Dokumentation im Embedded Software Process

Assignment zum Modul:

Embedded Software Development (EBS43)

21.09.2020

Vladimir Zhelezarov

.....

Studiengang: Digital Engineering und angewandte Informatik - Bachelor of Engineering

AKAD University

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Eingebettete Systeme sind allgegenwärtig. Rund 98 Prozent aller hergestellten Prozessoren sind für eingebettete Systeme[1]. Sie finden verbreitete Anwendung bei der industriellen Kontrolle und Automatisierung, Unterhaltungselektronik, Internet der Dinge, Automobilen, Medizin sowie bei vielem mehr. Mindestens die Hälfte dieser Projekte erfüllen eine wichtige bis kritische Funktion[2]. Somit ist ihre Entwicklung mit den Prozessen die dabei ablaufen, von besonderer Bedeutung in der Software-Welt.

Die vorliegende Arbeit untersucht die Dokumentation und ihre Bedeutung bei der Entwicklung und Benutzung von eingebetteten Systemen. Mit der immer zunehmenden Komplexität und Wichtigkeit dieser Systeme steigen auch die Anforderungen an der Dokumentation die sie für ihre Benutzung oder Weiterentwicklung brauchen.

Eine Definition der eingebetteten Systeme ist notwendig um zwischen diesen und anderen Software-Systemen zu unterscheiden. Auch wenn manche Eigenschaften ähnlich zur Applikationssoftware sind, haben die eingebetteten Systeme auch einzigartige Besonderheiten.

Die Dokumentation entsteht normalerweise als Teil des Entwicklungsprozesses. Um besser die erzeugten Dokumentenartefakte zu verstehen, ist es notwendig die Prozessprinzipien in Betracht zu nehmen. Die Art und der Umfang der erstellten Dokumente hängen auch wesentlich vom Projekt und von den Gegebenheiten ab. Es sind manche Gemeinsamkeiten mit der Software-Welt zu entdecken, sowie auch Unterschiede. Bei der Beschreibung eines Software-Produkts sind in den meisten Fällen mehrere Dokumente notwendig. Diese betrachten wir nach Abstraktionsebene, Sicht, Zielgruppe sowie auch nach anderen Kriterien.

Um ein Überblick zu gewinnen, wie das ganze in der Praxis umgesetzt wird, analysieren wir ausgewählte Open-Source Projekte. Dafür werden zusätzliche Metriken für den Umgang mit der Dokumentation eingeführt, die Ergebnisse werden verglichen und Schlussfolgerungen gezogen. Es wird zwischen Entwickler- und Benutzerdokumentation unterschieden. Die Untersuchung hat als Ziel eine Einschätzung zu machen, wie viel Wert auf die Erstellung und Wartung von Entwicklerinformation in den ausgewählten Projekten gestellt ist.

[1] Vgl. Barr; Massa (2006), S.10
[2] Vgl. Aspencore (2019), S.7-8

2 Grundlagen

2.1 Begriffe

Eingebettete Systeme sind Rechenmaschinen, die für den Anwender weitgehend unsichtbar in einem elektrischen Gerät „eingebettet“ sind.[1] Ein eingebettetes System ist typisch konstruiert um eine bestimmte Aufgabe auszuführen.

Eine Besonderheit dieser Systeme ist, dass sie mit ihrer Umwelt physikalisch interagieren[2]. Dabei ist ihre zeitliche Reaktion auf die externen Ereignisse wichtig. Somit sind eingebettete Systeme auch Echtzeitsysteme - Systeme, dessen Funktion nicht nur von den gelieferten Ergebnissen abhängt, sondern auch von der Zeit in der das System diese Ergebnisse liefert. Je nach dem wie kritisch oder gefährlich eine zeitliche Verzögerung ist, werden diese Systeme in weiche und harte Echtzeitsysteme geteilt - bei weichen Systemen degradiert nur die Qualität der Ergebnisse und bei harten Systemen kann es zu Schäden und/oder Gefahren für Menschen oder Eigentum kommen[3].

