HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

DIPLOMARBEIT

Herr Ing.
Manfred Ehrengruber

Praxisorientierte Herangehensweise an die Entwicklung von Embedded-Software Produkten

Mittweida, 2016

Fakultät Angewandte Computer-
und Biowissenschaften

DIPLOMARBEIT

Praxisorientierte Herangehensweise an die Entwicklung von Embedded-Software Produkten

Autor:
Ing. Manfred Ehrengruber

Studiengang:
Technische Informatik

Seminargruppe:
KT12wWA-F

Erstprüfer:
Prof. Dr.-Ing. Uwe Schneider

Zweitprüfer:
Prof. Dr.-Ing. Thomas Beierlein

Einreichung:
Mittweida, 10.10.2016

Verteidigung/Bewertung:
Mittweida, 2016

Faculty Applied Computer
Sciences & Biosciences

DIPLOMA THESIS

Practice-based approach to the development of embedded software products

author:
Ing. Manfred Ehrengruber

course of studies:
Computer Engineering

seminar group:
KT12wWA-F

first examiner:
Prof. Dr.-Ing. Uwe Schneider

second examiner:
Prof. Dr.-Ing. Thomas Beierlein

submission:
Mittweida, 10.10.2016

defence/ evaluation:
Mittweida, 2016

Bibliografische Beschreibung:

Ehrengruber, Manfred:
Praxisorientierte Herangehensweise an die Entwicklung von Embedded-Software Produkten. - 2016. - VII, 76, XV S., 39 Abbildungen, 4 Tabellen
Mittweida, Hochschule Mittweida, Fakultät Angewandte Computer- und Biowissenschaften, Diplomarbeit, 2016

Referat:

Die vorliegende Arbeit befasst sich mit der Erarbeitung von konkreten Herangehensweisen und Software-Architekturen für die Entwicklung der Software von eingebetteten Systemen. Es werden die Themen Objektorientiertes Programmieren, Programmablauf, Konfiguration, Persistenz, Benutzerschnittstelle, Berichte, Kommunikation, und Fehlermanagement behandelt.

Abstract:

The present work deals with the design of concrete approaches and software architectures for developing embedded-software. The following topics will be addressed: object-oriented programming, program execution sequence, configuration, persistence, user interface, reports, communication and error management.

Inhalt

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ASCII	American Standard Code for Information Interchange
ASI	Aktor-Sensor-Interface
bzw.	beziehungsweise
CAN	Controller Area Network
CDC	Communications Device Class
CIA	CAN in Automation
COM	Communication port
EEPROM	Electrically Erasable Programmable Read-Only Memory
EIB	Europäischer Installationsbus
ETX	End Of Text
FIFO	First In First Out
GUI	Graphical-User-Interface (grafische Benutzerschnittstelle)
HCI	Human-Computer-Interaction
HD	High Definition
HW	Hardware
I/O	Input/Output (Eingabe/Ausgabe)
ID	Identifier (Identifikator)
IDE	Integrated Development Environment (integrierte Entwicklungsumgebung)
IEC	International Electrotechnical Commission
IP	Internet Protocol
ISO	International Organization for Standardization
IT	Information Technology (Informationstechnologie)
LAN	Local Area Network
LED	Light-Emitting Diode (Leuchtdiode)
LIN	Local Interconnect Network
LNC	Local Control Network
MAC	Macintosh
MAX	Maximum
MCI	Mensch-Computer-Interaktion
MIN	Minimum
MMK	Mensch-Maschine-Kommunikation
MOST	Media Oriented Systems Transport
MRAM	Magnetoresistive random-access memory
MUTEX	Mutual exclusion (wechselseitiger Ausschluss)
OS	Operating System (Betriebssystem)
OSI	Open Systems Interconnection
PAR	Parameter
PC	Personal Computer
RAM	Random Accesss Memory
RTOS	Real-Time Operating System

RX	Receiver
SD Card	Secure Digital Card
STX	Start Of Text
TCP	Transmission Control Protocol
UART	Universal asynchronous receiver/transmitter
UI	User-Interface (Benutzerschnittstelle)
URL	Uniform Resource Locator
USB	Universal Serial Bus
UTF	Unicode Transformation Format
VCP	Virtual COM Port
vgl.	vergleiche (bei indirekten Zitaten eingesetzt.)
WIN	Windows
WLAN	Wireless Local Area Network
WPF	Windows Presentation Foundation
WWW	World Wide Web
XAML	Extensible Application Markup Language
z.B.	zum Beispiel

1 Einleitung

1.1 Motivation

Beim Entwurf und der Implementierung von Software für eingebettete Systeme steht der Entwickler immer wiederkehrenden Szenarien gegenüber. Aus der Praxis ist ersichtlich, dass eingebettete Systeme sich in ihrem Aufbau oft sehr ähnlich sind und dass nur ein Teil davon für dessen Einsatz speziell entwickelt wird. Entsprechend gleichen sich die Architekturen der eingebetteten Software. Es ist sinnvoll, immer wieder auftretende Software-Architekturen festzuhalten, zu dokumentieren, um bei den nächsten Entwürfen von eingebetteten Systemen darauf zurückgreifen zu können.

Für den objektorientierten Entwurf existiert mittlerweile zahlreiche Literatur, die sich mit Architektur- und Entwurfsmuster befasst. Diese Muster kommen auch in der eingebetteten Software zum Einsatz. Allerdings stellen diese Muster keine konkreten Lösungswege dar, sondern vielmehr eine, im Laufe der Zeit, erprobte Verfahrensweise zur Findung einer konkreten Lösung. (vgl. (Goll, 2014), (Hruschka, 2011) (Starke, 2013))

Die, speziell für eingebettete Systeme und eingebettete Software verfasste Literatur befasst sich dagegen mit den systemtechnischen Grundlagen der Hardware- und Software-Entwicklung. Dabei wird allgemein die Entwicklung eines eingebetteten Systems, von der Elektronik (Hardware) bis hin zur eingebetteten Software (Software-Modellierung) behandelt, aber keine konkreten Konzepte zur Implementierung dargestellt. (vgl. (Berns, 2010), (Marwedel, 2008), (Qian, 2009), (Noble, 2001))

Nach Auffassung des Autors deckt die recherchierte Literatur einen breiten Bereich der (technischen) Informatik ab. Sie dient zum Einstieg in die (eingebettete) Software und bietet das Wissen und die Technologien, Lösungswege zu erarbeiten. Konkrete Konzepte werden aber nicht vorgestellt.

1.2 Zielsetzung

In dieser Arbeit werden, basierend auf bestehende Literatur, konkrete Konzepte und Herangehensweisen für die Entwicklung von eingebetteter Software dargestellt. Mit Hilfe dieser soll es dem Entwickler bzw. dem Entwickler-Team möglich sein, immer wiederkehrende Szenarien abzudecken.

1.3 Methodik

Es werden mittels Brainstorming Themen und Teilbereiche der Entwicklung von eingebetteter Software, bei denen in der Vergangenheit immer wiederkehrende, gleichende Software-Architekturen erarbeitet wurden, ermittelt. Diese werden isoliert voneinander als Herausforderungen angeführt.

Die Literatur-Recherche wird in zwei Richtungen betrieben. Zum einen wird eine Literatur-Recherche bezogen auf Software-Architektur-/Entwurfs-Muster initiiert, zum anderen wird eine Literatur-Recherche zum Thema der eingebetteten Systeme vollzogen.
Die Literatur wird größtenteils aus der Hochschulbibliothek und dem existierenden Bestand des Autors bezogen. Die Recherche der Literatur beschränkt sich auf den Zeitraum von 2000 – 2016.

Es wird pro Herausforderung ein Kontext beschrieben damit der Leser den Bezug der Herausforderung wahrnehmen kann.
Darauf folgend wird die eigentliche Herausforderung beschrieben um den Leser aufmerksam zu machen, wo die Problembereiche liegen.
Im letzten Teil wird als Lösung das erarbeitete Konzept präsentiert.

1.4 Kapitelübersicht

Im Kapitel 1 ist die Einleitung dieser Arbeit verfasst.

Das Kapitel 2 befasst sich mit den Grundlagen der eingebetteten Systeme und der eingebetteten Software.

Im Kapitel 3 werden die Herausforderungen angeführt und behandelt.

2 Grundlagen

In diesem Kapitel werden die Grundlagen der eingebetteten Software und deren Einsatzgebiete erläutert.

2.1 Eingebettete-Software (Embedded Software)

2.1.1 Definition

„The embedded system software is an application-specific software that is dedicated to perform predesigned specific tasks repeatedly … “
“ … once the embedded software is loaded into the system, the software is expected to run for a very long time by itself without any changes to the software. The software developers must guarantee the reliability, safety, and correctness of the embedded software.” (Qian, 2009)

Eingebettete Software ist die Software, die für eingebettete Systeme verfasst wird, welche für spezielle Aufgaben entwickelt werden.

„Ein eingebettetes System ist ein binärwertiges digitales System (Computersystem), das in ein umgebendes technisches System eingebettet ist und mit diesem in Wechselwirkung steht.“ (Wikipedia, Embedded Software Engineering, 2015)

2.1.2 Eingebettete Software in der Praxis

Eingebettete Software (Embedded Software) wird als Überbegriff für die Software, die speziell für ein bestimmtes Gerät konzeptioniert und verfasst wird, verwendet.
Konkret kann die eingebettete Software aus Bootloader, Firmware, OS und Applikation bestehen. Diese Komponenten werden je nach Größe des eingebetteten Systems eingesetzt.

Bei kleineren Systemen besteht die eingebettete Software meist aus einer Programmabfolge, die vom Reset-Vektor weg gestartet wird. In der Praxis nennt man diese Programmabfolgen Firmware. Die Firmware wird konkret für eine Ziel-Hardware verfasst, die bei der Produktion des Gerätes in den Mikroprozessor programmiert wird. Die Firmware ist vom Bediener nicht aktualisierbar.

Bei mittleren Systemen besteht die eingebettete Software aus Bootloader und User-Programm. Damit besteht die Möglichkeit, das Programm nach der Produktion auszutauschen. Bei bestimmten Systemen wird auch der Bootloader austauschbar gestaltet. Das

User-Programm wird, ebenfalls wie die Firmware, konkret für die Ziel-Hardware verfasst. Gelegentlich kommen im User-Programm schmale (Echtzeit)-Betriebssysteme zum Einsatz.

Bei großen Systemen werden meistens vollständige Betriebssysteme eingesetzt. Das Betriebssystem übernimmt die Hardware-Abstraktion. Die darauf laufenden Applikationen werden hardwareunabhängig verfasst.

Genauso wie bei der allgemeinen hardwareunabhängigen Software-Entwicklung, beispielsweise für den PC, gibt es auch in der eingebetteten Software folgende Teilgebiete:

- Programmablauf
- Daten-Verarbeitung/-Management
- Benutzerschnittstelle
- Kommunikationsschnittstelle

Bei der eingebetteten Software kommt noch ein zusätzliches Teilgebiet zum Einsatz:

- I/O-Schnittstelle

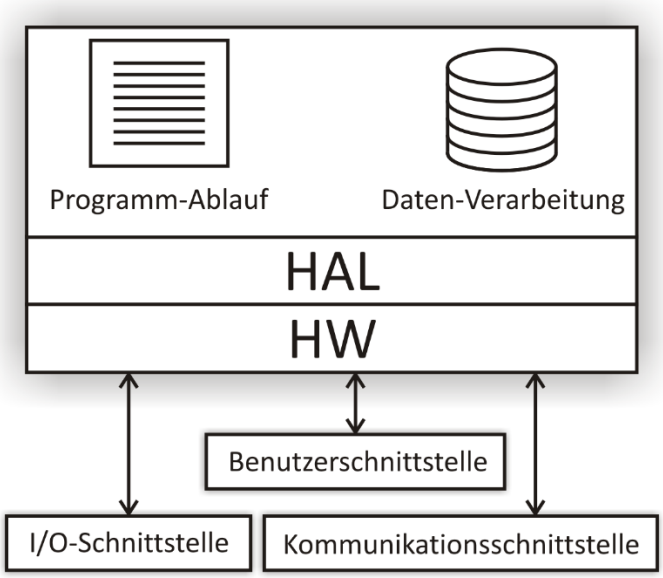

Abbildung 1: Software Teilgebiete
(eigene Darstellung)

2.1.2.1 Programmablauf

Der Programmablauf bestimmt das Verhalten eines eingebetteten Systems. Der Programmablauf entspricht einem Programm.
„Ein Programm ist eine statische Folge von Anweisungen in einer Programmiersprache unter Nutzung von Daten.“ (Schneider, 2012)

2.1.2.2 Daten-Verarbeitung/-Management

Die Daten-Verarbeitung bzw. das Daten-Management befasst sich mit dem Verarbeiten und dem Ablegen von Daten. Auch das Persistieren von Daten ist Teil dieses Gebietes.

2.1.2.3 Benutzerschnittstelle

Die Benutzerschnittstelle ist Teil der „Mensch-Computer-Interaktion“. Sie ist die Schnittstelle zwischen Menschen und Computersystemen. (Schneider, 2012)

2.1.2.4 Kommunikationsschnittstelle

Die Kommunikationsschnittstelle ist Teil der Datenkommunikation und stellt sämtliche Komponenten bereit, um Datenkommunikation ausführen zu können.
„Datenkommunikation (data communication) ist dadurch gekennzeichnet, dass die Kommunikationspartner binärcodierte Daten senden und/oder empfangen und speichern können.“ (Schneider, 2012)

2.1.2.5 I/O-Schnittstelle

Mit Hilfe der I/O (Input/Output) - Schnittstelle wird die Hardware des eingebetteten Systems bedient. Sensor-Signale werden damit ins System eingebracht, Aktoren und Signalgeber werden damit angesprochen.

2.2 Entwicklung von eingebetteter Software

Bei der Implementierung von Software wird zwischen prozeduraler und objektorientierter Programmierung unterschieden. Während in kleinen eingebetteten Systemen nach wie vor die prozedurale Programmierung (meist in Assembler oder C verfasst) zum Zuge kommt, wird in größeren eingebetteten Systemen die objektorientierte Programmierung herangezogen.

2.2.1 Prozedurale Programmierung

Die prozedurale Programmierung entstand aufgrund der Arbeitsweise von Prozessoren. Prozedurale Programme bestehen aus einer Folge von Anweisungen, die der Reihe nach vom Prozessor abgearbeitet werden. Diese Art der Programmierung wird auch Imperative Programmierung genannt. Folgende Prinzipien finden beim Entwurf von prozeduralen Programmen Beachtung:

- Kontrollstrukturen zur Steuerung der Anweisungen, z.B.: Schleife, Verzweigung.
- Wiederverwendung von Programmteilen, z.B.: mittels Unterprogramme.
- Deklarierte Variablen, festen Datentyps zur Ablage und Bearbeitung von Daten.
- Kapselung mittels Modulen bzw. globalen, statischen und lokalen Variablen.

2.2.2 Objektorientierte Programmierung

Bei der objektorientierten Programmierung steht nicht primär die Anweisungsabfolge für die HW-Architektur im Vordergrund, sondern die Abstraktion einer gegebenen Herausforderung in einem bestimmten Umfeld zur Bildung einer Lösung darin. Ziel ist es, Objekte der realen Welt in geeignete Modelle durch Abstraktion umzusetzen. Folgende Prinzipien finden beim Entwurf von objektorientierten Programmen Beachtung:

(Goll, 2014):

- Weiterentwickelbarkeit
  Damit ist die Erweiterbarkeit im Rahmen bestimmter Architekturen und Wiederverwendbarkeit vorhandener Komponenten anderer Architekturen gemeint.
- Korrektheit
  Die Korrektheit von Software ist dann gegeben, wenn die Software die vorausgesetzte Spezifikation erfüllt.
- Verständlichkeit
  Zur Verständlichkeit tragen Kapselung, Abstraktion, Information Hiding und das Single-Responsibility-Prinzip bei.
- Abstraktion
  Das Verhalten eines Objektes wird durch Schnittstellenmethoden abstrahiert. Außerhalb des Objekts sind nur die Schnittstellenmethodenköpfe sichtbar.

- Kapselung
  Methoden und Daten verschmelzen bei der Kapselung zu einem Objekt.
- Information Hiding
  Die Daten eines Objekts sollen nach außen nicht direkt sichtbar sein.
- Single Responsibility-Prinzip
  Jedes Objekt soll nur eine einzige Aufgabe haben. Alle Methoden des Objektes sollen zur Erfüllung dieser Aufgabe beitragen.
- Lose Kopplungen von Objekten
  Ein Objekt ist Teil eines Systems vieler Objekte. Zwischen diesen Objekten soll eine schwache Wechselwirkung (schwache, lose Kopplung) bestehen.
- Liskovsches Substitutionsprinzip
  Eine Referenz auf ein Objekt einer Basisklasse soll auch auf ein Objekt einer abgeleiteten Klasse zeigen können.
- Design by Contract
  „Design by Contract“ stellt ein Prinzip dar, bei dem formal zwischen Objekten präzise definiert wird, unter welchen Umständen ein korrekter Ablauf des Programms erfolgen wird.
- Verringerung von Abhängigkeiten
  Bei der Erzeugung von Objekten wird die Kontrolle darüber, welches Objekt wann erzeugt wird, von der nutzenden Klasse an eine dafür eigens vorgesehene Instanz abgegeben.

2.2.3 Herausforderungen bei der Entwicklung von eingebetteter Software

Beim Entwurf und dem Verfassen von eingebetteter Software steht der Entwickler besonderen Herausforderungen gegenüber. Zum einen herrscht bei eingebetteten Systemen meist Ressourcen-Knappheit, vorwiegend wenig Speicherplatz, zum anderen wird der Entwickler mit verschiedensten Programmiersprachen und Programmierparadigma konfrontiert.

„Häufig wird beispielsweise angeführt, dass sich die Entwicklung eingebetteter Software in naher Zukunft immer mehr der Entwicklung von IT-Systemen angleichen wird, da sich Standardplattformen etablieren, die, vergleichbar zu Windows und Intelprozessoren, eine Vereinheitlichung der Softwareentwicklung bewirken werden.“ (Berns, 2010)

2.2.3.1 Programmiersprache / Programmierparadigma

Nach Auffassung des Autors ist die am meisten eingesetzte Programmiersprache für das Verfassen von eingebetteter Software C. Vor 15 Jahren wurde ein kleines bis mittleres eingebettetes System (z.B.: Motoren-Regelung, Fieber-Thermometer, Fitness-Tracker) noch in Assembler verfasst. Mittlerweile werden von den Prozessoren-Hersteller C-Compiler/-Linker auch für die kleinsten 8-bit-Derivate angeboten. Assembler wird nur mehr für sehr zeitkritische Abläufe verwendet.

Das am häufigsten verwendete Programmierparadigma ist die imperative Programmierung. Meistens wird eine hardwarenahe Programmierung, angepasst auf einen bestimmten Prozessor, realisiert. Bei größeren eingebetteten Systemen (z.B.: Heizungs- / Smart-Home- Steuerungen, …) kommen die objektorientierte Programmierung und die imperative Programmierung als Programmierparadigma zum Einsatz.

2.2.3.2 Ressourcen

Charakteristisch bei der Entwicklung von eingebetteter Software ist der Umgang mit Ressourcen. Im Gegensatz zum PC verzeichnet ein eingebettetes System meistens Ressourcen-Knappheit. Der Architekt bzw. der Entwickler ist somit angehalten die Software möglichst ressourcenschonend zu konzipieren bzw. zu entwickeln.

In den meisten eingebetteten Systemen fehlt es an genug Speicherplatz für Laufzeitdaten aber auch für Programmcode und persistente Daten.

	Embedded System	**Wireless PDA**	**PC, Workstation**	**Mainframe or Server Farm**
Code Storage	●●	●●		
Code Working Set		●●	●	
Heap and Stack	●●●	●●	●	●
Persistent Data	●●●	●		

Tabelle 1: Importance of Memory Constraints (Noble, 2001)

2.2.3.3 Der Embedded-Software-Entwickler

Den Grundstein zum Berufsbild des Embedded-Software-Entwicklers wurde durch die Einführung von Mikroprozessoren in der Elektronik-Entwicklung gelegt. Waren es anfangs noch die Elektroniker selber, die meist mit der Programmiersprache Assembler imperative Programme zur Ansteuerung deren entwickelten Elektronik-Komponenten verfasst hatten, kamen später spezialisierte Informatiker (technische Informatiker) für die Programmierung dieser Systeme zum Einsatz.

Die Entwicklung von eingebetteten Systemen wurde somit in Hardware- und Software-Entwicklung aufgetrennt. Das Verhältnis des Zeitaufwands kehrte sich im Laufe der Zeit um. Die Zeit, die für die Hardware-Entwicklung aufgebracht werden musste, wurde weniger, die Zeit für die Entwicklung der Software stieg dagegen stetig an. Momentan liegt das

durchschnittliche Aufwandsverhältnis der Hardware-/Software-Entwicklung für mittlere und große eingebettete Systeme bei 10 / 90%.

Mittlerweile ist der Entwicklungsaufwand so hoch, dass mehrere Entwickler in Teams an der Software für eingebettete Systeme arbeiten. Der daraus resultierende Verwaltungsaufwand steht dem der Entwicklung großer Software-Systeme für den PC nichts nach. Verwaltungswerkzeuge wie Quellcode- und Ticket-Verwaltung mussten eingeführt werden. Zudem stieg der Konzeptionierungsaufwand.

Die System-Architektur und deren Dokumentation rückte in den Vordergrund, sie beschreibt die Strukturen des Systems, dessen Bausteine, Schnittstellen und deren Zusammenspiel. (Hruschka, 2011)

Entwickler von eingebetteter Software sind heute nicht mehr nur mit der Hardware-nahen, imperativen Programmierung, sondern auch mit dem Entwurf von objektorientierten Programmen / Applikationen konfrontiert. Das Berufsbild vom technischen Informatiker hat sich jenem des Informatikers in den letzten Jahren zunehmend angenähert.

2.2.3.4 Architektur- und Entwurfsmuster

Im Laufe der Zeit brachte die Software-Entwicklung Vorgehensweisen für immer wiederkehrende Szenarios vor. Diese Vorgehensweisen wurden als Muster (Patterns) in zahlreichen Werken dokumentiert und weiterentwickelt.

„Muster für den Entwurf sind bewährte Lösungsvorschläge für bestimmte Problemstellungen, die beim Entwurf von Systemen beachtet werden sollten, da sie sich bereits in mehreren Systemen bewährt haben.“ (Goll, 2014)
„Das Hauptziel von Mustern für den Softwareentwurf ist es, einmal gewonnene Erkenntnisse wiederverwendbar zu machen und durch ihre Anwendung die Flexibilität einer Architektur zu erhöhen.“ (Goll, 2014)

Diese Muster wurden vorwiegend für die objektorientierte Programmierung erstellt, es gibt aber auch Muster, die für die imperative Programmierung in eingebetteten Systemen eingesetzt werden können.

Grundsätzlich werden die Muster in folgende Teilbereiche unterteilt (Starke, 2013):

- Erzeugungsmuster
- Verhaltensmuster
- Strukturmuster
- Verteilungsmuster
- Integrationsmuster
- Persistenzmuster

Zusätzlich finden in kleinen und mittleren eingebetteten Systemen mit Speicherknappheit folgende Muster Einsatz:
(Noble, 2001):

- Small Architecture
- Secondary Storage
- Compression
- Small Data Structures
- Memory Allocation

Diese Muster sind allgemein gehalten, sie bieten keine konkrete Lösung basierend auf ein konkretes Problem. Vielmehr sind sie ein erprobter Leitfaden zur Bildung der Lösung.

