf Applikationsebene Funktionen aufzurufen.
      - Nur Grundgerüst umgesetzt, jedoch keine Codegenerierung implementiert.
  - Umgesetzte Komponenten:
    - Software Komponente
    - Sensor/Aktor Komponente
  - Strikte Trennung von logischer-/ und Hardwareseite.
  - Unterstützung von explizit konfigurierten Alarmen um Tasks zu starten.
  - Unterstützung von mehreren Runnables in einem Task.
- Einschränkungen
  - Nur ein Sender einer Sender/Receiver Kommunikation auf einem Brick möglich.
  - Server/Client Kommunikation lässt sich zwar konfigurieren, wird aber nicht generiert.
  - Explizite Events lassen sich zwar konfigurieren und werden auch automatisch generiert, wurden allerdings nicht getestet.
  - Die Klasse Message wird für das Senden und Empfangen von Nachrichten nicht benötigt.
  - Alle Objekte der Klassen die von SenderPorts oder ReceiverPorts erben, müssen wie die RTE-Funktionen benannt werden.
  - Bei einem konfigurierten I²C Expander, müssen die Attribute Portname und Pinnummer den gleichen Wert haben, nämlich den des Ports am Brick.
  - Entgegen der Logik ist es Software Komponenten möglich durch Konfiguration Zugriff auf OS-Ports zu erlangen.

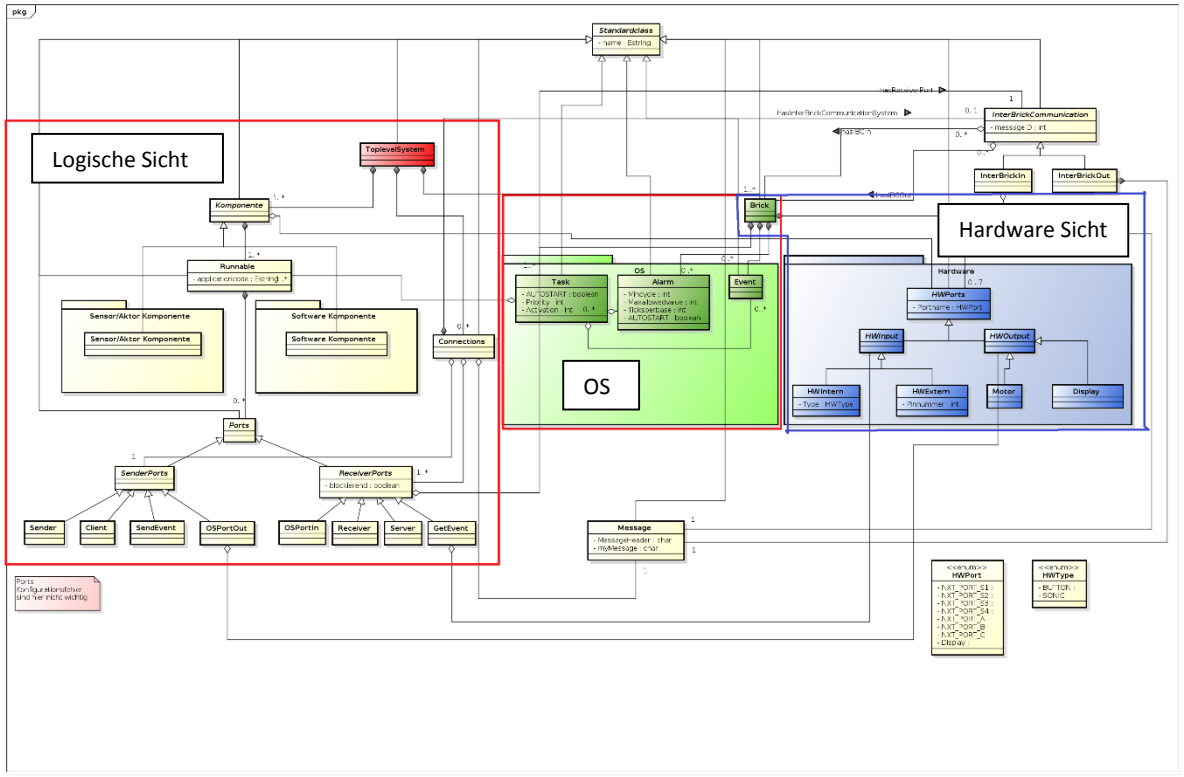
Schichtenarchitektur



Beschreibung der einzelnen Schichten:

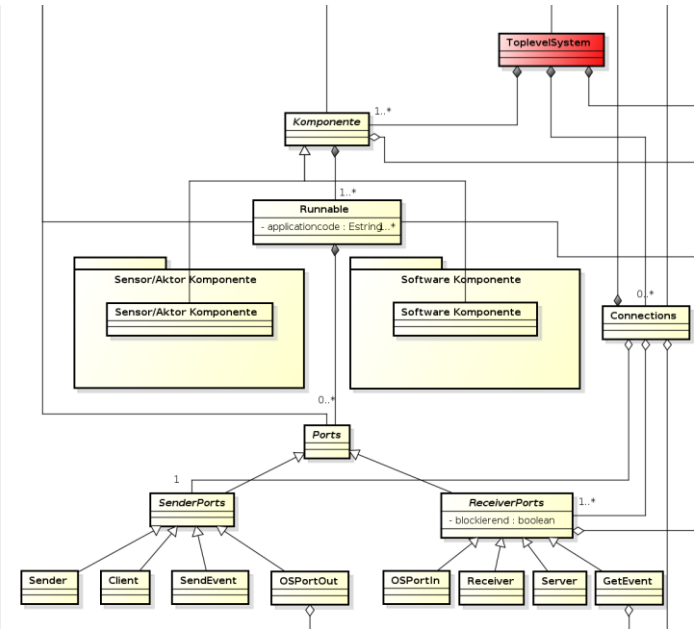
- I/O Drivers
  - Verwendung der NXTOSEK Hardware Funktionen.
- I/O Hardware Abstraction
  - Kapselung der I/O Drivers.
  - Funktionsaufruf, der durch unterschiedliche Makros unterschiedliche Hardwarefunktionalitäten anspricht.
- Communication Drivers
  - Wurde unter dem Namen BT_IMPLIZIT_MASTER/BT_IMPLIZIT_SLAVE umgesetzt.
  - Prüft ob von höhergelegener Schicht Nachrichten versendet werden sollen, oder Nachrichten angekommen sind.
- Communication Hardware Abstraction
  - Wurde unter dem Namen BT_INTERFACE_READER/BT-INTERFACE-WRITER umgesetzt.
  - Reicht beim Versenden die Nachrichten der höheren Schicht an die niedrigere weiter. Beim Empfangen, wird die Nachricht an die nächsthöhere Schicht weitergeleitet.
- Communication Service
  - Kapselung aller Kommunikationsschichten.
- System Services
  - Das eigentliche NXTOSEK-Betriebssystem.

Meta Modell



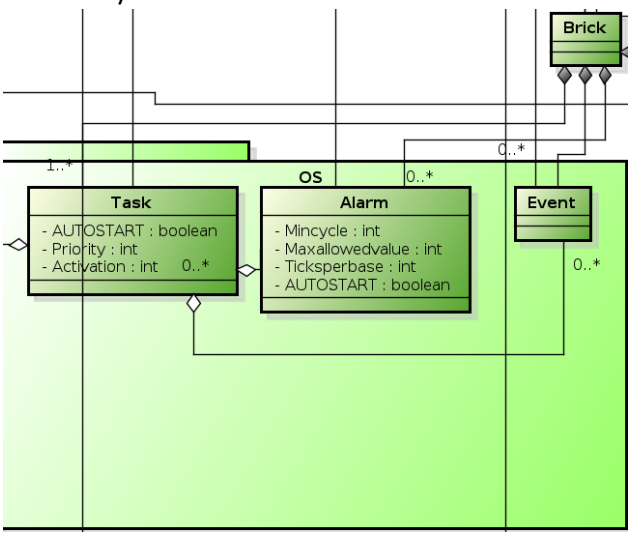
- Die umrandeten Bereiche werden auf der nächsten Seite genauer erläutert.
- Soll eine Verbindung über mehrere Bricks laufen, so wird die Klasse InterBrickCommunication benötigt, in welcher festgehalten wird, wer der Sender und wer der Receiver ist.
- Die Toplevel Klasse heißt Standardclass, welche allen anderen Klassen das Attribut „Name“ vererbt.
- Es wurden zur einfacheren Konfiguration enums für die Hardwareports und Hardwaretypen erstellt.