Die Software für eingebettete Systeme wird auch als Firmware bezeichnet - um zu betonen, dass diese Software nicht so einfach zu ändern ist, also nicht so „soft“ wie andere Software[4]. Eine andere mögliche Abgrenzung ist den Begriff Firmware nur für die Softwareschicht zwischen der Hardware und dem Betriebssystem zu verwenden. Weil beide schwer änderbar sind, wenn das System schon im Betrieb ist, benutzt diese Arbeit den Begriff Firmware für alle Software auf dem eingebetteten System.

Bei der Entwicklung von eingebetteten Systemen entsteht auch die dazugehörige technische Dokumentation. Als solche sind alle Dokumente gemeint, die dazu dienen das System in seinen Eigenschaften zu beschreiben[5].

Eine sehr verbreitete Form der Entwicklung folgt das agile Entwicklungsmodell - darunter ist ein Entwicklungsprozessmodell zu verstehen, bei dem die Menschen und deren Interaktionen einen höheren Wert haben, als Planung, Prozesse und Dokumentation[6].

Eine Hilfe bei der Erstellung von Dokumentationen bieten Software-Werkzeuge die dem Entwickler etwas von der Arbeit abnehmen. Als ein bekannter Vertreter ist dabei DoxyGen[7] zu nennen - ein Tool, das aus den Kommentaren im Code automatisch übersichtliche HTML-Dateien erstellt.

[1] Gessler (2014), S.8
[2] Vgl. Lee; Seshia (2016), S. X
[3] Vgl. Sommerville (2011), S.538
[4] Vgl. Lee; Seshia (2016), S.241
[5] Kothes (2011), S.2
[6] Vgl. Beck et al. (2001), Internetquelle
[7] https://www.doxygen.nl, (Zugriff am 21.09.2020)

2.2 Anforderungen an der Dokumentation von eingebetteten Systemen

Es gelten die grundsätzlichen Anforderungen an der Dokumentation von IT-Projekten[1]:

- Vollständigkeit; Einheitlichkeit, Strukturiertheit und Übersichtlichkeit; Benutzbarkeit und Anschaulichkeit; Änderbarkeit und Anpassungsfähigkeit; Widerspruchsfreiheit; Aktualität und Wirtschaftlichkeit der Erstellung.

Die Normungsinstitute ISO/IEC und IEEE pflegen internationale Normen, deren Befolgung bei der Erstellung von Software-Dokumentationen empfohlen ist. Darunter zählen IEEE 1063, ISO/IEC 6592, ISO/IEC 9294, ISO/IEC 18019, ISO/IEC 912 und ISO/IEC 26514[2].

Zusätzlich ist die eingebettete Software als Bestandteil des Produkts oder Geräts zu betrachten, indem sie enthalten ist[3]. Daraus folgt, dass für sie vorrangig alle für das Produkt anwendbaren Richtlinien, Gesetzte und Verordnungen gelten.

2.3 Entwicklungsprozess nach dem agilen Modell

Die Probleme mit der fehlenden Flexibilität und mit der Belastung der Entwickler bei den klassischen, spezifikationsorientierten Entwicklungsmodellen haben dazu beigebracht, dass neuere, weniger formale, "agile" Modelle immer beliebter werden[4]. Das Ziel bei diesen ist den Entwicklern ein besseres Arbeitsklima zu bieten, was zu mehr Motivation und dadurch auch mehr Produktivität und Qualitätsbewusstsein führen kann. Das agile Manifest fasst die Grundprinzipien zusammen[5]:

Hoher Wert		*Niedriger Wert*
Individuen und Interaktionen	>	Prozesse und Werkzeuge
Funktionierende Software	>	umfassende Dokumentation
Zusammenarbeit mit dem Kunden	>	Vertragsverhandlung
Reagieren auf Veränderung	>	Befolgen eines Plans