3 Herausforderungen

“A pattern is not just a particular solution to a particular problem. One of the reasons programming is hard is that no two programming problems are exactly alike, so a technique that solves one very specific problem is not much use in general (Jackson, 1995). Instead, a pattern is a generalised description of a solution that solves a general class of problems “ (Noble, 2001)

In dieser Arbeit werden keine Muster im Sinne der obigen Definition entwickelt und verfasst, vielmehr werden konkrete Herangehensweisen für die Lösung von konkreten, in der Welt der eingebetteten Software auftretenden, Herausforderungen definiert.

Die Herangehensweisen bieten dem Entwickler von eingebetteter Software konkrete Vorlagen, die je nach Anwendung entsprechend angepasst und implementiert werden können.
Für jede Herausforderung wird zuerst der Kontext, dann die Problematik und zuletzt die Lösung erläutert. Unterstützend können auf Visualisierungen und Quellcode-Beispiele, verfasst in C und C# zugegriffen werden.

3.1 Übersicht

Herausforderung - Objektorientierte Prinzipien

… befassen sich mit der Herausforderung, wie trotz imperativer Programmierung (z.B.: mittels C) eine objektorientiert-nahe Programmierung realisiert werden kann.

Herausforderung – Programmablauf

… befasst sich mit der Strukturierung vom Verhalten eines eingebetteten Systems, unabhängig von dessen Umgebung (Betriebssystem, Programmiersprache, Hardware, …)

Herausforderung – Konfigurationssystem

… befasst sich mit der Verwaltung von Parameter zur Konfiguration eines eingebetteten Systems.

Herausforderung - Persistenz-System

… befasst sich mit dem Ablegen von Daten in den nichtflüchtigen Speicher.

Herausforderung – Benutzerschnittstelle

… befasst sich mit der Vorgehensweise und den Technologien, eine Mensch-Computer-Schnittstelle zu realisieren.

Herausforderung – Berichte

… befasst sich mit dem Zusammenstellen von Daten für Ausdruck und Kommunikation.

Herausforderung - Kommunikationsschnittstelle

… befasst sich mit der Strukturierung der Datenkommunikation.

Herausforderung – Fehlermanagement

… befasst sich mit der Fehlerbehandlung, Fehlerbenachrichtigung und Fehlerprotokollierung.

3.2 Herausforderung - Objektorientierte Prinzipien

3.2.1 Kontext

Programmiersprache: C
Programmierparadigma: imperative Programmierung

Eingebettete Software wird sehr oft mit der Programmiersprache C und dem Programmierparadigma imperative Programmierung realisiert. Die meisten Mikroprozessoren-Hersteller bieten meist auch C++-Compiler-Tool-Suites an. Diese werden bis jetzt nur selten, vorwiegend wegen der Ressourcenknappheit in eingebetteten Systemen eingesetzt.

Bei kleinen eingebetteten Systemen, wo die Software von einem Entwickler entworfen und verfasst wird, stellt die reine imperative Programmierung kein Problem dar. Die Ressourcenverwaltung, die das eingebettete System bietet, kann direkt mit dem Programmablauf zu einem Programm kombiniert werden.
Bei mittleren bis großen eingebetteten Systemen, wo mehrere Entwickler mit der Entwicklung der Software betraut sind, müssen objektorientierte Prinzipien eingeführt werden.

Bei großen eingebetteten Systemen werden meist Betriebssysteme und eine objektorientierte Programmiersprache/Programmierparadigma eingesetzt. Somit werden die klassischen objektorientierten Prinzipien (siehe (Goll, 2014)) eingesetzt. Aber auch bei diesen Systemen kann die Herangehensweise: „Objektorientierte Prinzipien“ eine strukturelle Unterstützung bieten.

3.2.2 Herausforderungen

Bei größeren Programm-Konstrukten ist Übersichtlichkeit, Weiterentwickelbarkeit und Verständlichkeit, welches eines der wichtigsten objektorientierten Prinzipien entspricht, nicht mehr gewährleistet. Die Herausforderung besteht darin, Programmteile in gekapselter, abstrakter und normalisierter Form in das Gesamt-Programm-Konstrukt einbauen zu können, ohne auf moderne, objektorientierte Programmiersprachen angewiesen zu sein.

Ressourcen des Systems, im weiteren Verlauf Betriebsmittel genannt, sollen von gekapselten Programmteilen verwaltet und benutzt werden.

In Abbildung 2 ist eine herkömmliche Implementierung eines Kartenlesers in ein eingebettetes System veranschaulicht.

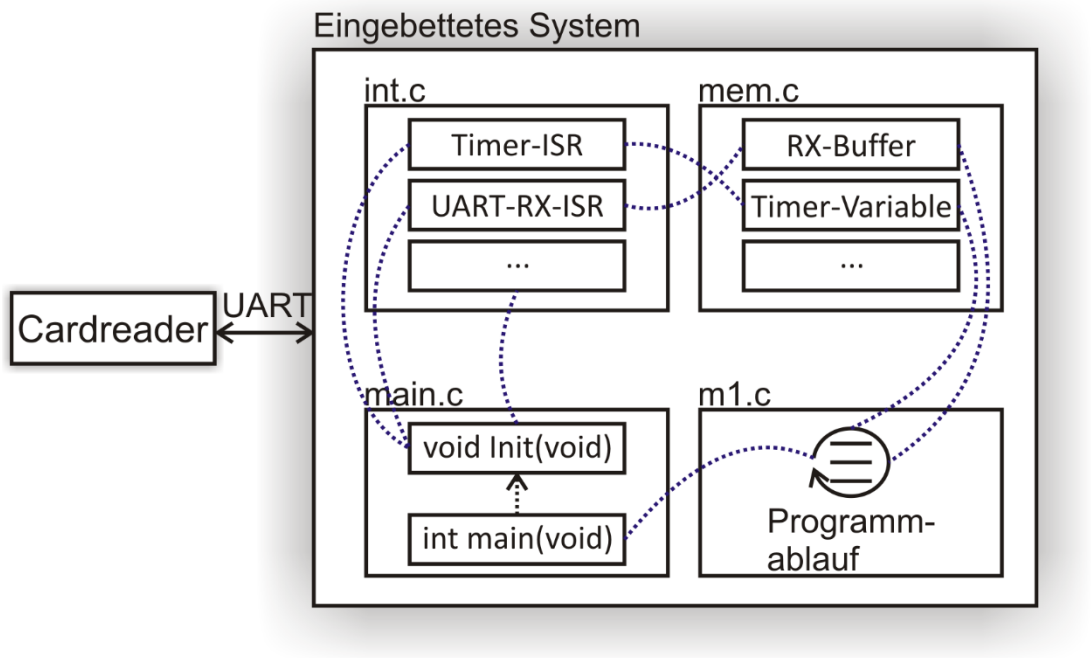

Abbildung 2: Herkömmliche Implementierung eines Kartenlesers (eigene Darstellung)

Der Kartenleser ist mittels UART am Mikroprozessor angebunden. Die UART-Kommunikation wird mittels Interrupt-Service-Routine bedient. Im Programmablauf wird auf globale Variablen zugegriffen und somit der Kartenleser bedient.

Die Problematik dieser Software-Architektur besteht darin, dass der Programmteil für die Bedienung des Kartenlesers im gesamten Programmablauf eingeflochten ist. Somit ist die Übersichtlichkeit, Wiederverwendbarkeit und Weiterentwickelbarkeit nicht gegeben.

Für kleine eingebettete Systeme, die beispielsweise speziell nur für die Bedienung des Kartenlesers entwickelt werden, kann diese Software-Architektur durchaus verwendet werden. Wird allerdings ein Kartenleser in ein bestehendes mittleres bis großes eingebettetes System implementiert, soll eine Kapselung des Quellcodes des Kartenlesers bezweckt werden. Somit können die Programmteile für den Kartenleser ausgetauscht und erweitert werden, beispielsweise aufgrund der Implementierung eines zweiten Kartenlesers.

3.2.3 Lösungen

C ist eine imperative Programmiersprache. Teile der Prinzipien einer modernen, objektorientierten Programmierung können jedoch auch mit C realisiert werden.

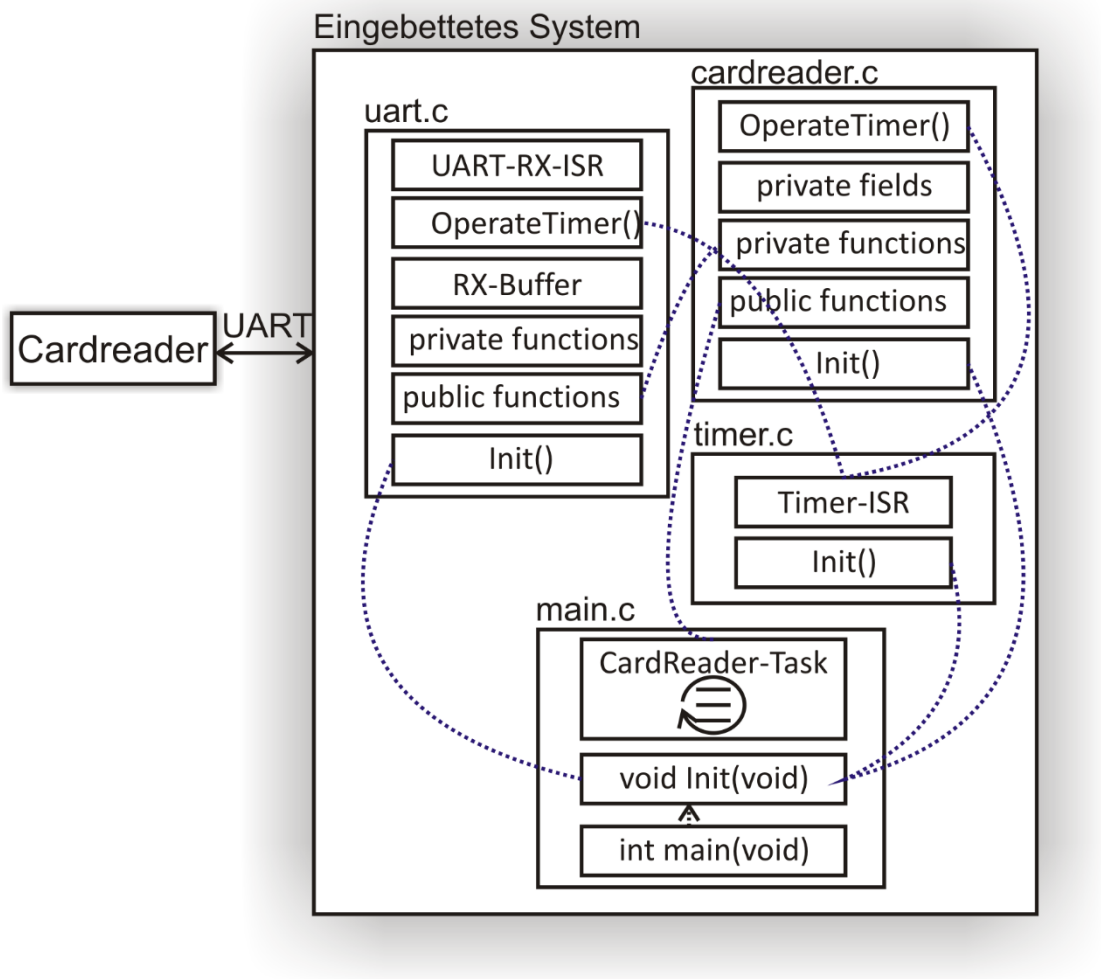

Abbildung 3: Objektorientierte Implementierung eines Kartenlesers (eigene Darstellung)

Ein Objekt wird als C-Modul (C-File) dargestellt. Ein solches „Objekt“ stellt immer eine Abstraktion eines realen Objekts dar. Es wurden Objekte für die Mikroprozessor-Peripherie-Komponenten „uart“ und „timer“ erstellt. Beide Objekte werden vom Objekt „cardreader“ konsumiert, welches wiederrum vom CardReader-Task konsumiert wird. Somit wurde eine Kapselung der Komponenten erreicht. Diese können jederzeit ausgetauscht werden. Wird in einer nächsten Hardware-Revision des eingebetteten Systems die UART-Kommunikationsschnittstelle durch eine USB-Kommunikationsschnittstelle ersetzt, kann das „uart“-Objekt durch ein „usb“-Objekt ausgetauscht werden; die Schnittstelle für den CardReader-Task bleibt gleich. Im CardReader-Task ist der Programmablauf für den Kartenleser implementiert. Dieser „Task“ muss nicht zwingend ein Task im Sinne eines Betriebssystems sein, es kann eine C-Funktion sein, die in einer Programm-Schleife neben anderen „Task“-Funktionen ausgeführt wird.

3.2.3.1 Elemente eines Objekts

In Abbildung 4 werden die Elemente (Teilbereiche) eines C-Objekts dargestellt. Diese Einteilung kann allgemein bei jedem C-Objekt realisiert werden.

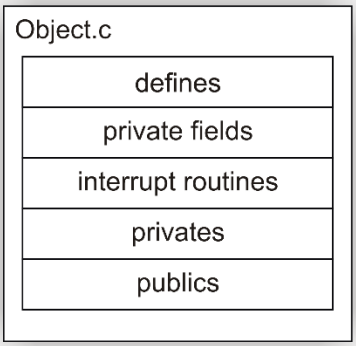

Abbildung 4: Aufbau einer C-Datei als Objekt (eigene Darstellung)

defines

In diesem Bereich sind dem Objekt zugehörige Definitionen vereinbart. Diese Definitionen sind nur in dieser einen C-Datei gültig, somit ist eine Daten-Kapselung gewährleistet.

private fields

In diesem Bereich sind private, dem Objekt zugehörige Variablen deklariert. Ähnlich wie bei privaten Klassen-Member sind diese außerhalb des Objekts nicht zugänglich. Erreicht wird dies mit dem Vorsetzen des C-Schlüsselwort „static“. Solche deklarierten Variablen können von einer anderen C-Datei nicht erfasst werden. Somit ist eine Daten-Kapselung realisiert.

interrupt routines

In diesem Bereich befinden sich die Interrupt-Service-Routinen, die vom Objekt benötigt werden. Diese Funktionen können den direkten Einsprung vom Interrupt-Vektor darstellen (z.B.: UART-RX-Interrupt), sie können aber auch normale, öffentlich-externe Funktionen sein, die vom Interrupt-Vektor-Einsprung eines anderen Objekts aufgerufen werden (z.B.: Timer-Interrupt).

privates

In diesem Bereich befinden sich die privaten Funktionen des Objekts. Diese Funktionen werden nur innerhalb des Objekts konsumiert und sind von einem anderen Objekt nicht zugänglich. Meistens werden hier Hilfsfunktionen platziert.

publics

In diesem Bereich befinden sich die öffentlich-externen Funktionen des Objekts. Diese Funktionen stellen die Schnittstelle dar und definieren somit das Verhalten des Objekts.

3.2.3.2 Übergabe von Schnittstellen

Das Verhalten von Objekten wird mit deren Schnittstellen-Funktionen implementiert. Diese Schnittstellen-Funktionen werden von anderen Objekten konsumiert. Eine Strategie für die Verringerung von Abhängigkeiten stellt die Übergabe der Schnittstellen-Funktion zur Laufzeit dar. Somit muss dem Object_1 das Object_2 zur Kompilierzeit nicht bekannt sein.
Realisiert werden kann dies durch das objektorientierte Prinzip „Dependency Injection“. Hierbei wird dem Object_1 beim Erstellen mittels „constructor injection“ das Object_2 übergeben, welches die Schnittstelle IMethods implementiert. (siehe Abbildung 5)

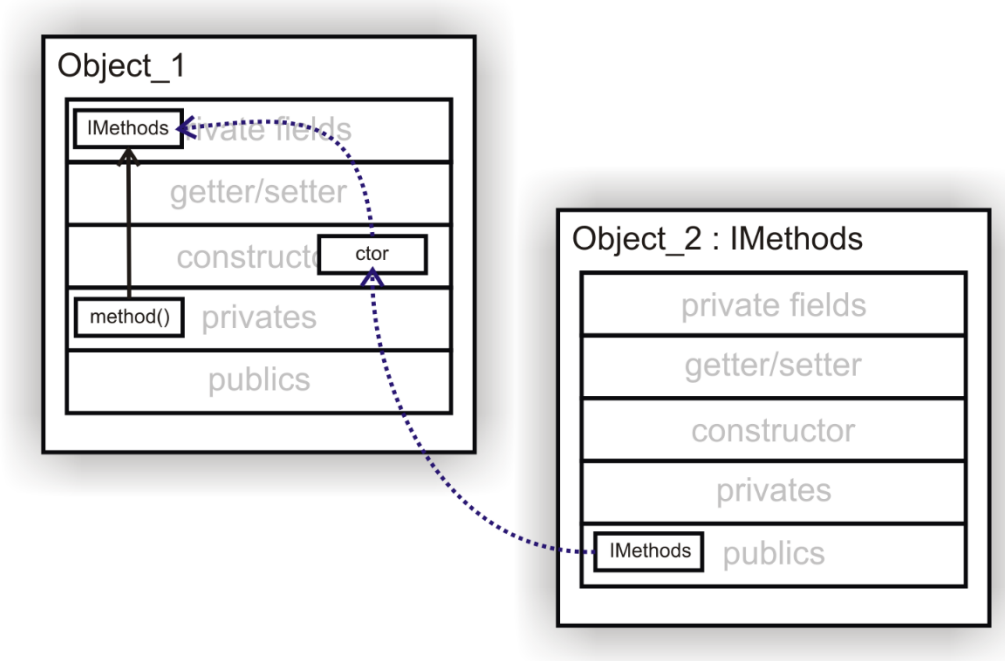

Abbildung 5: Dependency Injection
(eigene Darstellung)

Das Erstellen und Übergeben der Objekte Object_1 und Object_2 übernimmt dabei ein eigens dafür erstelltes Injector-Objekt. Meistens ist das Injector-Objekt Teil eines Dependency Injection Frameworks. Mittels Konfigurationsdateien wird festgelegt welches Objekt erstellt wird und zu welchem Objekt es abhängig wird.

In C-Objekten wird die Dependency-Injection mittels „interface injection“ realisiert. Dabei werden der Init()-Funktion des Object_1.c die Adressen der Funktionen des Object_2.c übergeben. Diese Funktionsadressen werden in private, statische Funktionszeiger des Object_1.c übergeben (siehe Abbildung 6).

Der Injector ist in diesem Fall kein eigenes C-Objekt, sondern die Haupt-Init-Funktion, in der auch die Init-Funktion des Object1.c aufgerufen wird.

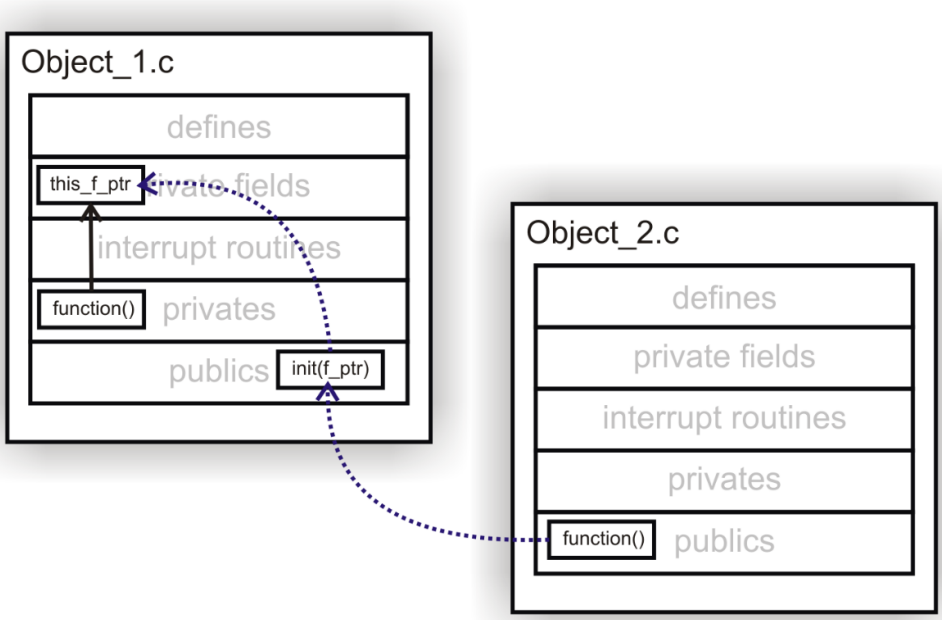

Abbildung 6: Übergabe von Schnittstellen (eigene Darstellung)

Die Übergabe von Schnittstellen eines Objekts zu einem anderen wird mittels C-Funktions-Zeiger realisiert.

Der Init()-Funktion des Object_1 werden die öffentliche-externen Funktionen des Object_2 übergeben. Diese werden in statisch-private Funktionszeiger übergeben. Sämtliche Funktionen des Object_1 verwenden nur die statisch-privaten Funktions-Zeiger um mit dem Object_2 zu kommunizieren.

Mit dieser Herangehensweise wird eine lose Kopplung der Objekte untereinander erzielt. Der Quellcode des Object_1 muss nicht verändert werden, wenn das Object_2 durch ein anderes Objekt ausgetauscht wird.

3.2.3.3 C-Quellcode-Datei eines Objekts

```
/*
C-Object-Template

Author: Ing. Ehrengruber Manfred
Date: 2016-08-03
*/

//============================================================================
//defines
#define WAIT_TIME       3000 //[msec]

//============================================================================
//private fields
static unsigned char this_is_intialized = FALSE;
static unsigned short this_timeout_timer = 0;
//Programm-Pointer, der je nach Kommunikation (USB,CAN,UART,Modem) auf die Schnittstelle
//initialisert werden.
static unsigned char(*this_char_rdy)(void) = NULL;
static unsigned char(*this_get_char)(void) = NULL;

//============================================================================
//interrupt routines

//----------------------------------------------------------------------------
//Timer-Interrupt-function which is called from the timer-object-isr-function every 1msec
void operate_1msec_timer()
{
        if (this_timeout_timer > 0)
                this_timeout_timer--;
}

//============================================================================
//privates

//----------------------------------------------------------------------------
//waits the wait time
static void wait(void)
{
        this_timeout_timer = WAIT_TIME;

        while (this_timeout_timer > 0);
}

//============================================================================
//publics

//----------------------------------------------------------------------------
//waits and returns a received byte, if available, otherwise -1 will be returned
short get_byte(void)
{
        if (!this_is_intialized)
                return -1;
        wait();
        if (this_char_rdy())
                return this_get_char();
        else
                return -1;
}

//----------------------------------------------------------------------------
//initializes this object
int init(unsigned char(*char_rdy)(void), unsigned char(*get_char)(void))
{
        this_char_rdy = char_rdy;
        this_get_char = get_char;
        this_is_intialized = TRUE;
}
```

Abbildung 7: Beispiel-C-Quellcode eines Objekts (eigene Darstellung)

3.2.3.4 Mit der Herangehensweise erfüllte objektorientierte Prinzipien

Folgend werden für dieses Konzept die Prinzipien für den objektorientierten Entwurf nach (Goll, 2014) begutachtet:

Weiterentwickelbarkeit und Verständlichkeit

Die Weiterentwickelbarkeit ist durch die Kapselung der Teilbereiche des gesamten Programmcodes gegeben. Es können Objekte für sich, unabhängig vom Rest des Programms weiterentwickelt werden.
Die Verständlichkeit wird durch das Abstrahieren von realen Objekten in virtuelle Objekte erreicht. Funktionell erfährt das gesamte Programm dadurch eine verständliche Auftrennung seiner Teilbereiche.
Somit ist die Lesbarkeit und Verständlichkeit gegeben.