- Logische Sicht:



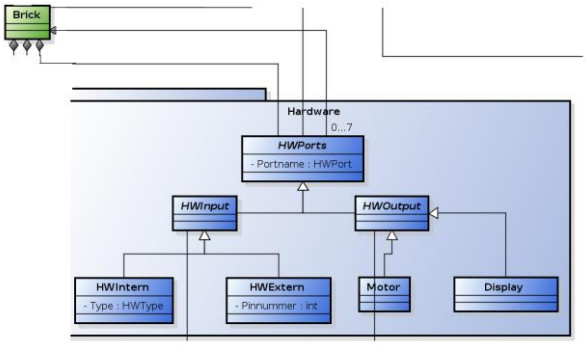
- Komponente kann 1…* Runnables haben.
- Jede Runnable kann 1…* Ports haben.
- Ein Port ist einer Runnable zugeordnet.
- Die Klasse Connection beschreibt die logische Kommunikation. Sie besteht aus einem Sender Port und beliebig vielen Receiver Ports.

- Betriebssystem:



- Es lassen sich Tasks konfigurieren, welche dann ins Oil File übernommen werden.
- Es gibt die Möglichkeit Alarme zu konfigurieren, über welche sich Tasks aktivieren lassen. Die zugehörigen Counter werden implizit erstellt.
- Es lassen sich explizite Events konfigurieren.
- All diese Komponenten werden einem Brick zugeordnet

- Hardware:



- Alle Hardwareports sind einem Brick zugeordnet.
- Es gibt sowohl Input als auch Output Ports
- Bei den Input-Ports wird zwischen HWIntern (NXT-Ports), HWExtern (über I²C Expander unterschieden.
- Als Output sind Motoren und Displays möglich.

Beschreibung der Code-Generatoren

Vor der Codegenerierung müssen folgende Punkte beachtet werden:

1. Soll ein Task zyklisch aufgerufen werden, so muss im RunnableCode am Ende ein TerminateTask(); eingefügt werden, da sämtliche Runnables in einer while(true) Schleife laufen. Dies steht mit der Unterstützung von mehreren Runnables in einem Task in Konflikt, da andere Runnables, die zu diesem Task gehören, nicht weiter ablaufen.
2. In der Konfiguration muss für jede Runnable der richtige Pfad zum eigentlichen Code eingetragen sein.
3. Der Pfad zum ecrobot Makefile Ordner, muss in Zeile 1448 in der Datei „ShootingmachineemfmodelExample.java“ angepasst werden. Als Default ist "C:\\nxtOSEK\\ecrobot" eingetragen.

Es wurde mithilfe eines EMF-Projekts der Codegenerator für alle Dateien in der Datei „ShootingmachineemfmodelExample.java“ implementiert. Diese Datei befindet sich unter „Modell/EMF/ShootingMachine.tests/src/shootingmachineemfmodel/tests“.

Die Datei besteht aus mehreren Funktionen, die alle eine Teilaufgabe der Codegenerierung übernehmen.

Die Funktion „generateOilFile“ generiert den Inhalt des Oil-Files und konfiguriert so das Betriebssystem.
Die Funktion „generatecFile“ generiert das Gerüst der c-Datei mit allen Tasks, Runnables, Alarmen, Counter und Events.
Die Funktion „generatedynamiccFile“ kümmert sich um die Kommunikation. Sie generiert in Abhängigkeit der Konfiguration die nötigen Codesequenzen, die für die Kommunikation notwendig sind. Dabei ist es unter anderem entscheidend, ob über mehrere Bricks kommuniziert wird.
Die Funktion „generateComService“ passt unter anderem die Defines für den Com-Service in der Datei YASA_generated_variables.h an.
Die Funktion „setPortDefines“ setzt alle Defines, die für das Zusammenspiel von Brick und externer Hardware nötig sind, hierbei wird beispielsweise festgelegt, welche Art von Hardware angeschlossen ist und an welchem Port diese angeschlossen ist.

Der wichtigste Schritt zum Einlesen einer Konfiguration ist die Anpassung des Pfads in Zeile 1173. Wurde die Konfiguration erstellt und abgespeichert kann der Java Code gestartet werden, wobei die Konfiguration eingelesen wird und es werden alle Dateien generiert.