Tabelle 1: Agile Prinzipien nach dem agilen Manifest

Ein typisch agiler Ansatz ist die kontinuierliche Integration von Software - Continuous Integration (CI). Bei diesem Vorgehen laden die Entwickler ihren Code mehrfach pro Tag

[1] Wieczorrek; Mertens (2011), S.307-308
[2] Kothes (2011), S.40
[3] Vgl. Kothes (2011), S.40-41
[4] Vgl. Gessler (2014), S.91
[5] Vgl. Beck et al. (2001), Internetquelle

im Projekt hoch, was automatisierte Tests auslöst. Dieses Vorgehen führt zur Reduktion von Fehlern, qualitative Software und schnellere Arbeitszyklen[1].

Welche konkreten Dokumente die eingebetteten Systeme benutzen, zeigt das nächste Kapitel.

3 Dokumente bei der Entwicklung von eingebetteten Systemen

Die erstellte Dokumentation ist am benutzten Entwicklungs-Modell angelehnt - bei den formellen Modellen sind genaue Dokumente vorgeschrieben, und bei mehr agilen und nichtformellen Projekten ist die Entscheidung über die Menge und Art der Dokumentation den Entwicklern überlassen. Die Dokumentation muss im Idealfall alle Informationsbedürfnisse der Stakeholder befriedigen. Dafür ist es hilfreich verschiedene Dokumente für alle benötigten Abstraktionsebenen, Sichten und Zielgruppen zu erstellen.

3.1 Von der Spezifikation zum Code

Eine Software-Entwicklung fängt mit der Anforderungsdefinition/Spezifikation an[2]. Das sind die Dokumente, die nach der Absprache mit dem Kunden, Beschreibungen über alle Eigenschaften des zu erstellenden Produkts beinhalten. Um alle vorstellbaren Szenarien abzudecken sind eventuell mehrere Termine und Diskussionen notwendig. Je detaillierter und eindeutiger diese Dokumente sind, desto einfacher wird die folgende Arbeit für die Entwickler. Vernachlässigung in dieser Phase bewirkt massive Code-Änderungen um die entstandenen Fehler zu beseitigen[3]. Darum ist es wichtig möglichst viel Aufmerksamkeit und Gedanken in der Spezifikation zu investieren.

Mit der vorliegenden Spezifikation kann die Design-Phase anfangen. Als erstes wird die Architektur entwickelt - der Hauptrahmen nach dem die Software aufgebaut ist. Die Architektur entscheidet wie die Komponenten strukturiert sind, wie sie miteinander interagieren und bestimmt die Richtung für die folgenden Entwicklungen[4]. Diese Richtung sollte dann vorzugsweise gefolgt werden und Änderungen dabei sind nur in begründeten Fällen zulässig. Als Dokumentenartefakte entstehen dabei ein oder mehrere Dokumente, je nach Umfang des Projekts. Außer Diagrammen, in denen typisch UML mit zahlreichen Darstellungsmöglichkeiten benutzt wird, kommen auch einfache Textbeschreibungen in Frage, je nach dem, wie

[1] Vgl. Fowler (2006), Internetquelle
[2] Vgl. Gessler (2014), S.83
[3] Vgl. Gessler (2014), S.87
[4] Vgl. Goll (2011), S.77

es die Software-Architekten für besser finden, ihre Ideen auszutauschen[1]. Die Kundenwünsche ändern sich mit der Zeit, sowie die Besonderheiten der Umgebung. Deshalb kann davon ausgegangen werden, dass Änderungen bei der Architektur zu einem späteren Zeitpunkt notwendig sein werden. Dabei ist es besonders wichtig diese Entscheidungen und die Gründe dafür zu dokumentieren, damit die weitere Entwicklung nicht ohne klar definierter Architektur ins Chaos gerät. Ein Ansatz dafür ist diese Information in Textform, eventuell mit zusätzlichem Markup, zusammen mit dem Code zu verwalten[2]. So kann die Entwicklungsgeschichte immer schnell verfolgt werden, was zu einem besseren Verständnis vom derzeitigen Stand führt.