Kapselung und Information Hiding

Die Kapselung und Information Hiding im Sinne eines „Objekts“ ist gegeben:
Methoden und Daten verschmelzen bei der Kapselung zu einem Objekt. Daten und Methoden befinden sich also zusammen in einer Kapsel vom Typ einer Klasse. (Goll, 2014)
Durch die öffentlichen und statisch-privaten Funktionen im C-Modul und den Einsatz statischer-privater Variablen kann Kapselung und Information Hiding realisiert werden.

Abstraktion

Die Abstraktion ist durch feste, öffentlich-extern-deklarierte Funktionen, die objektorientiert als Schnittstelle angesehen werden können, gegeben. Das Verhalten des Objekts ist somit mittels seiner Schnittstelle implementiert.

Single Responsibility-Prinzip

Jedes Objekt wurde für eine spezielle, getrennte Aufgabe erstellt. In dem obigen Beispiel gibt es Objekte für die UART-Peripherie-Bedienung, für die Timer-Peripherie-Bedienung und ein Objekt für die Kartenleser-Bedienung.

Lose Kopplungen

Die lose Kopplung von Objekten ist dadurch gegeben, da diese Objekte untereinander in schwacher Wechselwirkung stehen. Die Abhängigkeiten der Objekte untereinander sind auf die Konsum-Funktionalität beschränkt. Im Beispiel (siehe Abbildung 3) ist die Abhängigkeit des Kartenlesers-Objekts vom UART-Objekts lose. Das UART-Objekt kann dadurch durch ein USB-Objekt ausgetauscht werden.

3.3 Herausforderung - Programmablauf

3.3.1 Kontext

Programmiersprache: C, C++, C#
Programmierparadigma: imperative und objektorientierte Programmierung

Der Programmablauf bildet das Herzstück jedes eingebetteten Systems. Er ist Teil der Applikationssoftware und definiert das Verhalten des Gerätes. Es gibt bewährte Technologie wie Tasks, Prozesse, State-Machines für die Realisierung eines Programmablaufes.

Während bei kleinen eingebetteten Systemen der Programmablauf in der Hauptschleife untergebracht werden kann, muss bei mittleren und großen eingebetteten Systemen eine Separierung und Kapselung des Programmablauf-Quellcodes realisiert werden.

3.3.2 Herausforderungen

Der Programmablauf wird für ein bestimmtes Gerät und für eine bestimmte Anwendung konzipiert. Die Herausforderung besteht darin, dass der Programmablauf zum einen mit den Betriebsmitteln des eingebetteten Systems korreliert, zum anderen flexibel, austauschbar und erweiterbar bleibt. Betriebsmittel sind Peripheriekomponenten des Mikrocontroller-Systems oder externe Komponenten wie Drucker, Tastatur und Display.

Abbildung 8 veranschaulicht ein eingebettetes System welches mit den Eingabegeräten Kartenleser, Tastatur und den Ausgabegeräten Drucker und Display ausgestattet ist.

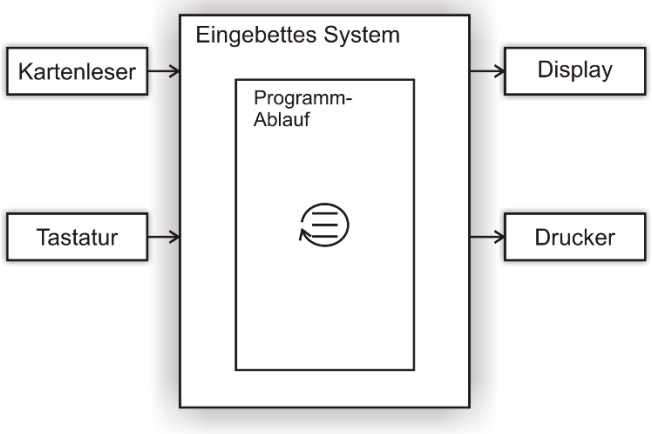

Abbildung 8: Programmablauf
(eigene Darstellung)

Anhand diesen Fallbeispiels werden Vorgehensweisen und Richtlinien für das Konzeptionieren eines Programmablaufes vorgestellt.

3.3.3 Lösungen

Der Programmablauf besteht immer aus einer Schleife (main loop). In dieser Schleife werden die Instruktionen für das Verhalten des eingebetteten Systems ausgeführt. Kleine eingebettete Systeme können mit nur einem Hauptprogrammablauf ihr Verhalten abbilden. Bei größeren eingebetteten Systemen ist der Programmablauf komplexer. Folgend wird auf die Möglichkeiten, Programmabläufe gekapselt, weiterentwickelbar und stabil zu konzipieren, eingegangen.

Grundsätzlich können Programmabläufe mit folgenden Technologien, auf die nicht näher eingegangen wird, realisiert werden:

- Single-Tasking-System
- Multi-Tasking-System
- Automaten (State-Machines)

Die folgenden Herangehensweisen können, müssen aber nicht mit Hilfe dieser Technologien realisiert werden.

3.3.3.1 Separation der Programmabläufe

Der Programmablauf wird in mehrere Teil-Programmabläufe aufgetrennt, die zeitlich gemeinsam oder getrennt voneinander ausgeführt werden (siehe Abbildung 9). Somit wird eine funktionelle Kapselung erzielt. Der Programm-Code der einzelnen Programmabläufe ist damit übersichtlicher und erweiterbarer. Außerdem können die partiellen Programmabläufe ausgetauscht oder bei Systemen mit kleinem primären Programmspeicher in einen sekundären Nebenspeicher ausgelagert werden.

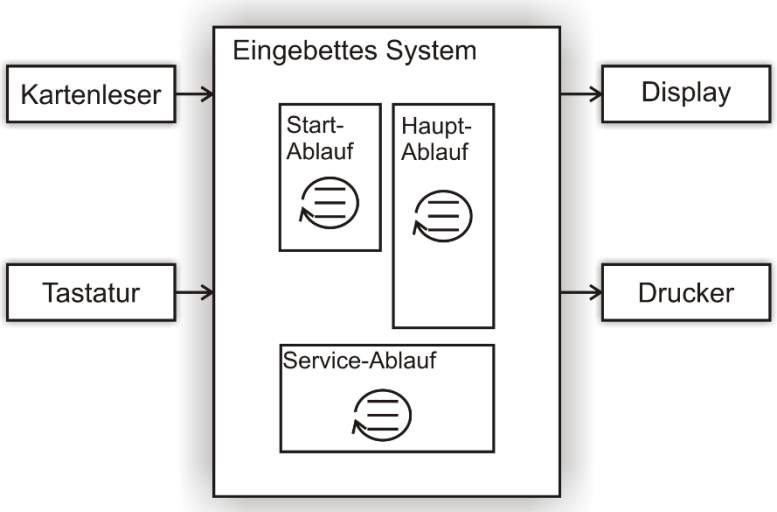

Abbildung 9: Programmablauf mit Separation (eigene Darstellung)

Für die Auslagerung von Programmcode in einem sekundären Nebenspeicher wurden bereits Muster erstellt. Im Kapitel „Major Technique: Secondary Storage“ im Buch „Small Memory Software“ (Noble, 2001) sind diese definiert. Mit Hilfe der Herangehensweise „Separation der Programmabläufe“ können diese Muster in die Praxis umgesetzt werden.

Im konkreten Beispiel (siehe Abbildung 9) existieren drei Programmabläufe: Start- und Hauptablauf werden parallel in einem Multi-Tasking-System ausgeführt. Der „Haupt-Ablauf“ übernimmt Monitor-Tätigkeiten während der „Start-Ablauf“ das Verhalten beim Start des eingebetteten Systems wiederspiegelt. Der dritte Programmablauf ist der Service-Ablauf, wo externe Komponenten (Betriebsmittel) wie Kartenleser, Display, Tastatur und Drucker auf Funktion getestet werden können. Ist dieser aktiv, werden der „Start-Ablauf“ und der „Haupt-Ablauf“ beendet.

Wird im eingebetteten System ein universelles Betriebssystem eingesetzt, welches eine Prozessverwaltung implementiert, werden die Programmabläufe Prozessen zugeordnet. Im konkreten Beispiel (siehe Abbildung 9) werden die Programmabläufe „Start-Ablauf“ und „Haupt-Ablauf“ durch jeweils separate Prozesse ausgeführt. Bei der Erzeugung des Prozesses, der den „Service-Ablauf“ ausführt, werden die Prozesse der Programmabläufe „Start-Ablauf“ und „Haupt-Ablauf“ beendet.

3.3.3.2 Task-Sicherheit bei Nebenläufigkeit

Es besteht die Gefahr, dass externe Komponenten (Betriebsmittel) des eingebetteten Systems durch die Separation der Programmabläufe parallel von mehreren Programmabläufen angesprochen werden. Dies hat zur Folge, dass Daten, die von den externen Komponenten ausgelesen werden an andere Programmabläufe verloren gehen. Die Lösung für diese Nebenläufigkeits-Problematik ist die Einführung eines entsprechenden Betriebsmittel-Ablaufs. Nur dieser Ablauf hat das Recht mit dem Betriebsmittel direkt zu kommunizieren. Die weitere Kommunikation der Verhaltens-Programm-Abläufe erfolgt mittels spezialisierten Pipes.

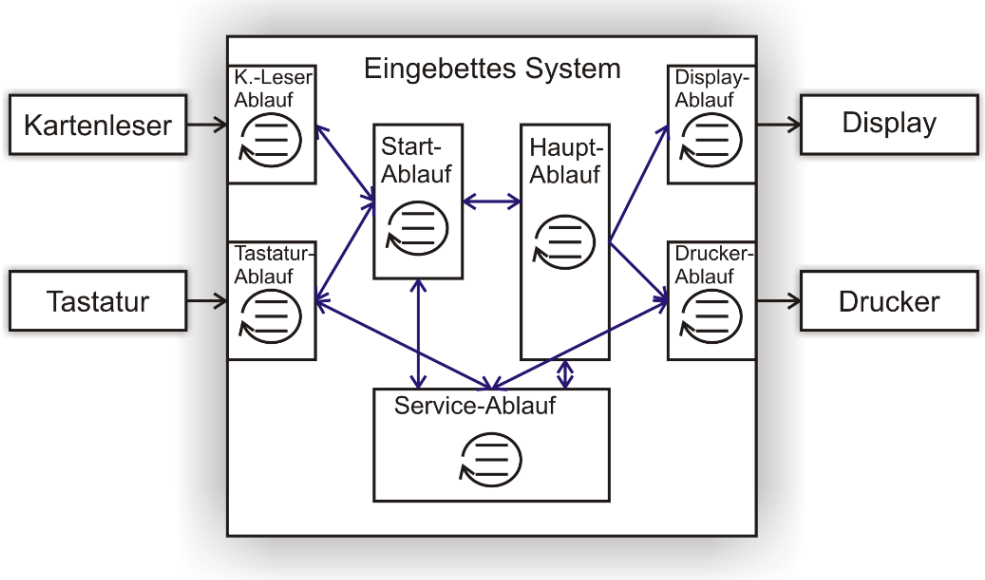

Abbildung 10: Betriebsmittelabläufe mit Pipes (eigene Darstellung)

Jedes externe Gerät (Betriebsmittel) besitzt einen eigenen Programmablauf. Diese Programmabläufe können in einem Multi-Tasking-System mittels Tasks, in einem Single-Task-System durch Funktionen, die vom Hauptablauf sequentiell aufgerufen werden, realisiert werden. Die blauen Linien stellen die spezialisierten Pipes dar.

Die Aufgabe der Betriebsmittelabläufe besteht darin, Status-Informationen der externen Geräte zyklisch auszulesen, für andere Programmabläufe bereitzustellen und Befehle, die von den anderen Programmabläufen gesendet werden abzuarbeiten.

3.3.3.3 Spezialisierte Pipes für die Inter-Programmablauf-Kommunikation

Werden mehrere Programmabläufe in einem eingebetteten System verwendet, ist eine Technologie gefordert, um Informationen oder Instruktionen zwischen den Programmabläufen austauschen zu können. Dies wird mittels speziell definierten Pipes realisiert (siehe Abbildung 11).

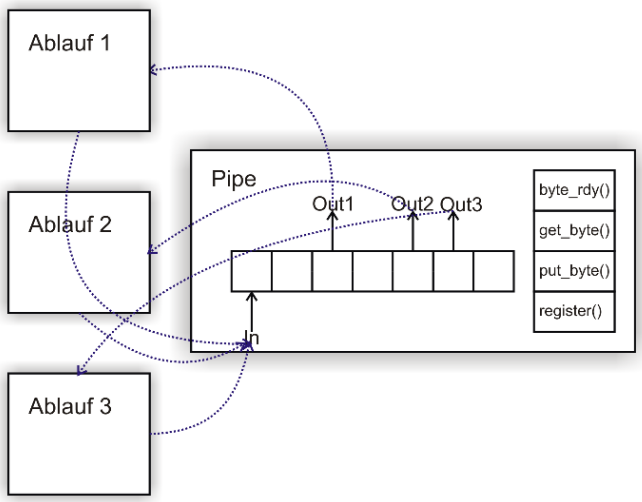

Abbildung 11: Pipe für Inter-Programmablauf-Kommunikation (eigene Darstellung)

Jeder Ablauf muss sich zu Beginn mit der register()-Funktion anmelden. Bei dieser Anmeldung wird dem Ablauf ein Out-Pointer zugewiesen.

Jeder Ablauf hat das Recht mit der Funktion put_byte() Bytes in die Pipe zu schieben. Werden die Pipes in einem Betriebssystem eingesetzt, muss die Funktion put_byte() mittels wechselseitigem Ausschluss (Mutex) geschützt werden. Dazu werden die Mutex-Funktionen mittels C-Makros definiert und je nach Plattform an das Betriebssystem portiert.
Jeder Ablauf hat seinen eigenen Out-Pointer. Mit den Funktionen byte_rdy() und get_byte() können die Bytes von der Pipe ausgelesen werden.

Mit dieser Technologie können Daten von einem Ablauf an mehrere andere Abläufe verteilt werden, ohne dass Informationen verloren gehen. Eine Tasten-Betätigung im dargestellten eingebetteten System (siehe Abbildung 10) wird im Tastatur-Ablauf behandelt und in die Pipe geschoben. Alle anderen Abläufe bekommen damit die Information des Tastendrucks durch die Pipe mitgeteilt.

3.3.3.4 Beispiel-Quellcode

Der Beispiel-Quellcode ist im Anlagenbereich untergebracht.

3.3.3.5 Zusammenfassung

Mit Hilfe dieses Konzeptes können Programme strukturiert implementiert werden, unabhängig vom eingesetzten Betriebssystem, Hardware und Programmiersprache. Eine Portierung bestehender Software in eine neue Umgebung, beispielsweise bei Einführung eines universellen Betriebssystems, ist somit einfach zu bewerkstelligen.

3.4 Herausforderung - Konfigurationssystem

3.4.1 Kontext

Programmiersprache: C, C++, C#
Programmierparadigma: imperative und objektorientierte Programmierung

Fast jedes eingebettete System soll dem Benutzer die Möglichkeit bieten, Einstellungen individuell vornehmen zu können. Somit können das Verhalten und der Funktionsumfang des eingebetteten Systems durch die Kunden angepasst werden.

Typische Konfigurationsparameter sind:

- Timing-Parameter
  Ändern des Programmablaufs, Timeouts
- Befehls-Freischalt-Parameter
  Verhalten des Gerätes individuell anpassbar gestalten, damit das Gerät in mehreren Kostenvarianten vertrieben werden kann.
- Peripherie-Parameter
  Individuelle externe Beschaltung des Gerätes konfigurieren. (z.B.: verwendeter Drucker-Typ)

Realisiert ist ein Konfigurationssystem aus:

- … einem Datenbereich, wo die Parameter-Werte persistent abgelegt werden.
- … einem Parameter-System, wo Parameter implementiert und deren Attribute (Default-Wert, Min-/Max-Grenzen, Zugriffsberechtigung) definiert sind.
- … Schnittstellen zur Manipulation der Parameter-Werte (z.B.: mittels Benutzerschnittstelle und/oder Kommunikationsschnittstelle)

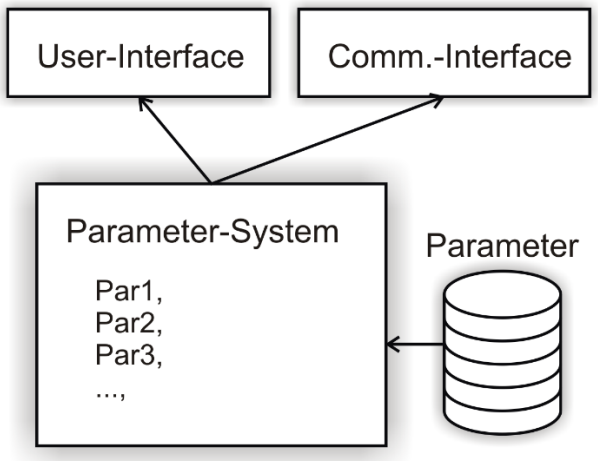

Abbildung 12: Konfigurationssystem (eigene Darstellung)

Ein Konfigurationssystem für eingebettete Software steht folgenden Herausforderungen gegenüber:

- Das Anlegen neuer Parameter soll schnell, einfach und intuitiv vom Programmierer ausgeführt werden können, ohne dazu externe Software-Tools benutzen zu müssen.
- Das Konfigurationssystem soll Plattform-unabhängig und dadurch im hohen Grade wiederverwendbar sein.
- Das Konfigurationssystem soll Schnittstellen zur Benutzer- und Kommunikationsschnittstelle für die Manipulation der Parameterwerte zur Verfügung stellen.
- Das Konfigurationssystem soll Multilingualität unterstützen. Die Parameter-Bezeichnungen sollen mehrsprachig ausgeführt werden können.
- Das Konfigurationssystem soll es ermöglichen, dass Parameter gruppiert und baumartig strukturiert werden können.
- Ein performanter Zugriff auf die Parameter- Werte und Attribute soll realisiert sein.
- Das Konfigurationssystem soll für den Einsatz in Quellcode-Verwaltungssystemen optimiert sein: Anlegen mehrerer Parameter von mehreren Entwicklern in verschiedenen Entwicklungs-Zweigen sollen einfach zu bewerkstelligen sein.

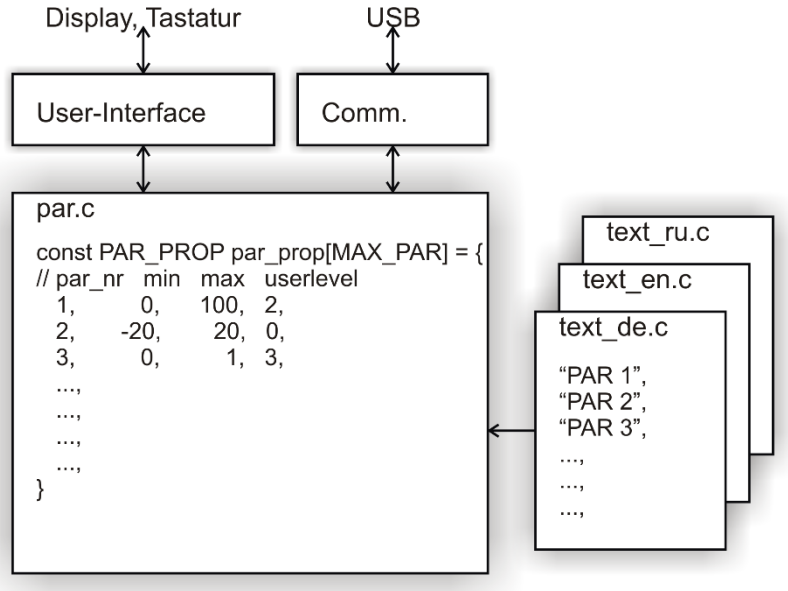

Abbildung 13: Herkömmliches Konfigurationssystem (eigene Darstellung)

Eine gängige Vorgehensweise ist das Deklarieren einer konstanten Parameterdefinition (Ein Array von Strukturen), wo für jeden Parameter deren Attribute wie Minimalwert, Maximalwert, Zugriffslevel, … definiert werden können.

Die Parameterbezeichnungen werden meistens in eigenen Text-Quellcode-Dateien als konstante Strings abgelegt. Für jede Sprache existiert ein Text-File. Somit kann Mehrsprachigkeit realisiert werden.

Die Konfiguration wird entweder direkt am Gerät mittels User-Interface oder über Kommunikationsschnittstellen (USB, UART, Ethernet, …) mit Einsatz eines Service-Programms am PC vorgenommen.

Dieses herkömmliche Konfigurationssystem birgt folgende Nachteile in sich:

- Es können Parameter nur in einer Ebene angelegt werden, eine baumartige Strukturierung und Gruppierung ist damit nicht möglich.
- Es ist ein hoher Aufwand des Programmierers nötig, einen Parameter neu anzulegen: Pro Parameter müssen Definitionen für Parameter- und Text-Nummer erstellt werden.
- Der Zugriff von der Benutzer- / Kommunikationsschnittstelle muss vom Programmierer für die Parameter individuell ausprogrammiert werden.
- Beim gleichzeitigen Anlegen neuer Parameter in unterschiedlichen Entwicklungszweigen kommt es beim anschließenden Zusammenführen (Merge) zu Parameterkonflikten:
  Ein Parameterkonflikt kann auftreten, wenn zwei Programmierer zur selben Zeit jeweils einen Parameter anlegen.

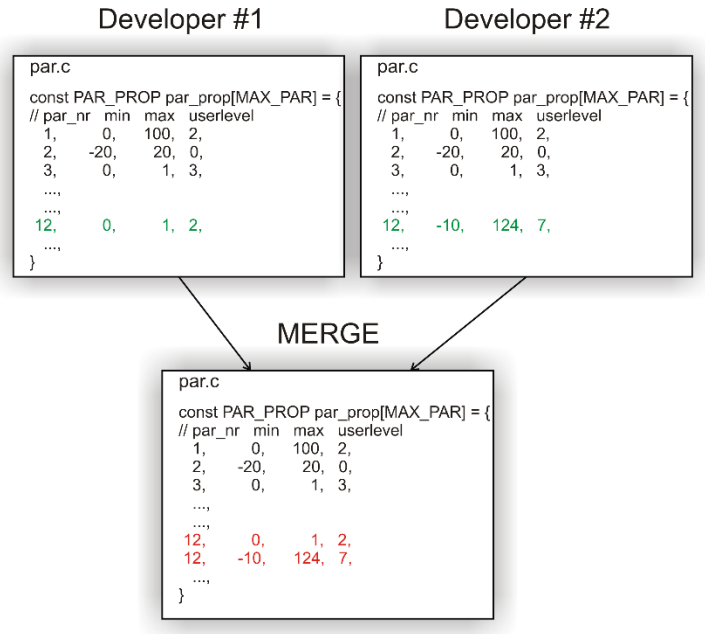

Abbildung 14: Parameter-Konflikt beim Merge (eigene Darstellung)

Beide Programmierer legen einen neuen Parameter unterschiedlicher Bedeutung mit der Nummer 12 an. Beim Quellcode-Verwaltungs-Merge der Quellcode-Dateien kommt es dann zu einem Konflikt. Einer der Programmierer muss seinen Parameter und deren Parameternummer im Nachhinein ändern.