Es wird pro Brick ein Ordner mit dem Namen des konfigurierten Bricks im Pfad „Modell/EMF/ShootingMachine.tests“ erstellt. In diesen Ordnern werden die entsprechenden Oil-Files angelegt. Außerdem wird pro Brick ein Makefile angelegt. Die RTE_OSPort Funktionen werden zusätzlich in den Ordner kopiert. Alle anderen RTE-Funktionen werden im Makefile mitcompiliert. Darüber hinaus werden wichtige Header-Files in die Ordner kopiert. Im Header-File YASA_generated_variables.h werden alle für die Kommunikationen und Input-/ und Ouputkomponenten erforderlichen Defines gesetzt. Um in der Communication Hardware Abstraction Schicht den richtigen Ablauf festzulegen, gibt es dort jeweils ein Define für das Versenden und eines für das Empfangen von Nachrichten. Beim Empfangen von Nachrichten wird mithilfe eines switch case die richtige Anweisung in Abhängigkeit von einer impliziten id ausgeführt. Wurden keine ein-/ oder ausgehenden Nachrichten konfiguriert, so wird dieser implizite Task automatisch beendet.

Darstellung der persönlichen Aufgaben und Umfänge

Name	Aufgabe	Umfang in Stunden
Florian Laufenböck	Projektbeschreibung Komponentendiagramm/Komponentenbeschreibung Namenskonvention Autosar Dokument (dieses Dokument in verschiedenen Versionen/Ausarbeitungen) Erarbeiten des Meta Modell: - Grobe Planung und Aufteilung - Initiale Version - Version 1.1 & 1.2 Funktions API: - RTE API (Applikationsspezifisch) - Applikations API Test: - Tests definieren - Tests durchführen - Komponententests - Bugfixing Codegenerierung: - Unterstützung bei Grundgerüst - ComService Generierung	3 3 3 5 5 7 10 3 3 1 5 5 10 10 10 **Gesamt: 83**
Markus Wildgruber	Projektbeschreibung Komponentendiagramm/Komponentenbeschreibung Namenskonvention Autosar Dokument (dieses Dokument in verschiedenen Versionen/Ausarbeitungen) Erarbeiten des Meta Modell: - Grobe Planung und Aufteilung - Initiale Version Funktions API: - RTE API (Applikationsspezifisch) - Applikations API Test: - Tests definieren - Tests durchführen - Komponententests - Bugfixing Codegenerierung: - RTE-Eventgenerierung - Unterstützung Sender/Receiver Komm.	3 3 3 5 5 7 3 3 1 5 5 10 10 10 **Gesamt: 73**
Philipp Eidenschink	Projektbeschreibung Komponentendiagramm/Komponentenbeschreibung Namenskonvention Autosar Dokument (dieses Dokument in verschiedenen Versionen/Ausarbeitungen) Erarbeiten des Meta Modell: - Grobe Planung und Aufteilung - Initiale Version Funktions API:	3 3 3 5 5 7

	- RTE API (Applikationsspezifisch) - Applikations API Test: - Tests definieren - Tests durchführen - Komponententests - Bugfixing Codegenerierung: - Grundgerüst - Oil File Generierung - Unterstützung bei Problemen	3 3 1 5 5 10 13 10 7 **Gesamt: 83**
Tim Schmidl	Projektbeschreibung Komponentendiagramm/Komponentenbeschreibung Namenskonvention Autosar Dokument (dieses Dokument in verschiedenen Versionen/Ausarbeitungen) Erarbeiten des Meta Modell: - Grobe Planung und Aufteilung - Initiale Version Funktions API: - RTE API (Applikationsspezifisch) - Applikations API Test: - Tests definieren - Tests durchführen - Komponententests - Bugfixing Codegenerierung: - Sender/Receiver Ports	3 3 3 5 5 7 3 3 1 5 5 10 12 **Gesamt: 65**
Tobias Schwindl	Projektbeschreibung Komponentendiagramm/Komponentenbeschreibung Namenskonvention Autosar Dokument (dieses Dokument in verschiedenen Versionen/Ausarbeitungen) Erarbeiten des Meta Modell: - Grobe Planung und Aufteilung - Initiale Version - Hardwarespezifikation Funktions API: - RTE API (Applikationsspezifisch) - Applikations API Test: - Tests definieren - Tests durchführen - Komponententests - Bugfixing Applikationscode OSPort Funktionen implementieren und testen	3 3 3 5 5 7 5 3 3 1 5 5 10 7 10 **Gesamt: 75**