Die Architekturentscheidungen setzt der Code um. Die Dokumentation in Form von Kommentaren, die sich direkt im Code befindet widmet sich an den kleinsten Details, die üblicherweise in den nebenstehenden Zeilen folgen.

3.2 Klassifikation nach Sichten

Eventuell ist nur bei kleinen Projekten möglich, dass ein einzelnes Dokument ausreichend Information über alle Aspekte der Software liefern kann. In den meisten Fällen sind Darstellungen aus mehreren Sichten notwendig um gutes Verständnis über dem Design zu gewährleisten. Die möglichen Sichten richten sich nach dem Standpunkt der Betrachtung. Grundsätzlich lassen sich die Sichten in folgende Gruppen einordnen[3]:

- Statischer Standpunkt: in dieser Sicht zeigt sich die Struktur der Software, der Aufbau und die Komponente. Diese Sicht bietet keine Information über Zusammenwirkung oder zeitliches Verhalten;
- Dynamischer Standpunkt: dieser stellt das System im Betrieb dar. Die Sicht zeigt wie die Komponente miteinander interagieren und wie die Kommunikation mit dem Benutzer oder der Umgebung abläuft. Es können Dokumente erstellt werden, die einen bestimmten Zeitpunkt abbilden, oder solche die einen ganzen Ablauf demonstrieren;
- Verteilungs-Standpunkt: wenn das System aus mehreren Hardware-Komponenten besteht ist es wichtig zu wissen, wie die Software auf diesen verteilt wird, und das ist die Aufgabe dieser Sicht.

3.3 Klassifikation nach Zielgruppe

Die Projekt-Stakeholder haben unterschiedliche Bedürfnisse an Information.

[1] Vgl. Bass et al. (2015), S.330
[2] Vgl. Nygard (2011), Internetquelle
[3] Vgl. Behrens et al. (2006), S.36-37

- Die Entwickler benötigen ausführliche Information über der Architektur, dem Aufbau und alle interne Einzelheiten vom Code, damit sie auch weiterarbeiten können;
- Die Benutzer interessieren sich wenig über den internen Aufbau und fokussieren sich meistens auf den Funktionen vom System und wie sie damit kommunizieren. Die Benutzer selber können auch weiter unterteilt werden in Administratoren, die sich um Einstellungen und Kontrolle kümmern und andere Benutzer, die das System nur nutzen;
- Zusätzliche Projektdokumentation ist oftmals auch benötigt. Für den internen Gebrauch können Auswertungen, Planungen und Statistiken nützlich sein, und externe Interessanten wie z.B. bei Audits können zusätzliche spezifische Dokumente verlangen.

3.4 Klassifikation nach Technologie und Format

Wie unten bei der Untersuchung festgestellt wird, sind bei Open-Source Projekten die Dokumente ausschließlich in Web-freundlichen Formaten angeboten, wie Text und Grafiken in html- oder md-Format. Die Beschreibung dabei muss nicht unbedingt nur aus Text bestehen. Hinsichtlich eingebettete Systeme existieren spezifische zusätzliche Darstellungsmöglichkeiten, wie:

- Zustandsautomaten - ein mathematisches Modell für die Beschreibung von Schaltungen mit Speicherglieder[1];
- Petri-Netze - für die Beschreibung von verteilten, nebenläufigen Prozessen[2];
- UML - eine Hochsprache mit vielen Ausprägungen für die grafische Darstellung von Architektur, Struktur und dynamisches Verhalten. Speziell für eingebettete Systeme ist das Sprachen-Profil MARTE vorgesehen[3];
- Hardware-Beschreibungssprachen - diese benutzen eine Programmiersprachen-ähnliche Definition, woraus Hardware in programmierbaren (FPGA) oder festverdrahteten Schaltungen (ASIC) erstellt wird[4].