3.4.2 Lösungen

Abbildung 15 stellt ein Konzept eines Konfigurationssystems, welches den diskutierten Anforderungen entspricht, dar:

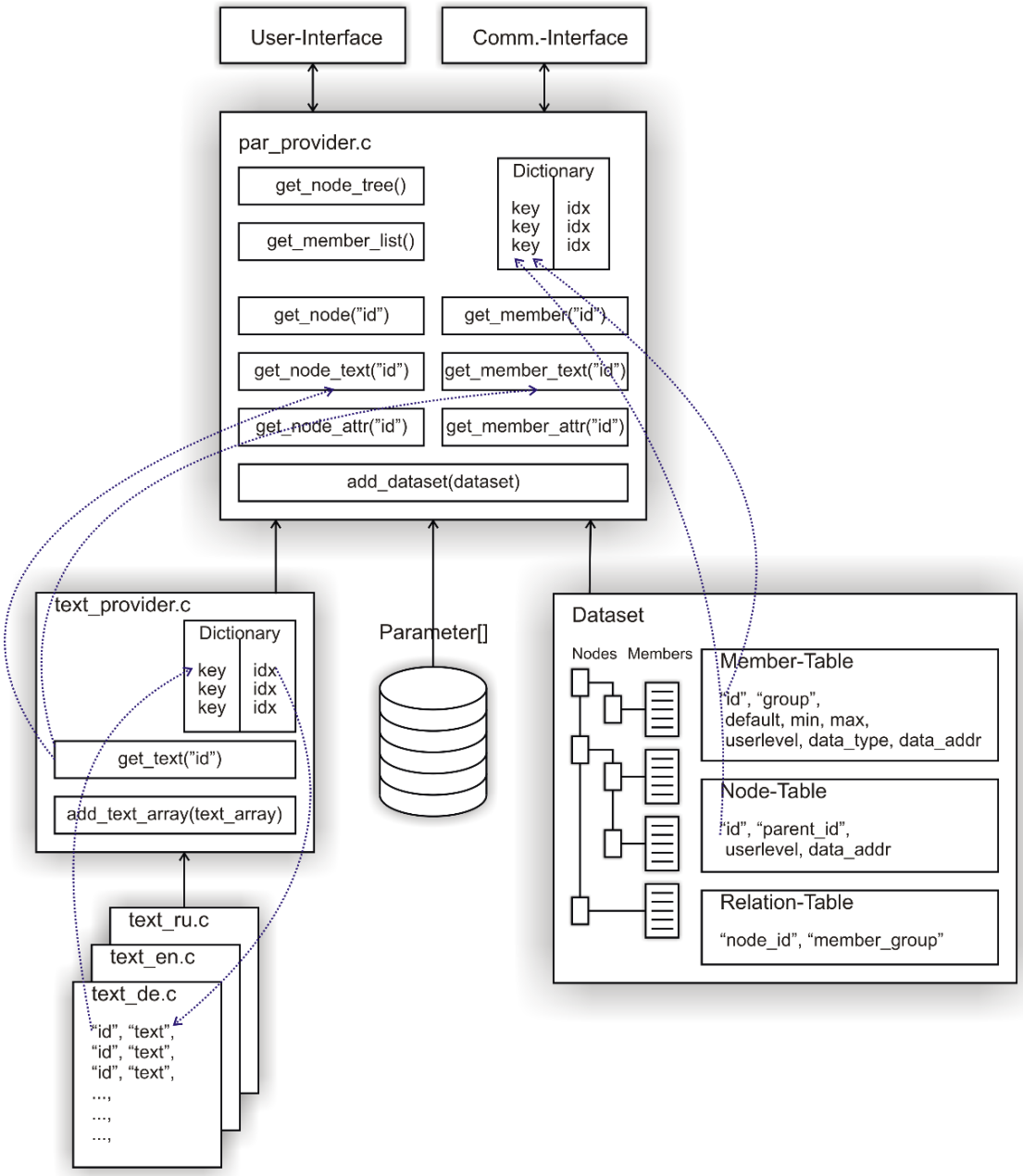

Abbildung 15: Konfigurationssystem (eigene Darstellung)

Dieses Konfigurationssystem besteht aus folgenden Teilen:

- Parameter-Werte-Speicher
- Parameter-Dataset
- Text-Provider und pro Sprache eine Text-Ressourcen-Datei
- Parameter-Provider
- Dictionaries zur Indizierung der Texte und Parameter

3.4.2.1 Parameter-Werte-Speicher

Der Parameter-Werte-Speicher kann frei deklariert werden. Er kann aus Feldern in Form von Arrays oder aus deklarierten Strukturen bestehen. In der Regel wird ein Array mit fixem Datentyp (unsigned long, float oder double) als Parameter-Werte-Speicher verwendet.

Dieses Parameter-Werte-Array wird in den Flashspeicher persistiert. Ausgelöst wird das Sichern der Parameter in den Flashspeicher mit einem „Save“-Request, der von der Benutzer- bzw. Kommunikations- Schnittstelle aufgerufen wird.

Beim Boot der Firmware werden die Parameter-Werte in den Parameter-Werte-Speicher geladen.

3.4.2.2 Parameter-Dataset

Das Parameter-Dataset beinhaltet die Attribute des Parameters. Soll ein neuer Parameter angelegt werden, so wird im Dataset der Parameter-Tabelle (Member-Tabelle) eine neue Datenzeile angefügt. In dieser Datenzeile werden die Attribute des Parameters, wie Standard-Wert, Minimal-/Maximal-Wert und Zugriffberechtigungen definiert.

Die Identifikation wird im Gegensatz zum herkömmlichen Konfigurationssystem nicht mit Parameternummern, sondern mit Schlüsselwörter vorgenommen. Der Programmierer wählt beim Neu-Anlegen eines Parameters einen aussagekräftigen Namen als Schlüsselwort. Mit diesem Namen kann der Parameter später im Programm adressiert werden. Der Vorteil dieser Methodik ist neben dem effizienten Anlegen von Parametern (es muss keine Parameternummer definiert werden), die verminderte Wahrscheinlichkeit des Auftretens eines Parameter-Konflikts beim parallelen Anlegen von Parametern.

Am Ende der Parameter-Attributs-Datenzeile wird die Daten-Adresse des Parameters definiert. Die Daten-Adresse kann auf ein beliebiges Feld im Speicher zeigen. Meistens ist die Daten-Adresse relativ auf einen Eintrag im Parameter-Werte-Speicher gerichtet.

```
//Parameter-Member-Attributs-Entität
typedef struct
{
        char *id;                       // Member-Id: eindeutiges Schlüsselwort
                                        // zur Identifikation des Parameters
        char *group_id;                 // Gruppen-Id: eindeutiges Schlüsselwort
                                                // zur Bildung von Member-Gruppen

        signed long long default_value; // Defaultwert
        signed long long min_value;     // Minimalwert
        signed long long max_value;     // Maximalwert
        unsigned short user_level;      // User Level
        PAR_DATA_TYPE data_type;        // Datentyp der Daten
        unsigned long data_addr;        // Relative Adresse der Par-Daten
}PAR_MEMBER_ENTITY;
```

Abbildung 16: Parameter Member-Attributs-Entität (eigene Darstellung)

Neben der Parameter-Tabelle (Member-Tabelle) beinhaltet das Parameter-Dataset eine Node- und Relation- Tabelle. Mit Hilfe dieser Tabellen ist eine baumartige Strukturierung und Gruppierung von Parametern möglich.

```
//Parameter-Node-Entität
typedef struct
{
        char *id;                       // Node-Id: eindeutiges Schlüsselwort
                                        // zur Identifikation des Knotens
        char *parent_id;                // Node-Id des Vater-Knotens:
                                        // dadurch kann baumartige Struktur
                                        // erstellt werden
        unsigned short user_level;      // User Level
        unsigned long data_addr;        // Relative Adresse der Daten-Member.

}PAR_NODE_ENTITY;

//Parameter-Relation-Entität
typedef struct
{
        char *node_id;                  // Node-Id: eindeutiges Schlüsselwort
                                        // zur Identifikation des Knotens
        char *member_group;             // Gruppen-Id: eindeutiges Schlüsselwort
                                        // von der Gruppe die der Node zugeordnet
                                        // werden soll
}PAR_RELATION_ENTITY;
```

Abbildung 17: Parameter - Node-/Relation- Entität (eigene Darstellung)

Das Parameter-Dataset stellt den Verbund aller Tabellen dar:

```
//Parameter-Dataset
typedef struct
{
        const PAR_MEMBER_ENTITY *member_table;          //Parameter-Tabelle
        const PAR_NODE_ENTITY *node_table;              //Knoten-Tabelle
        const PAR_RELATION_ENTITY *relation_table;      //Relation-Tabelle
}PAR_DATASET;
```

Abbildung 18: Parameter-Dataset (eigene Darstellung)

Der Quellcode in Abbildung 19 stellt ein gesamtes, befülltes Parameter-Dataset dar.

Das Dataset ist konstant deklariert und wird vom Linker in den Flash-Speicher des Mikroprozessors platziert. Mit der Node-Tabelle wird die baumartige Struktur vorgegeben. Mit der Relation-Tabelle kann eine Parameter-Gruppe auf mehreren Knoten abgebildet werden. Die Knoten „General“ und „General2“ benutzen die gleiche Gruppe „GeneralConfig“. Die Adressierung zu den Daten im Parameter-Werte-Speicher erfolgt relativ zu den Knoten. Die Datenadresse des Knotens stellt die Kopfadresse dar, die Datenadresse der Member entspricht den Offset von der Kopfadresse zu den eigentlichen Daten (siehe Abbildung 20).

```
static const PAR_MEMBER_ENTITY par_member_table[] = {

//_____________________________________________________________________________________________________
//      id, group_id, default_value, min_value, max_value, user_level, data_type, data_addr
//
// GENERAL CONFIG.
"MasterTimeout", "TimingsConfig", 10, 1, 100, UL_USER, LONG, ((unsigned long)(&par[0] - &par[0])),
"EnablePrintOut", "GeneralConfig", 0, 0, 1, UL_USER, LONG, ((unsigned long)(&par[1] - &par[0])),

NULL, NULL, 0, 0, 0, UL_GOD, LONG, 0,      //last entry
};

static const PAR_NODE_ENTITY par_node_table[] = {

//_____________________________________________________________________________________________________
//      id, parent_id, user_level, data_addr
//
// GENERAL CONFIG.
"Parameter", NULL, UL_USER,        0,
"General", "Parameter", UL_USER, ((unsigned long)&par[0]),
"General2", "Parameter", UL_USER, ((unsigned long)&par[32]),
"Timings", "Parameter", UL_USER,   ((unsigned long)&par[0]),
NULL, NULL, UL_GOD, 0,    //last entry
};

static const PAR_RELATION_ENTITY par_relation_table[] = {

//_____________________________________________________________________________________________________
//      node_id, member_group
//
"General", "GeneralConfig",
"General2", "GeneralConfig",
"Timings", "TimingsConfig",

NULL, NULL,      //last entry
};


const PAR_DATASET par_dataset = {
par_member_table,
par_node_table,
par_relation_table,
};
```

Abbildung 19: Befülltes Parameter-Dataset (eigene Darstellung)

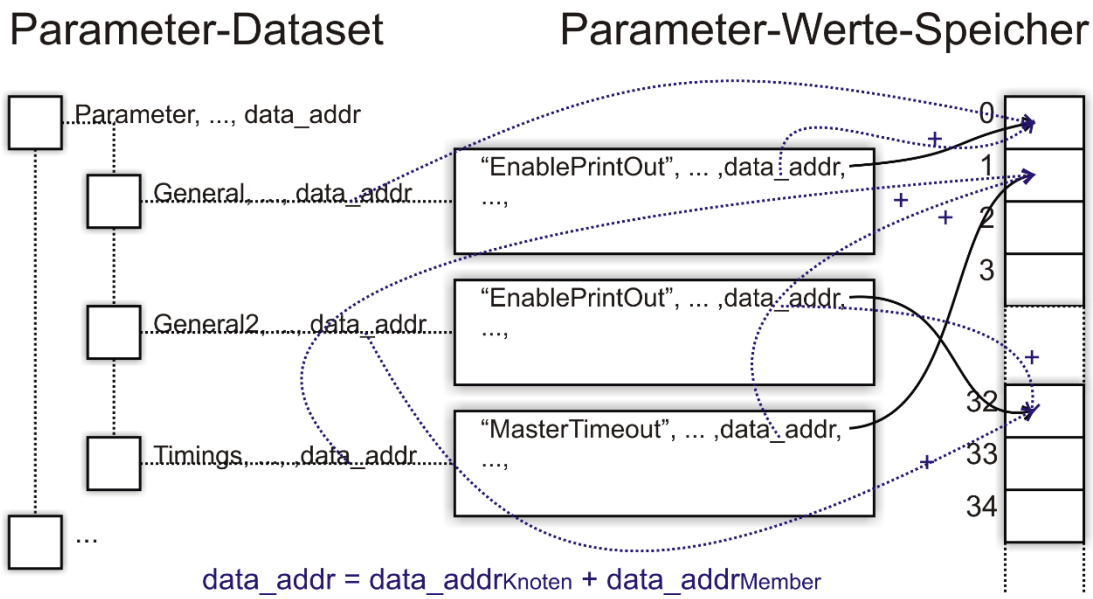

Abbildung 20: Adressierung der Parameter-Daten (eigene Darstellung)

3.4.2.3 Text-Provider

Der Text-Provider bietet eine Schnittstelle an, mit der die Parameter-Bezeichnungen, welche der Benutzer-/Kommunikationsschnittstelle bereitgestellt werden, abgefragt werden können.

Je nach Sprache wird eine Text-Ressourcen-C-Datei angelegt. In diesen Dateien werden die Texte in Kombination mit dem Parameter-Schlüsselwort eingetragen.

Mit der Funktion get_text(char *id) können die entsprechenden Parameter-Bezeichnungen abgefragt werden.

```
static const TEXT_ENTRY this_text_table[] = {
        "MasterTimeout", "Haupt-Ablauf-Timeout [sec]",
        "EnablePrintOut", "Bon-Ausdruck aktiv",
        NULL,   NULL,
};

const TEXT_ENTRY *get_text_de(char *key)
{
        return &this_text_table[0];
}
```

Abbildung 21: Text-Ressourcen-C-Datei für Deutsch (eigene Darstellung)

3.4.2.4 Parameter-Provider

Der Parameter-Provider bildet das Herzstück des Konfigurationssystems. Er konsumiert die Schnittstellen des Text-Providers, Parameter-Werte-Speicher und des Parameter-Datasets und bietet die Schnittstellen für die Benutzer-/Kommunikationsschnittstelle an.

Außerdem werden mit der Parameter-Provider-Schnittstelle die Parameter für das Programm bereitgestellt.

3.4.2.5 Dictionaries zur Indizierung

Für die Identifikation und Adressierung der Parameter und Parameter-Bezeichnungen werden Schlüsselwörter verwendet. Aus Performance-Gründen ist das klassische Iterieren und Vergleichen der Schlüsselwörter der Parameter und deren Bezeichnungen nicht möglich. Die Parameter und Parameter-Bezeichnungen werden mittels Dictionaries indiziert.

Die Dictionaries implementieren eine Hash-Tabelle. Die Schlüsselwörter werden in einen Hash-Code übersetzt. Dieser Hash-Code entspricht dem Index des Dictionary-Eintrags. Im Dictionary-Eintrag ist wiederrum der Index des Parameters in der Parameter-Member-Tabelle bzw. in der Text-Tabelle der Text-Ressourcen-C-Datei abgelegt. Mit dieser Technologie ist ein performanter Zugriff auf die Parameter gewährleistet.

Hashverfahren

„Beim Hashverfahren wird versucht durch Berechnung festzustellen, wo der Datensatz mit bestimmten Schlüsselwort gespeichert ist.“ (P.Widmayer, 2002)

„Eine gute Hashfunktion sollte möglichst leicht und schnell berechenbar sein und die zu speichernden Datensätze möglichst gleichmäßig auf den Speicherbereich verteilen um Adresskollisionen zu vermeiden.“ (P.Widmayer, 2002)

Als Hashverfahren wird in den Dictionaries die Divisionsrestmethode mit separater Verkettung der Überläufer angewendet. Tritt eine Kollision auf, so wird der Index in eine lineare verkettete Liste, welche am berechneten Dictionary-Index hängt, hinzugefügt.

Die Berechnung des Indexes im Dictionary aufgrund des Schlüsselwortes erfolgt mit dem Bernstein-Algorithmus:

```
//-----------------------------------------------------------------
//Berechne Index in Dictionary-Hash-Tabelle wo übergebener String
//zugeordnet werden kann.
//[str]... String von dem Hash-Tabellen-Index berechnet werden soll.
unsigned long calculate_hash_table_idx(char *str)
{
        unsigned long hashval = 0;
        for (hashval = 0; *str != '\0'; str++)
                hashval = *str + 33 * hashval;
        return hashval % HASH_TABLE_LENGTH;
}
```

Abbildung 22: Bernstein-Algorithmus für die Index-Berechnung im Dictionary (eigene Darstellung)

Tritt bei der Berechnung des Indexes eine Kollision auf, da an dem berechneten Index bereits ein anderes Schlüsselwort eingetragen ist, wird der aktuelle Eintrag einer linearen Liste hinzugefügt. Diese wird bei der Suche nach den Schlüsselwörtern durch-iteriert.

Sowohl der Text-Provider als auch der Parameter-Provider haben Dictionaries implementiert. Die Dictionaries werden beim Laden des Parameter-Datasets bzw. der Text-Ressourcen-C-Dateien befüllt.

Die Adressierung und die damit verbundene Suche der Parameter und deren Parameter-Bezeichnungen erfolgt immer mit Hilfe dieser Dictionaries.

Effizienz der Dictionaries

Die Parameter und deren Parameterbezeichnungen werden mit Hilfe der Dictionaries adressiert. Bei jedem Zugriff wird mit Hilfe des Hash-Algorithmus (siehe Abbildung 22) aufgrund des Schlüsselwortes die Position in der Dictionary-internen Hash-Tabelle berechnet. An dieser berechneten Position wird die, dort verwiesene, verkettete Liste durch-iteriert und der Wert-Eintrag passend zu dem Schlüsselwort ermittelt. In diesem Kapitel wird die Effizienz dieses Ablaufes untersucht.

Die Maße der Effizienz sind der, zur Ausführung eines Algorithmus benötigte Speicherplatz und die benötigte Rechenzeit. Für die Darstellung der Effizienz von Algorithmen wird die O-Notation verwendet. Diese Notation ist unabhängig von der verwendeten Hardware oder verwendeten Programmiersprache und beschreibt immer den schlechtesten Fall (worst case) der Ausführungszeit bzw. des Speicherplatzaufwandes, abhängig von der Anzahl der zu verarbeiteten Datenelementen N. Auf die genauen mathematischen Hintergründe der O-Notation wird hier verzichtet, lediglich die Klassifizierung von Algorithmen mittels der O-Notation wird erläutert:

- O(1)
  Die Laufzeit des Algorithmus ist konstant und unabhängig von der Anzahl der, zu verarbeiteten Datenelementen.
  Beispiel: Zugriff auf ein Array
- O(N)
  Die Laufzeit des Algorithmus ist direkt proportional zur Anzahl der, zu verarbeiteten Datenelementen.
  Beispiel: Durchsuchen eines Arrays mittels Iteration.
- $O(N^2)$
  Die Laufzeit des Algorithmus ist direkt proportional zum Quadrat der Anzahl der, zu verarbeiteten Datenelementen.
  Beispiel: Verschachteltes Durchsuchen von Arrays (Schleife in Schleife)

Die Effizienz der Dictionaries wird zum einem durch das Berechnen der Position in der Hash-Tabelle, zum andern durch das Iterieren der, dort angebundenen verketteten Liste

bestimmt.
Die Berechnung der Position in der Hash-Tabelle erfolgt in O(1).
Das Iterieren der Liste erfolgt in O(N), wobei N die mittlere Länge der Listen ist.

Folglich ist eine schnelle Zugriffszeit im Dictionary dann gegeben, wenn die mittlere Länge der Listen klein gehalten werden kann. Dies kann durch das Vergrößern der Hash-Tabelle realisiert werden. Eine größere Hash-Tabelle bedeutet allerdings mehr Speicherplatz-Bedarf.

Um konkrete Werte der Effizienz der Dictionary-Implementierung zu erhalten, wurden der Speicherplatz-Bedarf und die Laufzeit in einem realen eingebetteten System empirisch ermittelt. Dafür verwendet wurde der Mikroprozessor STM32F429ZI vom ST-Micro mit einem Systemtakt von nur 16 MHz. Das Messergebnis ist in den Anlagen zu finden.

3.4.2.6 Zusammenfassung

Dieses Konfigurationssystem eignet sich vorwiegend für eingebettete Systeme ab mittlerer Größe. Es können damit Parameter mit all deren Attributen zentral und baumartig unabhängig von deren Speicheradressen definiert werden. Die Adressierung der Parameter erfolgt mittels Schlüsselwörtern anstatt definierter Parameternummern, somit ist ein Quellcode-Verwaltungs-Merge-Konflikt schnell aufzulösen. Die Parameter-Texte können mehrsprachig definiert werden. Der Zugang zu den Parametern von Kommunikationsschnittstelle und Benutzerschnittstelle ist mittels einheitlichen Schnittstellen möglich.

3.5 Herausforderung - Persistenz-System

3.5.1 Kontext

Programmiersprache: unabhängig
Programmierparadigma: imperative und objektorientierte Programmierung

„Persistenz (von lateinisch persistere „durch, über (eine Zeit) hinweg bleiben“) ist in der Informatik der Begriff, der die Fähigkeit bezeichnet, Daten (oder Objekte) oder logische Verbindungen über lange Zeit (insbesondere über einen Programmabbruch hinaus) bereitzuhalten. Dafür wird ein nichtflüchtiges Speichermedium benötigt; auch das Dateisystem oder eine Datenbank, sowie eine durch Protokolle gesicherte bidirektionale und transaktionsorientierte Datenübertragung können als nichtflüchtiges Medium betrachtet werden.“ (Wikipedia, Persistenz (Informatik), 2016)

In eingebetteten Systemen werden vor allem Konfigurations-/Laufzeit- und Sensor-Daten im nichtflüchtigen Speicher abgelegt. Die gängigsten verwendeten, nichtflüchtigen Speicher sind:

- Flash-Speicher
- MRAM-Speicher
- EEPROM-Speicher
- SD-Karten-Speicher

3.5.2 Herausforderungen

Ziel ist es, ein System zu erschaffen, welches das persistente Ablegen von Daten ermöglicht. Dieses System soll folgende Kriterien erfüllen:

- Das Persistenz-System soll Hardware-unabhängig sein.
  Das System soll von der Hardware-spezifischen Implementierung abstrahiert sein.
- Das Persistenz-System soll für die konsumierenden Komponenten eine simple Schnittstelle zum Speichern und Laden von Daten bieten.
- Das Persistenz-System soll Laufzeitdaten, die dem System mit einer Registrier-Funktion bekannt gegeben werden, beim Ausschalten des eingebetteten Systems in den nichtflüchtigen Speicher sichern.
- Das Persistenz-System soll die Möglichkeit bieten, große Datenmengen, beispielsweise von einem Sensor-Datenlogger-Objekt, entgegenzunehmen und in den nichtflüchtigen Speicher zu sichern.

3.5.3 Lösungen

In diesem Kapitel wird die Herangehensweise an die Konzeptionierung eines Persistenz-Systems diskutiert. Aufgrund der unterschiedlichen Speichertypen und der damit verbundenen unterschiedlichen Grundimplementierungen wird hier kein konkreter Entwurf dargestellt.

Das Blockdiagramm (Abbildung 23) stellt ein Persistenz-System dar, welches die im Kapitel 3.5.2 beschriebenen Erwartungen erfüllt.

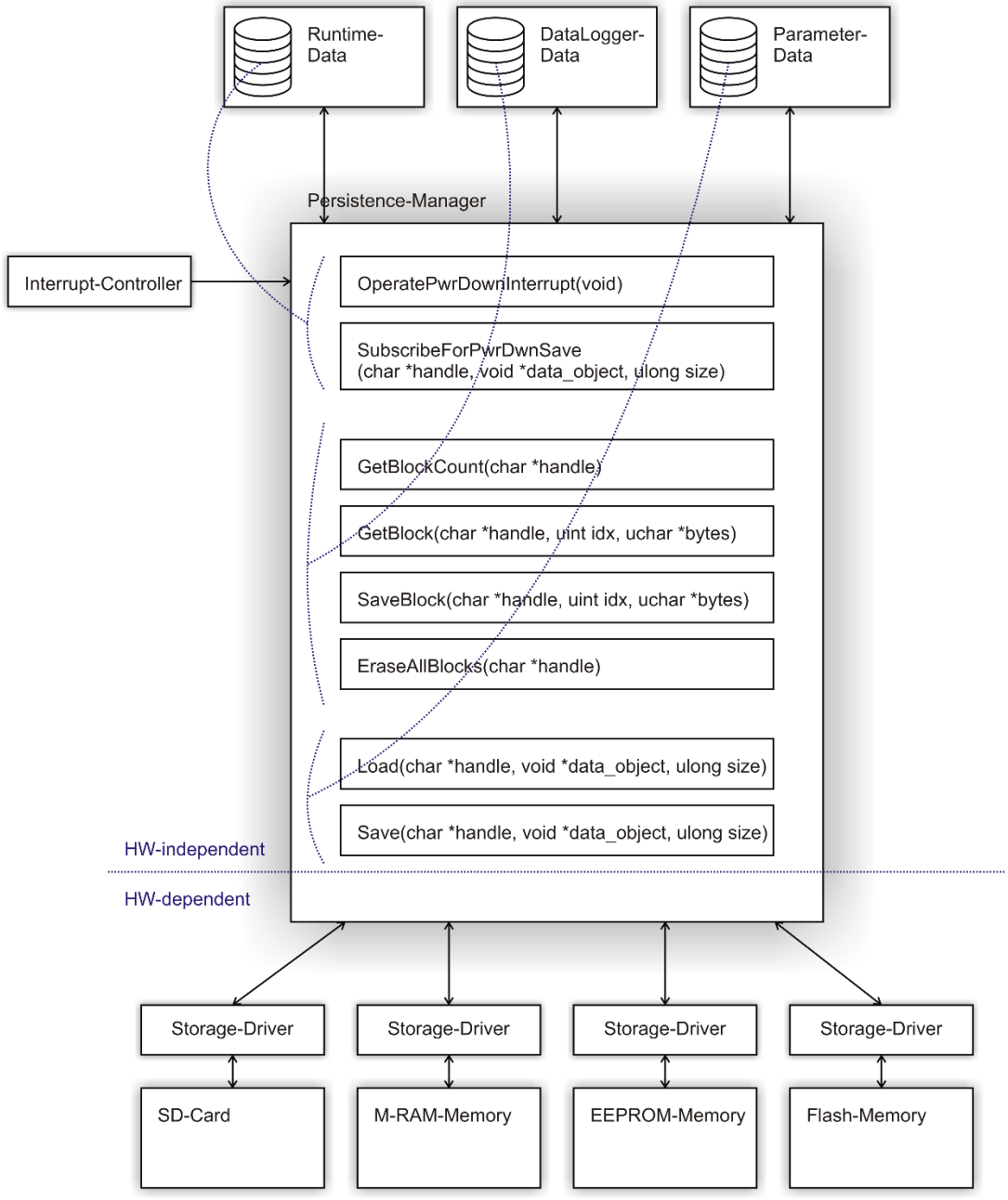

Abbildung 23: Persistenz-System (eigene Darstellung)

Die Kernkomponente stellt der Persistence-Manager dar. Er bietet die Schnittstellen für das Ablegen der Daten in die nichtflüchtigen Speicher an. Der Persistence-Manager besteht aus einem hardwareabhängigen Teil, welcher die konkrete Speicherimplementierung beherbergt und einem hardwareunabhängigen Teil, welcher die Schnittstellen für die konsumierenden Komponenten bereitstellt. Folgend werden die Schnittstellen-Funktionen erläutert:

3.5.3.1 OperatePwrDownInterrupt() / SubscribeForPwrDwnSave()

Mit diesem Funktions-Paar wird das Sichern von Datenobjekten beim Ausschalten (Power-Down) des Gerätes realisiert.

Oft müssen bei eingebetteten Systemen Laufzeitdaten persistent sein. Der Programmablauf beispielsweise muss begonnene Aktionen nach einem Stromausfall beim darauffolgendem Einschalten zu Ende führen können. Auch Laufzeitwerte, wie Betriebsstunden müssen beim Ausschalten im nichtflüchtigen Speicher gespeichert werden können.

Mit der Funktion SubscribeForPwrDwnSave() können Datenobjekte, die im RAM angelegt sind, dem Persistence-Manager bekannt gegeben werden. Beim Power-Down bzw. Stromausfall werden die registrierten Datenobjekte in den nichtflüchtigen Speicher übertragen.

Die Funktion OperatePwrDownInterrupt() wird vom Power-Down-Interrupt-Vektor des Mikroprozessors oder von einem externen I/O-Interrupt aufgerufen, sobald das Ausschalten initiiert wurde. Bevorzugt werden sollte die Variante mit externen I/O-Interrupt, der mit externer Beschaltung früh genug den Einbruch der Versorgungsspannung detektieren kann.

Das Speichern der Datenobjekte erfolgt nach dem „Ping-Pong“-Verfahren: Vor dem Ablegen der Datenobjekte in den nichtflüchtigen Speicher, wird eine Prüfsumme berechnet. Es werden im nichtflüchtigen Speicher zwei Bereiche für ein Datenobjekt reserviert. Diese Datenbereiche werden alternierend zum Speichern der Daten benutzt. Beim Hochfahren des Systems wird das Datenobjekt aus dem gerade aktiven Bereich geladen, der andere, nicht aktive Bereich, wird gelöscht und ist damit bereit beim nächsten Power-Down die zu sichernden Daten entgegenzunehmen. Tritt beim Sichern ein Fehler auf, kann dies beim nächsten Hochfahren festgestellt und eine Fehlerbenachrichtigung generiert werden. In diesem Fall werden die „alten“ Laufzeitdaten vom vorletzten Power-Down benutzt.

3.5.3.2 Load() / Save()

Mit diesen Funktionen werden allgemeine Daten persistiert. Daten, wie die Werte aus dem Parameter-Werte-Speicher können damit im nichtflüchtigen Speicher abgelegt und geladen werden.

Beim Speichern wird der Save()-Funktion die Referenz (Speicher-Adresse) und die Größe des Datenobjektes im flüchtigen Speicher (meistens RAM) übergeben. Zusätzlich wird ein

Schlüsselwort als Handle benutzt, um später die Daten mit der Load()-Methode adressieren zu können.

Beim Laden wird der Load()-Funktion das Schlüsselwort (Handle), die Referenz (Speicher-Adresse) und die Größe des Datenobjektes, wo die Daten übertragen werden sollen, übergeben.

Beim Speichern mittels der Save()-Funktion wird ebenfalls eine Prüfsumme berechnet und mitgespeichert. Diese Prüfsumme wird beim Laden der Daten mittels der Load()-Funktion validiert. Stimmt die berechnete Prüfsumme nicht mit der gespeicherten Prüfsumme überein, wird eine Fehlerbenachrichtigung generiert.

3.5.3.3 Block-Funktionen

Für große, kontinuierliche Datenmengen, beispielsweise beim permanenten Aufnehmen von Sensor-Daten durch ein Datenlogger-Objekt, hat der Persistence-Manager Block-Funktionen implementiert.

Mit diesen Funktionen können beliebig große Byte-Arrays im nichtflüchtigen Speicher abgelegt werden. Mit dem Handle-Argument wird die Zugehörigkeit der Daten festgelegt. Die GetBlockCount()-Funktion retourniert die aktuelle, dem Handle zugeordnete, Anzahl der Blöcke. Mit der GetBlock()-Funktion kann mittels übergebenen Index ein Datenblock ausgelesen werden. Mit der SaveBlock()-Funktion wird ein Byte-Array beliebiger Größe als Datenblock in den nichtflüchtigen Speicher abgelegt.

Die Blöcke sind in linear verketteten Listen im nichtflüchtigen Speicher abgelegt. Für die Datensicherheit/Datenkonsistenz werden pro Datenblock eine Prüfsummenberechnung/Prüfsummenvalidierung durchgeführt.

Mit der EraseAllBlocks()-Funktion werden alle, dem übergebenen Handle zugeordnete Blöcke entfernt und der dafür benutzte Speicher wieder freigegeben.

3.5.3.4 Zusammenfassung

Mit dieser Herangehensweise wird dem Entwickler ein Konzept für das Persistieren der Daten vorgestellt. Mit den darin definierten Schnittstellen-Funktionen kann eine hardwareunabhängige Implementierung des Persistierens realisiert werden. Die konsumierenden Komponenten (Data-Logger, Laufzeit-Daten, Konfigurationssystem) dieses Persistenz-Systems können damit plattformunabhängig eingesetzt werden.

3.6 Herausforderung - Benutzerschnittstelle

3.6.1 Kontext

Programmiersprache: unabhängig
Programmierparadigma: imperative und objektorientierte Programmierung

Fast jedes eingebettete System implementiert eine Benutzerschnittstelle. Sie stellt die Verbindung zwischen Mensch und Gerät dar. Benutzerschnittstellen sind meist bidirektional aufgebaut, die Kommunikation vom Gerät zum Menschen ist optisch und / oder akustisch realisiert, die Kommunikation vom Menschen zum Gerät wird mittels Taster, Touch-Komponenten oder Akustik-Empfänger bewerkstelligt. Die Implementierung einer Benutzerschnittstelle kann von einfachen LEDs und Taster bis hin zur grafischen Bedienoberfläche mittels Touchscreen oder Webserver realisiert werden.

Die Mensch-Computer-Interaktion (MCI, Englisch: human-computer-interaction, HCI) bzw. die Mensch-Maschine-Kommunikation (MMK) ist ein Teilgebiet der Informatik und befasst sich mit der Entwicklung von Benutzerschnittstellen (Englisch: user interfaces). Dabei werden die technischen und die menschlichen Aspekte behandelt. Es werden entsprechend der menschlichen Eingabe-/Ausgabe-Einheiten (Akkustik, Optik, Sensorik, Motorik) die Eingabe-/Ausgabe-Geräte des Computer-Systems konzipiert und entwickelt.

Das Modell der Mensch-Computer-Interaktion wurde von Andreas M. Heinecke im Werk „Mensch-Computer-Interaktion“ (Heinecke, 2012) treffend erläutert. Laut Heinecke wird die Interaktion zwischen Mensch und Computer durch Führen eines Dialogs bewerkstelligt. Dabei werden Aufgaben vom Menschen an den Computer delegiert und Informationen vom Computer zum Menschen transportiert. Selbst ein einzelner Befehl wird im Dialog mit dem Computer getätigt, da Befehle bestätigt werden sollen. (user feedback, z.B.: ein Tastendruck)

3.6.2 Herausforderungen

Benutzerschnittstellen sind von Gerät zu Gerät verschieden zu realisieren. Die Herausforderungen bei der Konzeptionierung einer Benutzerschnittstelle bestehen darin, das Gerät für den Benutzer intuitiv bedienbar zu gestalten. Auch die Mehrsprachigkeit und das Encoding verschiedener Sprachen stellt eine Herausforderung beim Design von Benutzerschnittstellen dar.

3.6.3 Lösungen

Da die Implementierung einer Benutzerschnittstelle von der einfachen LED bis hin zur grafischen Bedienoberfläche mittels HD-Touchscreen reichen kann, werden in diesem Kapitel keine konkreten Entwürfe vorgestellt, vielmehr verstehen sich die folgenden Lösungen als Anhaltspunkte und Referenzen auf Werkzeuge zur Implementierung von Benutzerschnittstellen.

3.6.3.1 Implementierung einfacher Benutzerschnittstellen

Einfache Benutzerschnittstellen bestehen aus Tasten, Tastaturen, LEDs, 7-Segment-Anzeigen, alphanumerische Displays oder monochrom-grafische Displays mit geringer Auflösung. Solche Benutzerschnittstellen werden direkt, nativ ohne Werkzeuge und Grafik-Bibliotheken implementiert:

Abbildung 24 stellt ein vereinfachtes Blockdiagramm für die Implementierung von Benutzerschnittstellen dar.

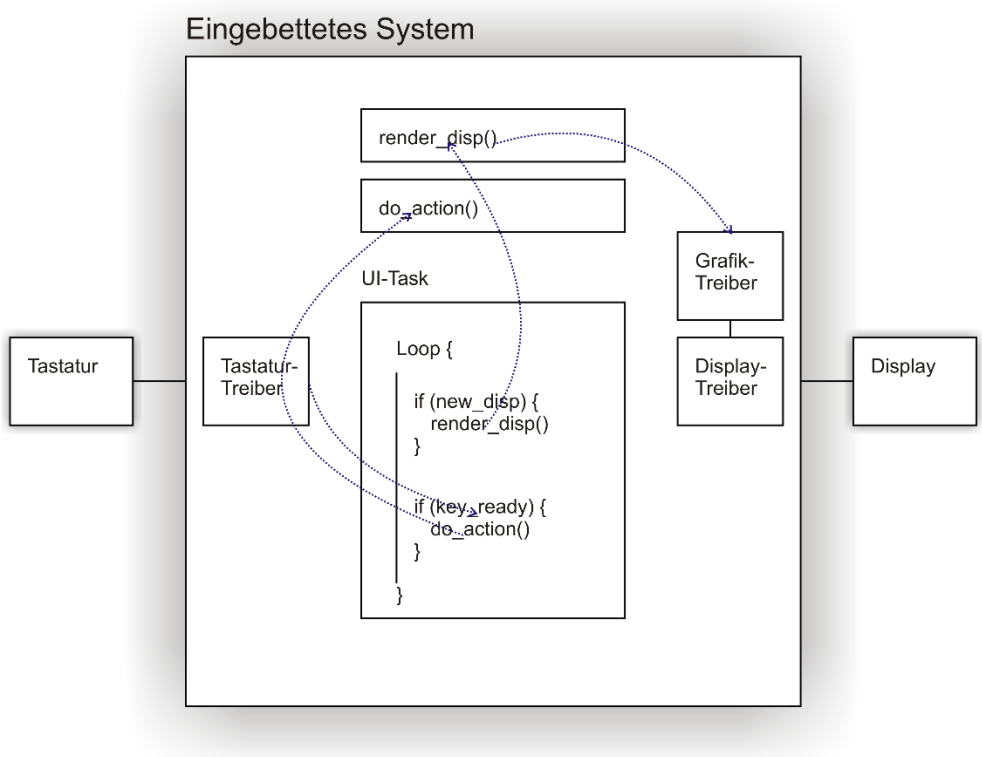

Abbildung 24: Simple Benutzerschnittstellen-Implementierung (eigene Darstellung)

Der Kern dieser simplen Benutzerschnittstellen-Implementierung (siehe Abbildung 24) besteht aus einem UI-Task (User-Interface-Task). Der UI-Task besteht aus einer Schleife in welcher auf eine Tastenbetätigung reagiert wird. Das new_disp-Flag wird immer dann gesetzt, wenn die Anzeige neu gerendert (gezeichnet) werden soll. Sobald das new_disp-Flag gesetzt ist, wird die render_disp-Funktion ausgeführt. Die render_disp()-Funktion

greift auf die Schnittstellen des Grafik-Treibers zu. In diesem sind die Funktionen zum Zeichnen einer Linie, Rechtecks oder Text implementiert. Der Display-Treiber ist verantwortlich für die Umsetzung der Befehle des Grafik-Treibers, somit ist eine Hardware-Abstrahierung gewährleistet.

Abbildung 24 ist eine vereinfachte Darstellung dieses Konzeptes. In der Regel wird die Schleife in mehreren Unterfunktionen realisiert, für jede Ansicht (Maske) wird eine separate Unterfunktion erstellt, die vom UI-Task aufgerufen wird. Somit lassen sich Bedien-Menüs realisieren.

Grundsätzlich kann dieses Konzept auch bei komplexeren grafischen Bedienoberflächen, unter Verwendung von hochauflösenden Farb-Grafik-Displays, zum Einsatz kommen. Die Grafik-/Display-Treiber-Schicht wird allerdings durch existierende Frameworks und Grafik-Bibliotheken ersetzt.

3.6.3.2 Implementierung von grafischen Benutzerschnittstellen (GUI (graphical user interface))

Das manuelle Ausprogrammieren moderner, hochauflösender Grafiken ist aufgrund des hohen Aufwandes nicht sinnvoll. Vielmehr werden für die Generierung komplexer grafischer Bedienoberflächen Bibliotheken und Frameworks benutzt. Dieses Kapitel stellt eine Übersicht gängiger GUI-Bibliotheken und -Frameworks dar.

Crank Storyboard Suite mit Embedded Engine

Website: http://cranksoftware.com/ (23.09.2016)

Crank-Software bietet mit dem Crank Storyboard eine Entwicklungsumgebung für das Erstellen von grafischen Benutzerschnittstellen an. Es werden neben allen gängigen PC-/MAC-Plattformen auch eingebettete Systeme unterstützt, somit ist Crank Storyboard universell einsetzbar.

EasyGUI

Website: http://www.easygui.com/ (23.09.2016)

EasyGUI von ibissolutions ist ein vollständiges Framework zur Erstellung von grafischen Benutzeroberflächen für eingebettete Systeme. EasyGUI unterstützt alle gängigen Display-Controller, C-Compiler und Mikroprozessoren. Mehrsprachigkeit mit optionalen Unicode-Encoding wird ebenfalls unterstützt.

µGFX

Website: http://ugfx.io/ (23.09.2016)

µGFX ist eine Grafik-Bibliothek, die speziell für kleine eingebettete Systeme konzipiert worden ist. Die Bibliothek ist in C quelloffen verfasst. Mit Hilfe des µGFX-Studio-Programms, welches in Windows, Linux und MAC OS lauffähig ist, kann per Drag&Drop die grafische Darstellung erstellt werden.

Embedded Wizard

Website: http://www.embedded-wizard.de/ (23.09.2016)

Embedded Wizard ist ein eigenständiges GUI-System für Plattform-unabhängige Benutzerschnittstellen-Entwicklung. Mit eigener Programmiersprache und IDE kann das GUI auch für kleine eingebettete Systeme erstellt werden.

WPF mittels Microsoft .Net Micro Framework

Website: http://www.netmf.com/ (23.09.2016)

Microsoft hat mit dem .Net Micro Framework ein .Net Framework für eingebettete Systeme entwickelt. Prinzipiell ist das .Net Micro-Framework eine verminderte Version des .Net-Frameworks für die PC-Applikations-Entwicklung. Es kann direkt „stand-alone“ am Mikroprozessor als Betriebssystem eingesetzt werden. Auch auf bestehenden Betriebssystemen (z.B.: FreeRTOS) kann das .Net-Micro-Framework aufgesetzt werden. Die Programmiersprache ist C#, Visual Studio wird als IDE verwendet.

Für die Entwicklung der grafischen Benutzerschnittstelle wird eine spezielle, für eingebettete Systeme optimierte Version des WPF (Windows Presentation Foundation) verwendet. Eine .NetMF-WPF-Applikation besteht aus mehreren Fenstern (Window). Im Gegensatz zur PC-WPF-Entwicklung wird das Aussehen des Fensters nicht mittels XAML[1], sondern im C#-Quellcode realisiert. Die Komponenten (z.B.: Textbox, StackPanel, MenuItem, Bitmap, …) sind die Gleichen, die auch in der PC-Version eingesetzt werden, sodass ein WPF-PC-Applikations-Informatiker ohne weitere Ausbildung die grafische Benutzerschnittstelle von eingebetteten Systemen erstellen kann.

In Abbildung 25 ist der Quellcode eines WPF-Fensters dargestellt. In diesem wird ein Canvas-Objekt implementiert, welches zum absoluten Platzieren von Grafik-Elementen benutzt wird. In diesem Beispiel werden Textelemente an bestimmten absoluten Koordinaten platziert.

[1] XAML: Extensible Application Markup Language, ist eine XML-basierte Sprache für die Gestaltung von grafischen Benutzeroberflächen.

```
/// <summary>
    /// This class demonstrates positioning text objects directly on a canvas
    /// object.
    /// </summary>
    internal sealed class CanvasPanelDemo : PresentationWindow
    {
        /// <summary>
        /// Constructs a CanvasPanelDemo for the given program.
        /// </summary>
        /// <param name="program"></param>
        public CanvasPanelDemo(MySimpleWPFApplication program)
            : base(program)
        {
            Canvas canvas = new Canvas();
            this.Child = canvas;
            this.Background = new SolidColorBrush(ColorUtility.ColorFromRGB(0, 255,
                255));

            for (Int32 x = 0; x < Width; x += Width / 4)
            {
                for (Int32 y = 0; y < Height; y += Height / 4)
                {
                    Text text = new Text(MySimpleWPFApplication.SmallFont,
                        " (" + x + "," + y + ")");
                    Canvas.SetLeft(text, x);
                    Canvas.SetTop(text, y);
                    canvas.Children.Add(text);
                }
            }
        }
    }
```

Abbildung 25: C#-Quellcode eines WPF-Fensters (eigene Darstellung)

Vergleich der GUI-Frameworks

	PC-Applikation	Multilingual	Emulator	Fonts	Controls	Animation	File-System	C-Quellcode
Crank Storyboard	x	x	x	x	x	x		
EasyGUI	x	x	x	x				x
µGFX	x	x		x	x		x	x
Embedded Wizard	x		x	x	x			x
.Net - WPF		x	x	x	x	x	x	

Tabelle 2: Vergleich - GUI-Frameworks

3.6.3.3 Mehrsprachigkeit

Benutzerschnittstellen beherbergen Informationen für den Benutzer, die mit Grafik und Text dargestellt werden. Bei eingebetteten Systemen ist oft eine Mehrsprachigkeits-Unterstützung gefordert.

Der Programmierer legt beim Entwickeln der Benutzerschnittstelle die Texte in seiner Muttersprache fest. Die Übersetzer generieren aus diesen Quell-Texten die Texte in anderen Sprachen.