3.5 Typische Dokumente für Eingebettete Software

In der Praxis sind oftmals folgende Dokumente vertreten:

[1] Vgl. Gessler (2014), S.134
[2] Vgl. Gessler (2014), S.120-121
[3] https://www.omg.org/spec/MARTE/About-MARTE/, Zugriff am 21.09.2020
[4] Vgl. Gessler (2014), S.258

- Das Datenblatt beinhaltet die Spezifikation in Einzelheiten für ein Produkt, inklusive sein Aufbau, Funktionen und Schnittstellen. Dieses Dokument bezieht sich normalerweise auf eine Hardware-Komponente wie z.B. ein Mikrocontroller, wird aber hier vorgestellt aufgrund der engen Verzahnung zwischen Software und Hardware in den eingebetteten Systemen. Das Datenblatt wird vom Hersteller kostenlos bereitgestellt und kann als Ausgangspunkt dienen, um ein spezifisches Produkt mit anderen zu vergleichen. Öffentliche Datenbanken mit Datenblättern helfen bei der Suche und Auswahl von Komponenten[1];

- Mit Software Design Document, manchmal auch unter dem Namen Firmware Design Document, wird die Architekturbeschreibung von einer bestimmten Software bezeichnet;

- Reference Manual ist eine durchsuchbare Liste mit allen möglichen Funktions- oder Klassenaufrufen und deren erwartetes Verhalten. Das Manual hat typischerweise die Form einer oder mehrerer Tabellen, wo unter anderem Beschreibung, Parameter und Rückgabewerte der Funktionen angegeben sind;

- Programming Manuals und Developer-Guides sind Anweisungen und Anleitungen für Programmierer oder Entwickler, die mit dem Produkt durch ihrer Software kommunizieren möchten, oder es als Komponente in der Software benutzen wollen;

- Use Cases sind Beispielanwendungen vom Produkt, wobei auch Hinweise für die optimale Funktion sowie für den Umgang mit Problemen angeboten sind. Gegen diese Abläufe wird auch reichlich bei den Tests getestet, um das korrekte Verhalten der Software zu prüfen;

- Lizenzen-Informationen zeigen den vom Entwickler/Hersteller erlaubten Rahmen, in dem sein Produkt benutzt werden darf, inwieweit der Hersteller Haftung für sein Produkt übernimmt, ob eine kommerzielle Benutzung erlaubt ist und andere Aspekte. Es gibt Webseiten die bei der Auswahl von Lizenzen Hilfe anbieten[2];

- Bei Projekten, die Versionierung benutzen, wie das typisch bei Open-Source Projekte ist, sind commit-Messages auch eine Art von Dokumentation. Diese sind kurze Text-Informationen, die jeder Änderung beigefügt werden. Sie beinhalten eine Beschreibung und eventuell auch eine Begründung für die vorliegende Änderung und erzählen dadurch die Geschichte der Entwicklung[3];

[1] https://www.datasheets.com/, Zugriff am 21.09.2020
[2] https://choosealicense.com/, Zugriff am 21.09.2020
[3] https://git-scm.com/docs/git-commit, Zugriff am 21.09.2020

Wenn keine anderen Dokumente vorhanden sind, kann als ein absolutes Minimum eine Readme-Datei dem Code zugefügt werden, in der Informationen über was die Software macht stehen, warum sie existiert, wie sie benutzt werden kann, wo der Benutzer weitere Informationen oder Hilfe findet und wer der Autor und die Mitwirkenden sind[1].

4 Benutzung von Dokumentation in Open-Source Projekten

Wie die vorgestellten Techniken in der Praxis umgesetzt werden, beschreibt dieser Kapitel mithilfe einer Untersuchung mancher Open-Source Projekte. Ohne Anspruch eine komplette Abbildung der Software-Welt in der Praxis zu sein, können diese Projekte als einen guten Startpunkt dienen.