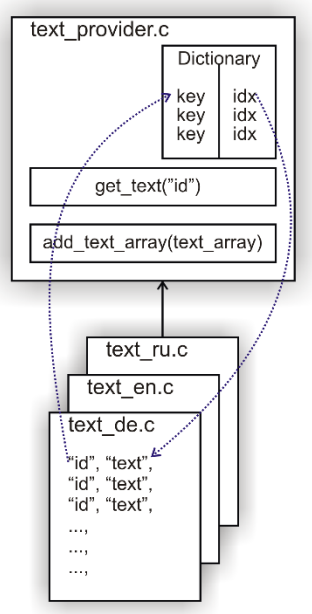

Abbildung 26: Mehrsprachiges Text-System (eigene Darstellung)

Wie im Konfigurationssystem (siehe Kapitel 3.4) bereits vorgestellt wurde, können mit solch einem Text-System Texte in mehreren Sprachen angelegt und verwaltet werden. Die Anzahl der Konflikte aufgrund paralleler Bearbeitung der Texte durch mehrere Entwickler werden dadurch auf ein Minimum reduziert. Außerdem ist dieses Text-System-Konzept unabhängig vom eingesetzten UI-Framework und kann in nahezu allen eingebetteten Systemen realisiert werden.

3.6.3.4 Encoding

Für die Darstellung von Textinformationen werden aufeinanderfolgende Codeworte mittels Zeichensätze zu Schriftzeichen zugeordnet. Es wurden im Laufe der Zeit Zeichensätze zur eindeutigen Interpretation der gespeicherten oder übertragenen Codeworte standardisiert.

Gängige Zeichensätze sind (vgl. (Schneider, 2012)):

- ASCII / ISO-7-Bit-Code:
  Jedes Zeichen wird mit 7 Bit dargestellt. Die Zeichentabelle beherbergt das lateinische Alphabet, Ziffern, Satz- und Steuerzeichen.
- ISO-8-Bit-Codes / ISO/IEC 8859:
  … ist die internationale Referenz-Version des ISO-7-Bit-Codes. Für jede Sprachgruppe existiert ein eigener Zeichensatz (z.B.: ISO/IEC 8859-1 für Westeuropa oder ISO/IEC 8859-5 für die Darstellung kyrillischer Schriftzeichen).
- Unicode:
  … ist ein universeller Zeichensatz, mit welchem Zeichen aller Sprachgruppen und Schriftkulturen dargestellt werden können. Die 8-Bit-Darstellung UTF-8 wird am häufigsten eingesetzt, da das lateinische Alphabet, Ziffern, Satz- und Steuerzeichen durch einzelne Bytes (wie bei ASCII) dargestellt werden können. Zeichen anderer Sprachgruppen und Schriftkulturen werden durch längere Bytefolgen dargestellt.

Auch in eingebetteten Systemen spielt das Encoding eine elementare Rolle. Es sind vor allem die limitierten Zeichensätze von Display und Drucker, die das Umsetzen von Zeichen anderer Sprachen erschweren.

In größeren eingebetteten Systemen, wo ein Betriebssystem eingesetzt wird, ist es ratsam UTF-8 zu benutzen. Mit diesem Encoding können Zeichen sämtlicher Sprachen eingesetzt werden.

Bei kleinen eingebetteten Systemen, beispielsweise mit alphanumerischen Displays, müssen je Sprache Sonderzeichen in speziell definierte Zeichencodes umgewandelt werden. Diesen Zeichencodes ist meistens ein individuell erstelltes Zeichen zugeordnet. Bei Sprachen mit nur wenigen Sonderzeichen ist dieser Aufwand gerechtfertigt, Sprachen wie Arabisch oder Chinesisch können mit dieser Methode aber nicht implementiert werden.

3.7 Herausforderung - Berichte

3.7.1 Kontext

Programmiersprache: unabhängig
Programmierparadigma: imperative und objektorientierte Programmierung

Berichte sind Zusammenstellungen von Informationen, die ausgedruckt oder über Kommunikationsschnittstellen exportiert werden. Entscheidend bei der Erstellung von Berichten sind die Formatierung, der Inhalt und die Sprache.

Einsatz-Beispiele sind: Journal-Ausdruck, Status-Ausdruck, Status-Datei, Status-email, Messdatensatz, …

3.7.2 Herausforderungen

Folgende Herausforderungen ergeben sich beim Konzeptionieren eines Berichtes:

- Berichte sollen für den Programmierer und später auch für den Benutzer des eingebetteten Systems schnell und intuitiv zu erstellen sein.
- Das Aussehen bzw. Format des Berichtes soll jederzeit änderbar sein.
- Berichte sollen für mehrere Sprachen erstellt werden können.

3.7.3 Lösungen

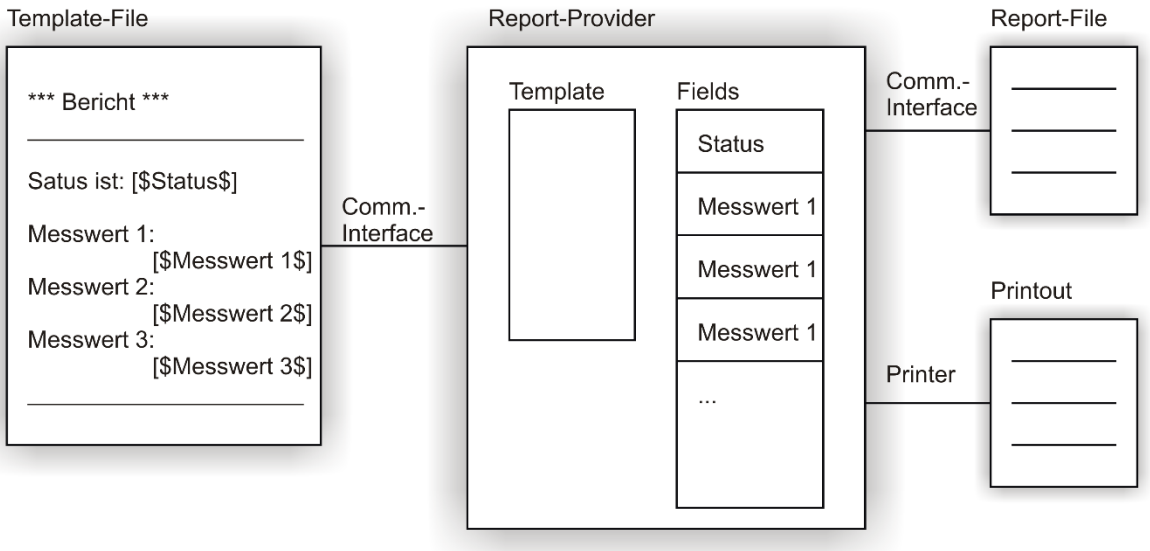

Abbildung 27: Report-Provider
(eigene Darstellung)

Die Lösung ergibt sich durch den Einsatz von Template-Dateien. Template-Dateien sind Textdateien, deren Inhalt aus Text und Feldern besteht. Diese Felder sind Platzhalter, die beim Generieren der Berichte durch die tatsächlichen Werte ausgetauscht werden.

Pro Sprache wird eine Template-Datei erstellt und dem Report-Provider übermittelt. Die Übermittlung ist durch die Kommunikationsschnittstelle realisiert. Somit kann der Benutzer diese Text-Dateien erstellen und mit einem entsprechenden Programm über die Kommunikationsschnittstelle transferieren. Im Report-Provider wird das Template-File interpretiert und abgelegt.

Beim Ausdruck oder beim Datenexport wird die Generierung des Berichtes ausgeführt. Der Bericht wird entweder über die Kommunikationsschnittstelle weitergeleitet oder über den Drucker ausgedruckt.

3.8 Herausforderung - Kommunikationsschnittstelle

3.8.1 Kontext

Programmiersprache: unabhängig
Programmierparadigma: imperative und objektorientierte Programmierung

Für eingebettete Systeme ist neben der Benutzerschnittstelle die Kommunikationsschnittstelle eine wichtige Komponente, um mit der Umgebung Informationsaustausch betreiben zu können.

Die Kommunikationsschnittstelle ist definiert als Kommunikationssoftware die mehrere Kommunikationsschichten implementiert (siehe Abbildung 28). Diese Kommunikationssoftware wird oft auch Kommunikationsstapel (englisch: communication stack) genannt.

Die Kommunikation eingebetteter Systeme hat in den letzten Jahren immer mehr Relevanz erfahren. So werden in der Automatisierungstechnik ganze Netzwerke von eingebetteten Systemen etabliert, die digitalisierte Daten von Sensoren und Befehlsanforderungen zu Aktoren transportieren. Auch die Drahtloskommunikation wird bereits in Industrie und Automatisation eingesetzt, um ebenso Netzwerke von eingebetteten Systemen zu realisieren. (vgl. (Berns, 2010))

Auch Web-Technologien, die es ermöglichen die Wartung und Kontrolle eines Gerätes von der Ferne zu betreiben, setzten sich im Bereich der eingebetteten Systeme durch. So ist es möglich, mittels eines ganz normalen WWW-Browsers von jedem beliebigen PC, Tablet oder Smartphone aus, ein eingebettetes System, wo ein simpler Web-Server implementiert ist, zu bedienen oder zu warten. (vgl. (Berns, 2010))

Ein großer Anwendungsbereich für die Kommunikation von eingebetteten Systemen ist neben dem Austausch von Laufzeit-/Mess-Daten, die Wartung und Kontrolle von eingebetteten Systemen. Es werden Konfigurationsdaten ausgetauscht und Programmaktualisierungen damit vollzogen.

3.8.2 Herausforderungen

Die Entwicklung von Kommunikationsschnittstellen gehört seit jeher zu den herausforderndsten Gebieten der Technik. Die Konzeptionierung, Ausführung und das Testen von Kommunikation sind aufwendige Prozesse im Zuge der Entwicklung von eingebetteten Systemen.

Eine Kommunikationsschnittstelle soll folgende Merkmale aufweisen:

- Die Kommunikationsschnittstellen-konsumierenden Schichten sollen eine einheitliche Daten-Schnittstelle zur Verfügung gestellt bekommen.
- Es soll die Kommunikation über verschiedene Übertragungskanäle (Ethernet, USB, UART, WLAN, …) bewerkstelligt werden können.
- Die Kommunikationsschnittstelle soll Anbindungen zur Kommunikations-Peripherie des Mikroprozessors implementieren.

3.8.3 Lösungen

In diesem Kapitel wird ein Konzept zur Realisierung einer Kommunikationsschnittstelle vorgestellt. Es werden die einzelnen Komponenten darin erläutert und dabei auf bereits bestehende Ressourcen verwiesen. Diese Verweise bestehen größtenteils aus URLs von wo Dokumentation und Quellcode bezogen werden können. Dieses Konzept besteht also aus einem Leitfaden zur Erstellung einer Kommunikationsschnittstelle und einer Sammlung von Verweisen auf Ressourcen für die konkrete Implementierung.

Der Kommunikationsstapel (communication stack) (siehe Abbildung 28) besteht in den unteren Schichten aus den Anbindungs-Treibern zu der Kommunikations-Peripherie des Mikroprozessors (USB, Ethernet, UART, CAN, …). Über den Anbindungstreibern sitzt die Kommunikationsprotokoll-Schicht in welcher das Kommunikationsprotokoll realisiert ist. Der Remote-Task bedient das Kommunikationsprotokoll. Dort werden Anfragen (Requests) behandelt und Antworten (Responses) retourniert.

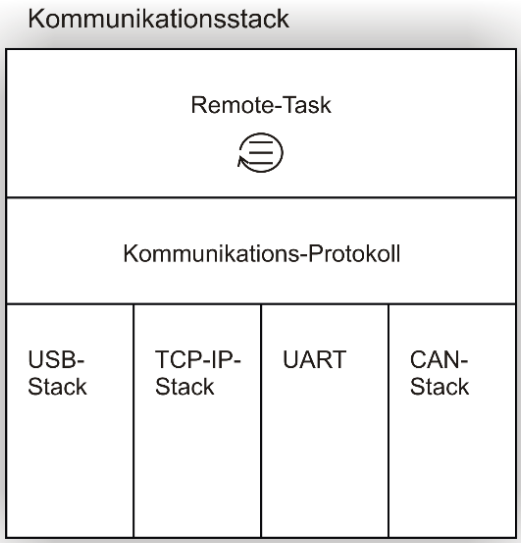

Abbildung 28: Kommunikationsstapel der Kommunikationsschnittstelle (eigene Darstellung)

3.8.3.1 Anbindungstreiber

Anbindungstreiber implementieren die Initialisierung und Bedienung der Kommunikations-Peripherie des Mikroprozessors. Gängige Kommunikations-Peripherien sind USB, Ethernet, UART und CAN. Teilweise beinhalten die Anbindungstreiber wiederrum Kommunikations-Stapel wie TCP-IP-Stack, USB-Stack oder CAN-Stack.

USB

Die USB-Schnittstelle hat in eingebetteten Systemen längst Einzug erhalten und hat größtenteils als Daten-/Service-Schnittstelle die UART-Schnittstelle ersetzt. Der USB-Standard definiert verschiedene Geräteklassen, die sich mittels generischen Treibern ansprechen lassen. Somit muss nicht für jedes eingebettete System ein eigener PC-Treiber entwickelt werden.

In eingebetteten Systemen wird meist die „Communications Device Class“ (CDC) verwendet. Damit lässt sich u.a. ein virtueller COM-Port (VCP) realisieren. Somit wird eine Kompatibilität mit Vorgänger-Geräten, welche noch über die UART-Schnittstelle kommunizierten, erreicht. Die meisten Mikroprozessor-Hersteller bieten einen Quellcode für VCP an. Eine detaillierte Implementierungs-Beschreibung inklusive Quellcode ist zu finden unter: http://janaxelson.com/usb_virtual_com_port.htm (22.09.2016)

Eine weitere Möglichkeit, USB zu implementieren, stellt der generische Treiber „WinUSB“ dar. Dieser besteht aus einem Kernel-Mode-Treiber (Winusb.sys) und einen User-Mode-Treiber (Winusb.dll). Die im User-Mode-Treiber enthaltenen Funktionen können direkt von der Applikation verwendet werden. Am Mikroprozessor können die Daten direkt von den

USB-Endpunkten entgegengenommen werden. Weiterführende Informationen und Quellcode sind zu finden unter:
http://janaxelson.com/winusb.htm (22.09.2016)

Ethernet

Von vielen eingebetteten Systemen wird gefordert, Daten im Netzwerk (LAN, Internet) bereitzustellen. Um dies bewerkstelligen zu können, muss ein TCP-IP-Stapel implementiert werden. Mittlerweile gibt es eine große Anzahl von TCP-IP-Stapel, die speziell für eingebettete Systeme entwickelt worden sind. Viele eingebettete Betriebssysteme wie Embedded-Linux, .Net-Micro-Framework oder FreeRTOS bieten TCP-IP-Stapel an. Auch die Mikroprozessor-Hersteller bieten in deren Framework-Bibliotheken TCP-IP-Stapel an. Gängige TCP-IP-Stapel sind:

- uIP: … ist eine Open-Source-Implementierung eines TCP-IP Stapels. uIP ist optimiert für kleine 8bit/16bit - Mikroprozessor-Derivate.
  http://dunkels.com/adam/software.html (22.09.2016)
- mIP … ist ein TCP-IP Stapel, der zu 100% gemanagten Quellcode[2] enthält und im .Net Micro Framework eingesetzt wird.
  https://mip.codeplex.com/ (22.09.2016)
- lwIP … ist ein TCP-IP-Stapel ähnlich dem uIP, der für kleine Mikroprozessor-Derivate konzipiert worden ist.
  http://savannah.nongnu.org/projects/lwip/ (22.09.2016)
- Keil TPC-IP Networking Suite … ist ein kostenpflichtiger TCP-IP-Stapel mit http-Server, der speziell für ARM-Mikroprozessoren optimiert wurde.
  http://www.keil.com/rl-arm/rl-tcpnet.asp (22.09.2016)

Weiterführende Informationen und Quellcode für Ethernet und Internet sind zu finden unter:
http://janaxelson.com/ethernet.htm (22.09.2016)

UART

UART (universal asynchronous receiver/transmitter) ist in eingebetteten Systemen immer noch weit verbreitet. Sie wird vorwiegend für die Anbindung externer Peripherie-Geräte (Journaldrucker, Kartenleser, Transponder, …) verwendet. Der Vorteil von UART besteht durch die simple Implementierungs- und Test-Möglichkeiten. Dadurch eignet sich UART auch hervorragend für das Debugging der eingebetteten Software, da Werte und Zustände einfach übertragen werden können.

[2] Gemanagter Quellcode ist der Quellcode, der mit einer .Net-Programmiersprache verfasst wird und kann nur mit der Common Language Runtime ausgeführt werden.

Weiterführende Informationen und Quellcode für UART sind zu finden unter: http://janaxelson.com/serport.htm (22.09.2016)

Feldbusse / CAN

Feldbusse sind Bussysteme zur Vernetzung von Sensoren, Aktoren und Steuergeräten. Sie werden vorwiegend in der Automatisierungstechnik zur automatisierten Produktion und im Automobilbereich eingesetzt. Die Gebäudeautomatisierung (Smart Home) ist ein weiteres, wachsendes Einsatzgebiet für Feldbusse. Bekannte Feldbusse für Gebäudeautomatisierung sind KNX, LNC und EIB.

Die Feldbusse werden in folgende Ebenen eingeteilt (vgl. (Berns, 2010)):

- die Sensor-Aktuator-Ebene
- die Systembusebene

Feldbusse zeichnen sich durch die Echtzeitfähigkeit und die Verlässlichkeit aus. Der Datenaustausch zwischen Sensor und Steuergerät muss rechtzeitig ausgeführt werden können. Nur so ist eine korrekte Regelung aufgrund der übertragenen Eingangsgrößen möglich. Die Verlässlichkeit wird durch Fehlersicherung gewährleistet. Dabei werden die empfangenen Daten auf Fehler untersucht und im Falle eines Übertragungsfehlers eine Fehlerkorrektur durchführt.

Bekannte Feldbusse sind (vgl. (Berns, 2010)):

- ASI (Aktor-Sensor-Interface) (Sensor-Aktuator-Ebene)
  … ist ein Zweidraht-Bussystem für die Kommunikation von Sensoren und Aktoren mit dem übergeordneten Steuergerät.
- Interbus (Sensor-Aktuator-Ebene)
  … ist ein Bussystem mit zeitlich-deterministischem Verhalten. Die Topologie entspricht einem Ring mit getrennter Hin- und Rückleitung durch alle Stationen.
- Profibus (Systembusebene)
  … ist ein Bussystem, welches speziell für Fertigungsautomatisierung entwickelt wurde. Es gibt 3 Varianten: Profibus-DP (Dezentrale Peripherie), Profibus-PA (Prozess-Automation) und Profibus-FMS (Fieldbus Message Specification).
- Busse im Automobilbereich: LIN, FexRay, CAN, MOST
  Der CAN-Bus hat sich im Laufe der Zeit als das Bussystem im Automobilbereich durchgesetzt. Da die Kommunikationsanforderungen stetig steigen wird auf der Sensor-Aktuator-Ebene mittlerweile LIN eingesetzt um den CAN-Bus zu entlasten. Auf der Steuergeräteseite wird CAN durch das Bussystem FlexRay mit Übertragungsraten von bis zu 10Mbit/sec erweitert.

CAN (Controller Area Network) ist ein (Feld-)Bussystem, welches vor allem im Automobilbereich Einsatz findet. Die Kommunikation im CAN-Bus zeichnet sich durch die stabile und störungssichere Übertragung von Daten aus. Der CAN-Controller im Mikroprozessor ist aufwendig konstruiert und beherrscht dadurch hervorragend das Fehler-/Kollisions-Management beim Übertragen der Daten. Der Implementierungsaufwand eines CAN Software-Treibers ist vergleichbar mit dem, des UART-Treibers.

Für die Kommunikation mittels CAN-Bus etablierten sich im Laufe der Zeit OSI-Modell-Schicht-7-Protokolle. Das bekannteste davon ist CANopen, welches speziell in der Automatisierungstechnik angewendet wird. CANopen ist als Norm EN 50325-4 standardisiert und wird von „CAN in Automation (CiA) gepflegt. (http://www.can-cia.org/can-knowledge/canopen/canopen/ (22.09.2016)). Gängige CANopen-Stacks sind:

- CAN-Festival CANopen
  http://www.canfestival.org/ (22.09.2016)
- IxxAT – CANopen Protocol Software:
  http://www.ixxat.com/products/products-industrial/protocol-software-and-apis/protocol-software/canopen-protocol-software (22.09.2016)
- CANopenNode
  https://github.com/CANopenNode/CANopenNode (22.09.2016)

3.8.3.2 Kommunikationsprotokoll

Das Kommunikationsprotokoll ist eine Vereinbarung, nach der die Übertragung von Daten zwischen mehreren Kommunikationsteilnehmern vollzogen wird.
Konkret wird im Kommunikationsprotokoll vereinbart, wie und mit welchem Format Daten zwischen den Kommunikationsteilnehmern übertragen werden. Das Kommunikationsprotokoll ist hardware-unabhängig und kann mit unterschiedlichen Anbindungstreibern (siehe Abbildung 28) betrieben werden.

Protokoll-Aufbau

Das Kommunikationsprotokoll vereinbart Botschaften, die zwischen den Kommunikationsteilnehmern ausgetauscht werden. Die Kommunikationssynchronisation verläuft nach dem Master-Slave-Prinzip. Es existiert ein Master, welcher Daten anfordert oder Befehle erteilt. Der Slave retourniert die geforderten Daten oder führt die Befehle aus. Auf jeden Fall wird pro Anforderungs-Botschaft eine Antwort-Botschaft retourniert.

Format der Botschaften

Eine Botschaft besteht aus mehreren Elementen. Diese Elemente, in weiterer Folge Felder genannt, bestimmen das Format einer Botschaft. In Abbildung 29 ist ein praxiserprobtes Botschaften-Format dargestellt.

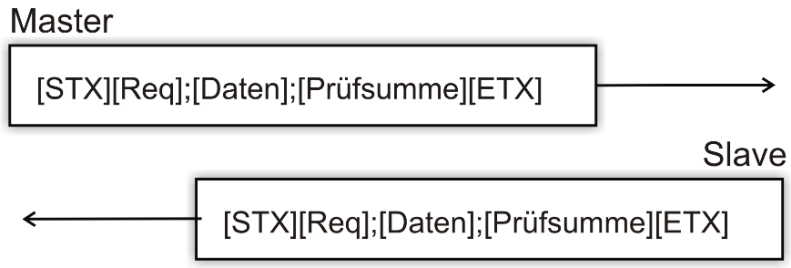

Abbildung 29: Format einer Botschaft (eigene Darstellung)

Folgende Felder sind in der Botschaft implementiert:

- [STX]
  start of text (0x02)
- [Req]
  16bit Request-Nummer, wird im Remote-Task definiert und interpretiert. Die Request-Nummer wird dezimal dargestellt.
- [Daten]
  Optionales Datenfeld. Die Daten werden in Form eines Key-Value-Strings dargestellt.
  Beispiel: `voltage=25.4,temperature=-12.4,current=3.23,status=IDLE,`
- [Prüfsumme]
  8bit-Prüfsumme, die die Summe aller Bytes der Botschaft bildet. Die Prüfsumme wird hexadezimal dargestellt.
- [ETX]
  end of text (0x03)

Beispiel einer Datenanforderung

Es werden mit dem Request Nr. 10012 Messdaten vom Slave angefordert:

```
[STX]10012;59[ETX]
```

Der Slave antwortet mit folgender Botschaft:

```
[STX]10012;voltage=25.4,temperature=-12.4,current=3.23,sta-
tus=IDLE,;47[ETX]
```

Key-Value-String

Die zu übertragenden Daten werden in Form eines Key-Value-Strings dargestellt. Der große Vorteil dieser Methode ist, dass die Daten erweitert werden können, ohne die Kompatibilität zu älteren Versionen zu gefährden.

Einzelne Datenwerte werden in Form eines Key-Value-Paars dem Key-Value-String hinzugefügt. Der Key (Schlüssel) muss dafür eindeutig definiert werden. Pro Key (Schlüssel) kann ein Value (Wert) im Key-Value-String adressiert werden. Zwischen Key (Schlüssel) und Value (Wert) wird ein Relations-Zeichen, meistens ‚=', eingefügt. Die einzelnen Key-Value-Paare werden durch ein Separator-Zeichen, meistens ‚,' getrennt. Einem Schlüssel kann wiederum als Wert ein eingenisteter Key-Value-String zugeordnet sein:

Folgende syntaktische Varianten von Key-Value-Strings können implementiert werden:

```
key=value,key=value,                                    normal key-value

key={key=value,key=value,},                             nested key-value

key=[{key=value,key=value,},{key=value,key=value,},],   array of key-values
```

3.8.3.3 Remote-Task

Der Remote-Task ist in der obersten Schicht des Kommunikationsstapels beherbergt. Er implementiert den Programmablauf für die Kommunikation zwischen den Kommunikationspartnern. Die Aufgabe des Remote-Tasks besteht in der Abarbeitung von Anforderungen. Er interpretiert empfangene Botschaften und stellt entsprechend der, in der Botschaft enthaltenen Anforderungsnummer (Request-Nummer) eine Antwort-Botschaft mit den geforderten Daten zusammen. Diese Antwort-Botschaft übermittelt der Remote-Task dem Sender der Anforderungs-Botschaft.

Umgekehrt werden zyklisch Anforderungsbotschaften dem Kommunikationsteilnehmer übermittelt, deren Antwort-Botschaften interpretiert und deren Daten abgelegt.

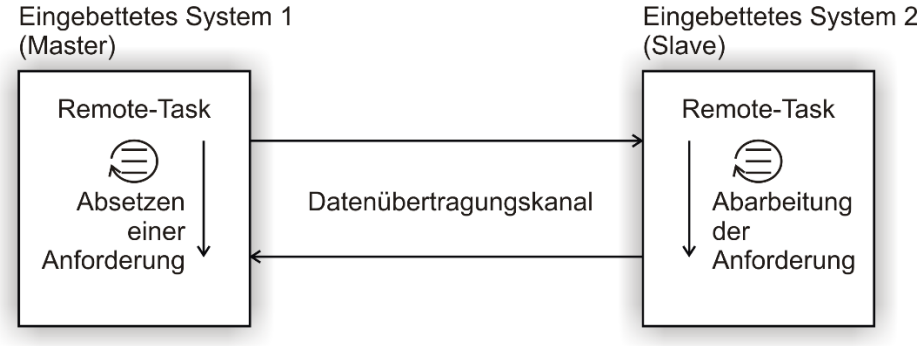

Abbildung 30: Remote-Task Kommunikations-Ablauf (eigene Darstellung)

3.8.3.4 Zusammenfassung

Die Kommunikationsschnittstelle wird je nach Anwendung und Einsatzgebiet angepasst. Elementar ist, dass die Datenschnittstellen im Kommunikationsstapel (siehe Abbildung 28) für alle Kommunikationskanäle einheitlich konzeptioniert wird.

Das Kommunikationsprotokoll bzw. das Format der Botschaften muss entsprechend der Kommunikationsteilnehmer angepasst werden. Soll beispielsweise mit der Kommunikationsschnittstelle eine Datenübertragung zu einer eigens, für die Konfiguration und Wartung erstellten PC-Applikation realisiert werden, kann das Kommunikationsprotokoll bzw. das Format der Botschaften durchaus entsprechend dem Konzept aus Kapitel 3.8.3.2 entwickelt werden. Wird die Kommunikationsschnittstelle allerdings benutzt, um mittels Feldbus-vernetzten Sensoren und Aktoren zu kommunizieren, muss das Kommunikationsprotokoll bzw. das Format der Botschaften an die, im Feldbus eingesetzten Geräte, angepasst werden.

In der Praxis wird es vorkommen, dass ein eingebettetes System mit mehreren unterschiedlichen Kommunikationsteilnehmern Datenaustausch betreibt. In solch einem Fall werden mehrere Kommunikationsschnittstellen mit unterschiedlichen Kommunikationsprotokollen implementiert. (siehe Abbildung 31)

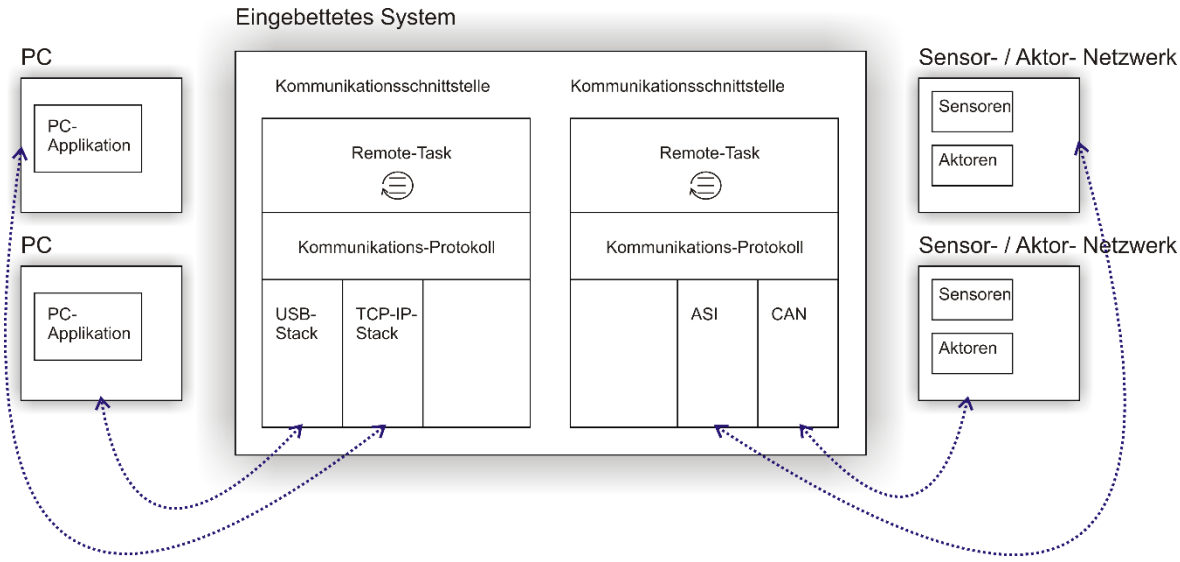

Abbildung 31: Multiple Kommunikationsschnittstellen in einem eingebetteten System (eigene Darstellung)

3.9 Herausforderung - Fehlermanagement

3.9.1 Kontext

Programmiersprache: C
Programmierparadigma: imperative und objektorientierte Programmierung

Das Fehlermanagement befasst sich mit der Behandlung, Benachrichtigung und Protokollierung von Fehlern, die in einem Programm auftreten können. Verschiedene Programmiersprachen beherbergen verschiedene Technologien mit denen das Fehlermanagement realisiert werden kann. Programmiersprachen wie C++, Java, C#, usw. wurden mit „Ausnahmen“ (exceptions) ausgestattet. Die Programmiersprache C hingegen implementiert keine Hilfsmittel für das Fehlermanagement. Stattdessen werden Fehlercodes benutzt, die als Rückgabewert von Funktionen den aufrufenden Funktionen bereitgestellt werden.

3.9.2 Herausforderungen

Die Behandlung von Fehlern mit der Programmiersprache C wurde im Buch „Programmieren in C“ (Klima, 2010) zielführend behandelt.

Dabei werden Fehlercodes mit Aufzählungstypen realisiert, sodass gewährleistet ist, dass für jeden Fehler ein eindeutiger Wert definiert wird. Beispiel:

```
typedef enum
{
        ERR_NO_ERROR = 0,
        ERR_NO_MEMORY,
        ERR_INVALID_NAME,
        ERR_DIV_BY_ZERO,
        ERR_FILE_NOT_FOUND,
        ERR_TOTAL_NUM
} ERR_Type_t;
```

Abbildung 32: Fehlercodes realisiert mit Aufzählungstyp (Klima, 2010)

Den Fehlercodes sind aussagekräftige Texte zugeordnet, diese werden in einem Array der C-Struktur (siehe Abbildung 33) definiert. Dabei wird bewusst eine Redundanz der Fehlercodes erzeugt.

```
typedef struct
{
        ERR_Type_t errNum;
        const char *errStr;
} ERR_t;
const ERR_t ErrTable[] =
{
        { ERR_NO_ERROR,
        "No error occured."
        },
        { ERR_NO_MEMORY,
        "Memory could not be allocated."
        },
        { ERR_INVALID_NAME,
        "The given name is invalid."
        },
        { ERR_DIV_BY_ZERO,
        "Division by zero."
        },
        { ERR_FILE_NOT_FOUND,
        "File not found."
        }
};
```

Abbildung 33: Fehlertexte
(Klima, 2010)

Mit dieser Redundanz der Fehlernummer und der Position im Array kann die Konsistenz der Fehlertexte zu den Fehlercodes überprüft werden. Die realisierende Funktion ist in Abbildung 34 dargestellt.

```
        long code;
        // ...
        for (code = 0; code < ERR_TOTAL_NUM; code = code + 1)
        {
                if (ErrTable[code].errNum != code)
                {
                        printf("ErrTable inkonsistent an Position %ld\n", code);
                        break;
                }
        }
```

Abbildung 34: Konsistenzüberprüfung der Fehlertexte
(Klima, 2010)

Die Fehlerweitergabe wird mittels Rückgabewert der Funktionen realisiert. Jede Funktionsebene muss die Fehlercodes bis zu der Ebene transportieren, wo der Fehler behandelt und die Fehlerbenachrichtigung generiert wird.

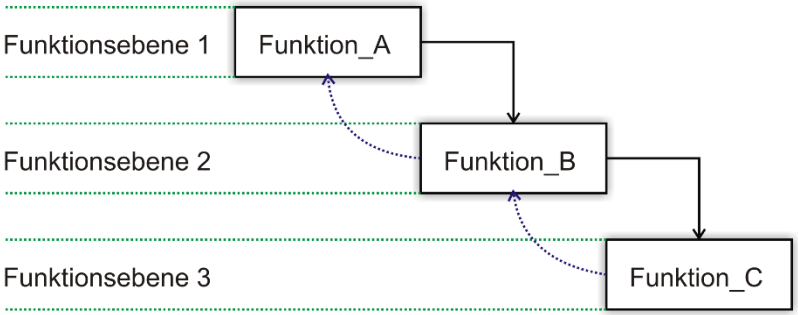

Abbildung 35: Fehlerweitergabe (eigene Darstellung)

Laut (Klima, 2010) gibt es zwei Ansätze in welcher Funktionsebene Fehler behandelt werden sollen:

- Der Fehler wird in derselben Funktion behandelt.
- Die Behandlung des Fehlers ist in derselben Funktion nicht möglich. Die Funktion wird abgebrochen und der Fehler an eine aufrufende Funktion weitergeleitet, die ihn behandeln soll.

Das Behandeln der Fehler in derselben Funktion birgt den Nachteil in sich, dass der Quellcode zur Fehlerbehandlung in jeder Funktion implementiert werden muss. Somit wird redundanter Quellcode erzeugt, der bei Änderungen zur Inkonsistenz der Fehlerbehandlung führen kann. Auch die Fehlerprotokollierung muss in allen Funktionen, wo ein Fehler auftreten kann, realisiert werden.
Die Behandlung der Fehler in den oberen Funktionsebenen ist zu bevorzugen. Der Quellcode der Fehlerbehandlung und der Fehlerprotokollierung muss dann nur an einer oder wenigen Stelle(n) im Programm implementiert werden. Der Nachteil dieser Methode ist, dass aus allen Funktionsebenen die Fehlercodes retourniert werden müssen. Wird dies beim Erweitern des Programms einmal versäumt, ist die Fehlerweitergabe-Kette unterbrochen und es kommt weder zu einer Fehlerbehandlung noch zu einer Fehlerbenachrichtigung.

Ein gutes Fehlermanagement muss folgende Kriterien erfüllen:

- Die Behandlung, Benachrichtigung und Protokollierung der Fehler muss in jeder Funktionsebene implementiert werden können.
- Die Behandlung, Benachrichtigung und Protokollierung der Fehler soll in, voneinander getrennten Funktionsebenen implementiert werden können.
- Die Fehlerweitergabe soll nicht über die Rückgabewerte der Funktionen realisiert werden müssen.
- Das Fehlermanagement muss systemübergreifend Gültigkeit haben.

3.9.3 Lösungen

Abbildung 36 zeigt ein Konzept für das Fehlermanagement, welches die, in Kapitel 3.9.2 beschriebenen Herausforderungen abdeckt.

Die Fehlercodes sind genauso wie in Abbildung 32 mit einem Aufzählungstyp eindeutig definiert. Hinzu kommen die Fehlergruppen, die auch mit einem Aufzählungstyp definiert werden. Die Zuordnung, welcher Fehlercode zu welcher Fehlergruppe gehört, wird in einer Tabelle getätigt. In dieser Tabelle werden den Fehlercodes Text-IDs zugeordnet, mit denen der Fehlertext aus dem mehrsprachigen Text-System (siehe Kapitel 3.6.3.3) ausgelesen werden kann.

In der Funktion, wo der Fehler auftritt, wird dieser mit der set_error()-Funktion dem Fehlerspeicher hinzugefügt. Der Fehlerspeicher ist eine Warteschlange (FIFO-Prinzip: First In First Out).
Die set_error()-Funktion fügt auch dem Trace-Speicher einen Fehler-Eintrag hinzu. Der Trace-Speicher ist persistent (siehe Kapitel 3.5) und ist über die Benutzer- bzw. Kommunikationsschnittstelle auslesbar. Somit wird bereits beim Bekanntgeben eines Fehlers die Fehlerprotokollierung bewerkstelligt.

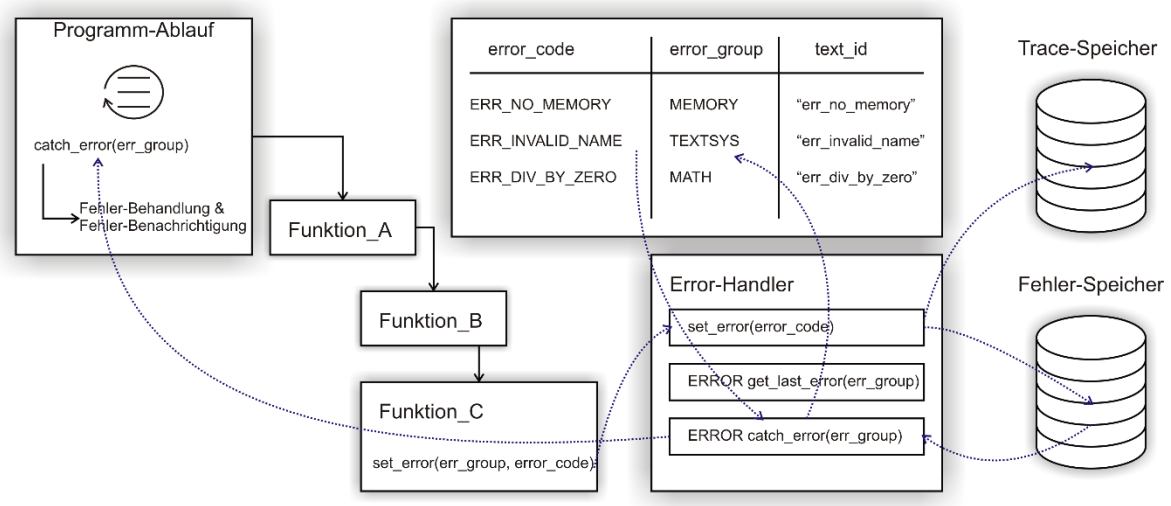

Abbildung 36: Fehlermanagement (eigene Darstellung)

In den oberen Funktionsebenen, in Abbildung 36 konkret im Programmablauf, wird der Fehler mit der catch_error()-Funktion aus dem Fehler-Speicher geholt und entsprechend die Fehler-Behandlung und Fehler-Benachrichtigung initiiert. Die catch_error()-Funktion retourniert den Fehler in Form der ERROR-Struktur (siehe Abbildung 37), diese beinhaltet die Fehlergruppe, den Fehlercode und die Fehlertext-ID zum Auslesen des Fehlertextes.

```
typedef struct
{
        BOOL is_catched;        // true if this error has been catched
        ERR_CODE error_code;    // error code
        ERR_GROUP err_group;    // error group
        char text_id[32];       // text id to get error text
                                // from multilingual text provider
}ERROR;
```

Abbildung 37: ERROR – Struktur
(eigene Darstellung)

Ist aktuell kein Fehler eingetragen, retourniert die catch_error()-Funktion NULL an die aufrufende Funktion.

Mit der Funktion get_last_error() kann der jeweils letzte, noch nicht behandelte Fehler, ausgelesen werden. Somit könnte beispielsweise in den mittleren Funktionsebenen auf Fehler reagiert werden, während in den oberen Funktionsebenen die eigentliche Fehlerbehandlung ausgeführt wird.

Beide Funktionen, catch_error(), get_last_error() übernehmen als Argument die Fehlergruppe. So können Fehler unterschiedlicher Gruppen in unterschiedlichen Programmteilen behandelt werden.
Realisiert wird dies mit dem is_catched-Flag, welches in der ERROR-Struktur beinhaltet ist.

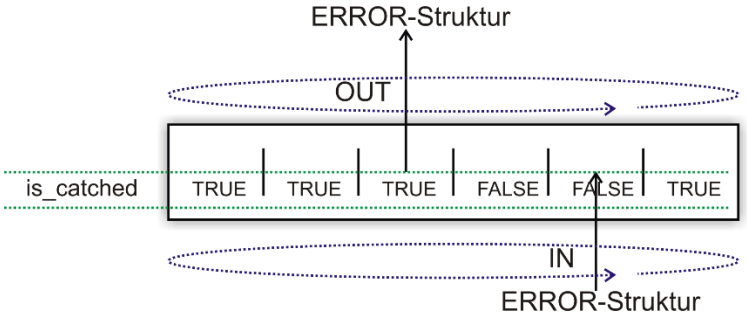

Abbildung 38: Fehlerspeicher
(eigene Darstellung)

In der catch_error()-Funktion werden alle Fehler, die zwischen dem IN- und OUT- Pointer liegen, durch-iteriert. Bei jedem Fehler wird die Fehlergruppe mit der übergebenen Fehlergruppe verglichen Bei Übereinstimmung wird das is_catched-Flag dieses einen Fehlers zurückgesetzt, der Fehler ist nun als behandelt markiert. Ist bei diesem behandelten Fehler gerade der OUT-Pointer positioniert, wird dieser auf den nächsten Fehler-Eintrag gesetzt. Die Differenz zwischen IN- und OUT- Pointer entspricht der aktuellen Speicherauslastung des Fehlerspeichers.

Die get_last_error()-Funktion arbeitet nach dem gleichen Prinzip, setzt aber das is_catched-Flag nicht.

3.9.3.1 Zusammenfassung

Mit dieser Methode kann ein systemweites Fehlermanagement realisiert werden, welches der Ausnahmebehandlung hoher, objektorientierter Programmiersprachen entspricht. Es kann damit

- die Fehlerbehandlung
- die Fehlerbenachrichtigung
- und die Fehlerprotokollierung

realisiert werden.

Es werden mit dieser Methode redundante Fehlerbehandlung und Fehlerbenachrichtigung vermieden. Die Erweiterbarkeit durch neue Funktionsebenen ist unproblematisch, da die Fehlerweitergabe nicht durch Rückgabewerte der Funktionen realisiert ist.
Auch spezielle Fehler die nur in wenigen Funktionen auftreten können, können mit diesem Konzept effektiv behandelt werden, da die Fehlerprotokollierung und Fehlerbenachrichtigung nicht eigens ausprogrammiert werden müssen.

4 Zusammenfassung und Ausblick

4.1 Ergebnisse

In dieser Arbeit wurden Konzepte für immer wiederkehrende Herausforderungen bei der Entwicklung von eingebetteter Software erarbeitet. Dabei wurde bewusst bei der Ausarbeitung der einzelnen Konzepte auf die Sicht der Praxis mit Bezug auf bestehende Literatur geachtet.

Es wurden Konzepte/Herangehensweisen für

- … die Entwicklung eingebetteter Software nach den objektorientierten Prinzipien erarbeitet.
- … das Implementieren der Programmabläufe, also dem Verhalten eines eingebetteten Systems, erarbeitet.
- … das Konfigurationssystem im Sinne der Konfigurations-Parameter-Verwaltung, mit der Einstellungen am eingebetteten System vorgenommen werden können, erarbeitet.
- … das Persistenz-System zum dauerhaften Speichern von Daten erarbeitet.
- … die Benutzerschnittstelle, mit welcher die Interaktion zwischen Mensch und Gerät ermöglicht ist, erläutert.
- … das Erstellen und Verwalten von Berichten im Sinne des Daten-Exports/-Ausdrucks erarbeitet.
- … die Kommunikation zwischen eingebetteten Systemen erarbeitet.
- … für das Fehlermanagement, also die Fehler-Behandlung, Benachrichtigung, Protokollierung erarbeitet.

Nach Auffassung des Autors sind diese Kernthemen der Entwicklung eingebetteter Software in der bestehenden, recherchierten Literatur nicht oder nur sehr allgemein behandelt. Mit dieser Arbeit wird dem Entwickler eine Sammlung von konkreten, praxistauglichen Konzepten bereitgestellt, welche direkt implementiert werden können.

4.2 Ausblick

In dieser Arbeit sind die Kernthemen der Entwicklung eingebetteter Software behandelt worden. Die darin erarbeiteten Konzepte sind zur konkreten Implementierung geeignet. Diese werden im Laufe der Zeit, entsprechend ihres praktischen Einsatzes, erweitert und optimiert. Weitere Konzepte werden zu dieser Sammlung von Konzepten und Herangehensweisen hinzugefügt werden.

Möglicherweise werden diese Konzepte auch online verfügbar gemacht, sodass immer auf den aktuellen Stand zugegriffen werden kann. Damit wäre mit dieser Sammlung von, speziell für eingebettete Systeme konstruierten Konzepte eine Plattform für den Austausch zwischen Entwicklern vorhanden.

Literaturverzeichnis

Berns, K. (2010). *Eingebettete Systeme - Systemgrundlagen und Entwicklung eingebetteter Sofware.* Wiesbaden: Vieweg+Teubner Verlag.

Goll, J. (2014). *Architektur- und Entwurfsmuster der Softwaretechnik.* Wiesbaden: Springer Vieweg.

Heinecke, A. M. (2012). *Mensch-Computer-Interaktion - Basiswissen für Entwickler und Gestalter.* Berlin Heidelberg: Springer-Verlag.

Hruschka, G. S. (2011). *Software Architektur kompakt.* Heidelberg: Spektrum Akademischer Verlag.

Jackson, M. (1995). *Software Requirements & Specifications: a lexicon of practice, principles.* ACM Press. Addison-Wesley.

Klima, R. (2010). *Programmieren in C.* Wien: Springer-Verlag.

Marwedel, P. (2008). *Eingebettete Systeme.* Berlin Heidelberg: Springer-Verlag.

Noble, J. (2001). *Small Memory Software.* Addison-Wesley.

P.Widmayer, T. /. (2002). *Algorithmen und Datenstrukturen.* Heidelberg . Berlin: Spektrum Akademischer Verlag GmbH.

Qian, K. (2009). *Embedded Software Development with C.* Dordrecht Heidelberg London New York: Springer Science +Business Media.

Schneider, U. (2012). *Taschenbuch der Informatik.* München: Carl Hanser Verlag.

Starke, K. E. (2013). *Patterns kompakt, Entwurfsmuster für effektive Software-Entwicklung.* Berlin Heidelberg: Springer Vieweg.

Wikipedia. (11.06.2015). *Embedded Software Engineering*. (D. f. Wikipedia, Herausgeber) Abgerufen am 12.06.2016 von Embedded Software Engineering: https://de.wikipedia.org/w/index.php?title=Embedded_Software_Engineering&oldid=142996453

Wikipedia. (20.09.2016). *Persistenz (Informatik)*. (D. f. Wikipedia, Herausgeber) Abgerufen am 25.09.2016 von Persistenz (Informatik): https://de.wikipedia.org/w/index.php?title=Persistenz_(Informatik)&oldid=158078796

Anlagen

Anlagen, Teil 1

In diesem Anlagenteil ist der Quellcode vom Kapitel 3.3: „Herausforderung - Programmablauf“ untergebracht.

Quellcode-Header-Datei der spezialisierten Pipe (siehe Kapitel 3.3.3.3):

```
/*
Sequence-Pipe für Inter-Programmablauf-Kommunikation

*/
#ifndef SEQUENCE_PIPE_H
#define SEQUENCE_PIPE_H
#include "task_manager.h"

//============================================================================
//defines
#define PIPE_BUFFER_SIZE 8192
#define PIPE_BUFFER_MASK = PIPE_BUFFER_SIZE-1
#define MAX_OUT_PTR_COUNT 16

//Errors
typedef enum
{
	ERR_NONE = 0,
	ERR_TASK_ID_OVERFLOW = 1,
	ERR_MUTEX_TIMEOUT = 2,
}PIPE_ERROR;

typedef struct seq_pipe_structure SEQUENCE_PIPE;

//----------------------------------------------------------------------------
//key-value-string alias - Struktur
struct seq_pipe_structure
{
	//============================================================================
	//Private Fields
	unsigned char pipe_buffer[PIPE_BUFFER_SIZE];
	unsigned short pipe_in_ptr;
	unsigned short pipe_out_ptrs[MAX_OUT_PTR_COUNT];
	PIPE_ERROR last_error;
	OS_MUT mutex;

	//============================================================================
	//Methods
	//----------------------------------------------------------------------------
	//Retourniert TRUE wenn wenn mindestens ein Byte in der Pipe ist.
	//[this]... this-Pointer
	//[task_id]... Task-Id vom zugreifenden Task
	unsigned char (*byte_rdy)(SEQUENCE_PIPE *this, OS_ID task_id);
	//----------------------------------------------------------------------------
	//Retourniert das nächste Byte aus der Pipe
	//[this]... this-Pointer
	//[task_id]... Task-Id vom zugreifenden Task
	unsigned char (*get_byte)(SEQUENCE_PIPE *this, OS_ID task_id);
	//----------------------------------------------------------------------------
	//Übergibt ein Byte in die Pipe
	//[this]... this-Pointer
	//[byte]... Byte, welches in die Pipe geschoben werden soll
	void (*put_byte)(SEQUENCE_PIPE *this, unsigned char byte);
	//----------------------------------------------------------------------------
	//Melde Task, der sich durch die übergebene Task-Id identifiziert, bei der Pipe an.
	//[this]... this-Pointer
	//[task_id]... Id des Task, der sich bei der Pipe anmelden soll.
	void (*register_task)(SEQUENCE_PIPE *this, OS_ID task_id);
	//----------------------------------------------------------------------------
	//Retourniere letzten aufgetretenen Fehler
	//[this]... this-Pointer
	PIPE_ERROR (*get_last_error)(SEQUENCE_PIPE *this);
```

```
};

//-------------------------------------------------------------------------------
//Initialisiere statische Pipe
//[backfield]... Backfield wo Arbeitsdaten abgelegt werden
//[backfield_max_len]... Maximale Länge des Backfields
//Retourniert Referenz auf initialisierten Key-Value-String oder NULL im Fehlerfall
extern SEQUENCE_PIPE* create_pipe(unsigned char* backfield, unsigned short backfield_max_len);

#endif //SEQUENCE_PIPE_H
```

Quellcode-C-Datei der spezialisierten Pipe (siehe Kapitel 3.3.3.3):

```
/*
Sequence-Pipe für Inter-Programmablauf-Kommunikation
*/
#include "sequence_pipe.h"
//============================================================================
//defines
#define MUTEX_TIMEOUT   3000 //[msec]
//============================================================================
//privates
//----------------------------------------------------------------------------
//Retourniert TRUE wenn wenn mindestens ein Byte in der Pipe ist.
//[this]... this-Pointer
//[task_id]... Task-Id vom zugreifenden Task
static unsigned char this_byte_rdy(SEQUENCE_PIPE *this, OS_ID task_id)
{
        return (this->pipe_in_ptr != this->pipe_out_ptrs[task_id]);
}
//----------------------------------------------------------------------------
//Retourniert das nächste Byte aus der Pipe
//[this]... this-Pointer
//[task_id]... Task-Id vom zugreifenden Task
static unsigned char this_get_byte(SEQUENCE_PIPE *this, OS_ID task_id)
{
        unsigned char byte = 0;
        while (!this->byte_rdy(this, task_id)) os_yield();
        byte = this->pipe_buffer[this->pipe_out_ptrs[task_id]++];
        this->pipe_out_ptrs[task_id] &= PIPE_BUFFER_MASK;
        return byte;
}
//----------------------------------------------------------------------------
//Übergibt ein Byte in die Pipe
//[this]... this-Pointer
//[byte]... Byte, welches in die Pipe geschoben werden soll
static void this_put_byte(SEQUENCE_PIPE *this, unsigned char byte)
{
        if (os_mut_wait(this->mutex, MUTEX_TIMEOUT) != OS_R_TMO) {
                this->pipe_buffer[this->pipe_in_ptr++] = byte;
                this->pipe_in_ptr &= PIPE_BUFFER_MASK;
                os_mut_release(this->mutex);
        }
        else
                this->last_error = ERR_MUTEX_TIMEOUT;
}
//----------------------------------------------------------------------------
//Melde Task, der sich durch die übergebene Task-Id identifiziert, bei der Pipe an.
//[this]... this-Pointer
//[task_id]... Id des Task, der sich bei der Pipe anmelden soll.
static void this_register_task(SEQUENCE_PIPE *this, OS_ID task_id)
{
        if (task_id >= MAX_OUT_PTR_COUNT)
                this->last_error = ERR_TASK_ID_OVERFLOW;
        else
                this->pipe_out_ptrs[task_id] = 0;
}

//----------------------------------------------------------------------------
//Retourniere letzten aufgetretenen Fehler
//[this]... this-Pointer
static PIPE_ERROR(*get_last_error)(SEQUENCE_PIPE *this)
{
        return this->last_error;
}
//============================================================================
//publics
//----------------------------------------------------------------------------
//Initialisiere statische Pipe
//[backfield]... Backfield wo Arbeitsdaten abgelegt werden
//[backfield_max_len]... Maximale Länge des Backfields
//Retourniert Referenz auf initialisierten Key-Value-String oder NULL im Fehlerfall
SEQUENCE_PIPE* create_pipe(unsigned char* backfield, unsigned short backfield_max_len)
{
```

```
        if (sizeof(SEQUENCE_PIPE) > backfield_max_len) return NULL;
        SEQUENCE_PIPE *this = (SEQUENCE_PIPE *)backfield;
        this->pipe_in_ptr = 0;
        this->last_error = ERR_NONE;
        os_mut_init(this->mutex);
        this->byte_rdy = this_byte_rdy;
        this->get_byte = this_get_byte;
        this->put_byte = this_put_byte;
        this->register_task = this_register_task;
        this->get_last_error = this_get_last_error
        return this;
}
```

Quellcode eines Programmablaufes

```
/*
Programmablauf-Template

Author: Ing. Ehrengruber Manfred
Date: 2016-08-10
*/

#include "task_manager.h"
#include "sequence_pipe.h"

//============================================================================
//defines
#define WAIT_TIME       3000 //[msec]

#define START_WITH_STATE1       1       //Command-Byte für's Starten von SM_STATE1

//============================================================================
//private fields
static unsigned char this_is_intialized = FALSE;
static SEQUENCE_PIPE *this_pipe = NULL;
static OS_ID this_task_id = 0;

//----------------------------------------------------------------------------
//Main-Task: Haupt-Programmablauf
void main_task(void)
{
        SM_STATE state = SM_IDLE;

        this_pipe->register_task(this_pipe, this_task_id);      //Task in Pipe anmelden

        while (TRUE)
        {
                //State-Machine für den Ablauf
                switch (state)
                {
                case SM_IDLE:

                        if (this_pipe->bye_rdy(this_pipe))
                        {
                                if (this_pipe->get_byte(this_pipe) == START_WITH_STATE1)
                                        state = SM_STATE1;
                        }

                        break;
                case SM_STATE1:
                                        //ToDo: Ablauf für State 1 verfassen
                        break;

                case SM_STATE2:
                                        //ToDo: Ablauf für State 2 verfassen
                        break;

                        //ToDo: Hier folgend weitere States ensprechend implementieren.
                }
        }
}

//----------------------------------------------------------------------------
//initializes this object
int init(SEQUENCE_PIPE *pipe)
{
        this_pipe = pipe;
        this_task_id = os_task_create(main_task);       //Task erstellen
        this_is_intialized = TRUE;
}
```

Anlagen, Teil 2

In diesem Anlagenteil ist der Quellcode vom Kapitel 3.4: „Herausforderung - Konfigurationssystem“ untergebracht.

Quellcode-Header-Datei des ParProviders (siehe Kapitel 3.4.2)

```
//---------------------------------------------------------------------
//Daten-Typen
//unteres Nibble ist Datenlänge in Bytes
//oberes Nibble ist Datentyp-Identifikation
typedef enum
{
        CHAR = 0x01,
        UCHAR = 0x11,
        SHORT = 0x22,
        USHORT = 0x32,
        LONG = 0x44,
        ULONG = 0x54,
        FLOAT = 0x64,
        ULONGLONG = 0x78

}PAR_DATA_TYPE;

//---------------------------------------------------------------------
//User-Level-Definitionen
#define UL_USER         0
#define UL_TECH         1
#define UL_GOD          2

//Parameter-Member-Attributs-Entität
typedef struct
{
        char *id;                       // Member-Id: eindeutiges Schlüsselwort
                                                // zur Identifikation des Parameters
        char *group_id;         // Gruppen-Id: eindeutiges Schlüsselwort
                                                // zur Bildung von Member-Gruppen

        signed long long default_value; // Defaultwert
        signed long long min_value;             // Minimalwert
        signed long long max_value;             // Maximalwert
        unsigned short user_level;              // User Level
        PAR_DATA_TYPE data_type;        // Datentyp der Daten
        unsigned long data_addr;        // Relative Adresse der Daten-Member.

}PAR_MEMBER_ENTITY;

//Parameter-Node-Entität
typedef struct
{
        char *id;                       // Node-Id: eindeutiges Schlüsselwort
                                                // zur Identifikation des Knotens
        char *parent_id;// Node-Id des Vater-Knotens:
                                                // dadurch kann baumartige Struktur
                                                // erstellt werden
        unsigned short user_level;              // User Level
        unsigned long data_addr;        // Relative Adresse der Daten-Member.

}PAR_NODE_ENTITY;

//Parameter-Relation-Entität
typedef struct
{
        char *node_id;          // Node-Id: eindeutiges Schlüsselwort
                                                // zur Identifikation des Knotens
        char *member_group;     // Gruppen-Id: eindeutiges Schlüsselwort
                                                // von der Gruppe die der Node zugeordnet
                                                // werden soll
}PAR_RELATION_ENTITY;
```

```
//Parameter-Dataset
typedef struct
{
        const PAR_MEMBER_ENTITY *member_table;    //Parameter-Tabelle
        const PAR_NODE_ENTITY *node_table;              //Knoten-Tabelle
        const PAR_RELATION_ENTITY *relation_table; //Relation-Tabelle

}PAR_DATASET;



extern const PAR_DATASET par_dataset;
```

Quellcode-C-Datei des ParDataSets (siehe Kapitel 3.4.2.2)

```
#include "ParProvider.h"

unsigned long par[256];

static const PAR_MEMBER_ENTITY par_member_table[] = {

//_____________________________________________________________________________________________
//      id, group_id, default_value, min_value, max_value, user_level, data_type, data_addr
//
// GENERAL CONFIG.
"MasterTimeout", "TimingsConfig", 10, 1, 100, UL_USER, LONG, ((unsigned long)(&par[0] - &par[0])),
"EnablePrintOut", "GeneralConfig", 0, 0, 1, UL_USER, LONG, ((unsigned long)(&par[1] - &par[0])),

NULL, NULL, 0, 0, 0, UL_GOD, LONG, 0,    //last entry
};

static const PAR_NODE_ENTITY par_node_table[] = {

//_____________________________________________________________________________________________
//      id, parent_id, user_level, data_addr
//
"Parameter", NULL, UL_USER,     0,
"General", "Parameter", UL_USER, ((unsigned long)&par[0]),
"General2", "Parameter", UL_USER, ((unsigned long)&par[32]),
"Timings", "Parameter", UL_USER,((unsigned long)&par[0]),
NULL, NULL, UL_GOD, 0,  //last entry
};

static const PAR_RELATION_ENTITY par_relation_table[] = {

//_____________________________________________________________________________________________
//      node_id, member_group
//
// GENERAL CONFIG.
"General", "GeneralConfig",
"General2", "GeneralConfig",
"Timings", "TimingsConfig",

NULL, NULL,     //last entry
};


const PAR_DATASET par_dataset = {
        par_member_table,
        par_node_table,
        par_relation_table,
};
```

Quellcode-Header-Datei des Dictionary (siehe Kapitel 3.4.2.5)

```
/*
E.M. Dictionary

Author: Ing. Ehrengruber Manfred
Date: 2016-08-15
*/

//============================================================================
//defines

#define MAX_KEY_LENGTH 32

struct node
{
        struct node *next;
        char key[MAX_KEY_LENGTH];
        unsigned long value;
};


//============================================================================
//interrupt routines


//============================================================================
//publics

//----------------------------------------------------------------------------
//Set Dictionary-Entry
extern struct node *dict_add_entry(char *key, unsigned long value);

//----------------------------------------------------------------------------
//Get Dictionary-Entry
extern struct node *dict_get_entry(char *key);

//----------------------------------------------------------------------------
//initializes this object
extern void dict_init();
```

Quellcode-C-Datei des Dictionary (siehe Kapitel 3.4.2.5)

```
/*
E.M. Dictionary

Author: Ing. Ehrengruber Manfred
Date: 2016-08-15
*/

#include "EMDictionary.h"


//============================================================================
//defines

#define HASH_TABLE_LENGTH       10000



//============================================================================
//private fields

static struct node *this_hashtab[HASH_TABLE_LENGTH];

unsigned long list_entry_ctr = 0;
double list_entry_ctr_average = 0;

//============================================================================
//interrupt routines


//============================================================================
//privates

//---------------------------------------------------------------
//Calculates hash of str
//[str]... string of which the content will be calculated.
static unsigned long calculate_hash_table_idx(char *str)
{
        unsigned long hashval = 0;
        for (hashval = 0; *str != '\0'; str++)
                hashval = *str + 33 * hashval;
        return hashval % HASH_TABLE_LENGTH;
}

//---------------------------------------------------------------------------
//Find Dictionary-Entry
static struct node *find_entry(char *key)
{
        struct node *np = NULL;
        struct node *r_np = NULL;
        unsigned long hashtab_idx = calculate_hash_table_idx(key);
        list_entry_ctr = 0;
        for (np = this_hashtab[hashtab_idx]; np != NULL; np = np->next)
        {
                list_entry_ctr++;
                if (strcmp(key, np->key) == 0)
                {
                        if (r_np == NULL)
                                r_np = np;
                        //return np; //found
                }
        }
        list_entry_ctr_average += list_entry_ctr;
        //T E S T return NULL; //not found
        return r_np;    //T E S T
}

//============================================================================
//publics

//---------------------------------------------------------------------------
//Set Dictionary-Entry
```

```
struct node *dict_add_entry(char *key, unsigned long value)
{
        struct node *np = NULL;
        unsigned long hashtab_idx = 0;
        if ((np = find_entry(key)) == NULL)
        {
                //add new list entry
                np = (struct node *) malloc(sizeof(*np));
                if (np == NULL)
                        return NULL;
                strcpy(np->key, key);

                //... and put it to the hash-table
                hashtab_idx = calculate_hash_table_idx(key);
                np->next = this_hashtab[hashtab_idx];
                this_hashtab[hashtab_idx] = np;
        }
        np->value = value;
        return np;
}

//------------------------------------------------------------------------------
//Get Dictionary-Entry
struct node *dict_get_entry(char *key)
{
        return find_entry(key);
}

//------------------------------------------------------------------------------
//initializes this object
void dict_init()
{

}
```

Anlagen, Teil 3

In diesem Anlagenteil ist das Messergebnis von der Untersuchung der Effizienz von Dictionaries (siehe Kapitel 3.4.2.5) untergebracht:

Vorgang:
Es wurden 172 Einträge, wo als Schlüsselwörter die Parameter-IDs aus einem bestehenden Parameter-Dataset (siehe Abbildung 19) verwendet wurden, in das Dictionary hinzugefügt. Die mittlere Länge dieser Schlüsselwörter beläuft sich auf 18 Zeichen.

```
void dict_add_test()
{
        unsigned long idx = 0;

        dict_add_entry("mss_machine_type", idx++);
        dict_add_entry("rear_load_enable", idx++);
        dict_add_entry("start_decision", idx++);
        dict_add_entry("char_display_nr", idx++);
        ...
        ...
```

Dann wurden alle 172 Einträge aus dem Dictionary wieder ausgelesen.

```
void dict_get_test()
{
        unsigned long value = 0;
        value = dict_get_entry("mss_machine_type")->value;
        value = dict_get_entry("rear_load_enable")->value;
        value = dict_get_entry("start_decision")->value;
        value = dict_get_entry("char_display_nr")->value;
        ...
        ...
```

Die Laufzeit des Hinzufügens und Auslesens aller Dictionary-Einträge wurde mittels Oszilloskop gemessen.

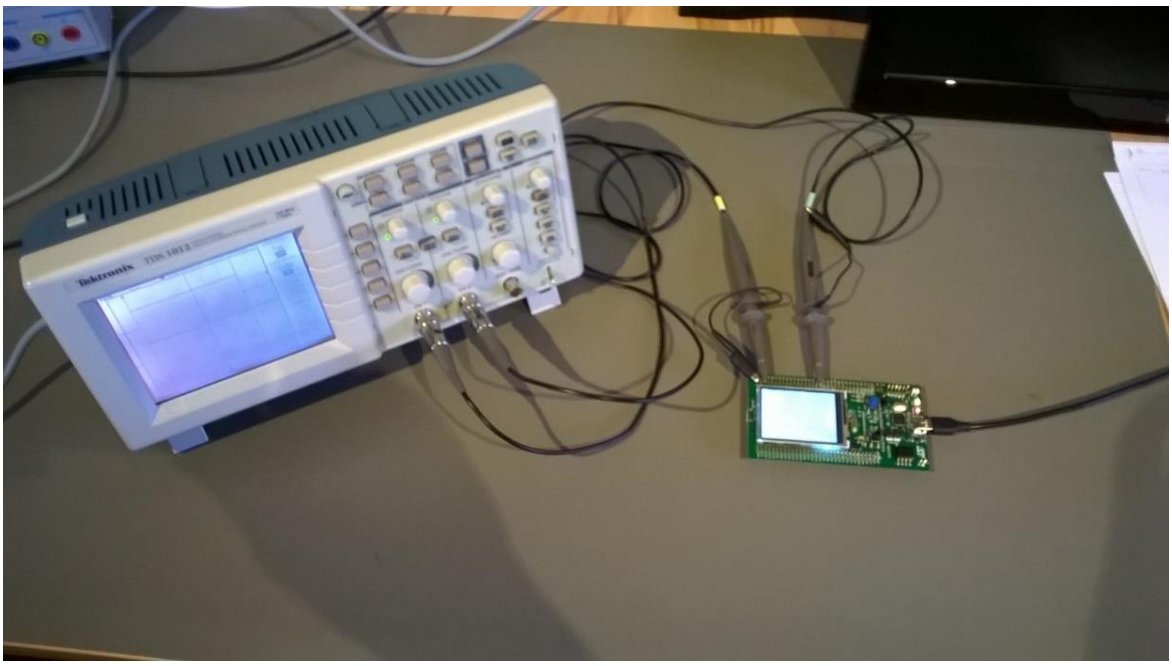

Abbildung 39: Messaufbau - empirisches Ermitteln der Effizienzdaten (eigene Fotografie)

Die Messungen wurden mit unterschiedlichen Hash-Tabellen-Größen durchgeführt.

Hinzufügen von 172 Einträge in das Dictionary:

Größe der Hash-Tabelle	**Laufzeit [msec]**	**Speicherverbrauch [Byte] (RAM)**	**Mittlere Länge der verketteten Listen**
1	44,0	6884	172,0
10	10,7	6920	18,31
100	7,3	7280	2,73
1000	7,0	10880	1,22
10000	6,9	46880	1,02

Tabelle 3: Messwerte - Hinzufügen von Einträgen in das Dictionary

Auslesen von 172 Einträge vom Dictionary:

Größe der Hash-Tabelle	**Laufzeit [msec]**	**Speicherverbrauch [Byte] (RAM)**	**Mittlere Länge der verketteten Listen**
1	43,6	6884	172,0
10	10,5	6920	18,31
100	7,2	7280	2,73
1000	6,8	10880	1,22
10000	6,7	46880	1,02

Tabelle 4: Messwerte - Auslesen von Einträgen vom Dictionary

Diese Messungen wurden in einem authentischen eingebetteten System, wo das Konfigurationssystem (siehe Kapitel 3.4) zukünftig eingesetzt wird, durchgeführt. Empirisch konnte festgestellt werden, dass für 172 Einträge die Hash-Tabelle 1000 Einträge groß sein sollte. Aus den Messdaten lässt sich schließen, dass sich das Auslesen eines Eintrages von einem Dictionary mit einer Hash-Tabellen-Größe von 1000 im Mittel auf 40µsec beläuft.
Das Auslesen eines Parameters und deren Parameterbezeichnungen erfordert zwei Abfragen der Dictionaries. Dadurch ergibt sich im Mittel eine Rechenzeit von 80µsec. Dazu kommt die Rechenzeit für das Ermitteln der Parameter-Werte aus den Parameter-Datasets, sodass die Parameterdaten eines Parameters in ca. 100µsec vollständig ermittelt werden können. Im realen Betrieb dieses eingebetteten Systems werden meist höhere Systemtaktraten (168 MHz) verwendet, womit die Rechenzeit entsprechend vermindert wird. Bei zeitkritischen Operationen können die Parameter am Beginn der Operation in eine lokale Variable ausgelesen und später in der Operation verwendet werden.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Wallern, den 05.10.2016

Manfred Ehrengruber