4.1 Objekt und Methodik der Untersuchung

Die vorgestellte Untersuchung basiert auf sieben populäre Open-Source Projekte für eingebettete Software gehosted im Online-Dienst GitHub. Die Entscheidung nach Popularität basiert auf die vergebenen Sterne. Ziel der Untersuchung ist eine Einschätzung über den Stellenwert der Dokumentation bei den untersuchten Projekten.

Bei der Analyse achten wir auf Vorhandensein und Art der Dokumentation. Dabei wird zwischen Entwickler- und Benutzerdokumentation unterschieden. Nach dem Kriterium bei dem sich die Information befindet, unterscheidet die Untersuchung zwischen interner Entwickler-Dokumentation, die praktisch aus den Kommentaren im Code besteht, und externe - die separat aus dem Code gewartet wird, z.B. in einem anderen Pfad, oder auf eigener Webseite im Internet.

Bei der Auswahl von Metriken, haben sich die folgende als geeignet gezeigt:

- *Verhältnis Kommentare/Code:* Die Zahl der Zeilen, die Kommentare beinhalten wird durch die Zahl der Code-Zeilen geteilt. Vor allem bei agilen Projekten können die Kommentare im Code die einzige Dokumentation zur Software sein, wie später unten gezeigt wird;

- *Anzahl der Commits die nur aus kommentarlosem Code bestehen:* Das ist die Anzahl von Commits aus den letzten 20, die nur aus Code bestehen. Es sind also dabei keine Kommentare zu finden. Die benötigte Information wird mithilfe der Software git[2] extrahiert.

[1] https://docs.github.com/en/github/creating-cloning-and-archiving-repositories/about-readmes, Zugriff am 21.09.2020

[2] https://www.git-scm.com/, (Zugriff am 21.09.2020)

Weil bei den meisten Dateien in den untersuchten Projekten ein Header mit Lizenzen-Information vorhanden ist, kann dieser die Ergebnisse verfälschen. Die Lizenz stellt keine nützliche Information dar, was die Verständlichkeit von der Programm-Funktion angeht. Um dieses Problem umzugehen sind für die Untersuchung diese Header aus den Quellendateien entfernt. Die so vereinfachten Dateien analysiert nach Code-Struktur die Software *cloc*[1]. Aufgrund von Übersichtlichkeit sind nur die Dateien der Hauptprogrammiersprache vom Projekt, welche bei den betrachteten Projekten ausschließlich C/C++ ist, dargestellt. Dazu gehören die Quelldateien und die Header-Dateien.

4.2 Arduino AVR core

Arduino AVR core[2] ist die Repository für die Firmware zur Interaktion mit Atmel-Boards für die Entwicklungsplattform Arduino.

Externe Dokumentation ist nirgendwo zu finden, es fehlt sogar die Standard-Datei Readme. Arduino stellt sehr umfangreiche Information auf seine Webseite[3] zur Verfügung, allerdings nur was die Benutzung von der IDE und die Entwicklung von Software für die Plattform angeht. Als Hilfe können die sehr ausführlichen Spezifikationen und Dokumentation dienen, die der Hersteller vom benutzten Mikrocontroller anbietet[4], oder auch die Dokumentation der benutzten Framework „Wiring“[5]. Der Rest an benötigter Information muss der Entwickler aus dem Code entnehmen. Dieser Mangel könnte an der vergleichbar kleinen Größe des Projekts liegen. Es befindet sich keine Angabe zur Benutzung einer CI oder DoxyGen.

4.3 RT-Thread

RT-Thread[6] ist ein Open-Source Betriebssystem für Internet der Dinge. Sehr detaillierte und strukturierte html/md Dokumentation über der Architektur und Benutzeranleitungen sind in extra Repository[7] untergebracht. Noch eine Repository[8] ist für die Spezifikation geplant, diese ist aber zum heutigen Datum noch nicht fertig und besteht nur aus einer leeren Vorlage in chinesisch. Das Projekt benutzt Continuous Integration mit dem Tool Travis. Ein Nachteil ist , dass nicht alle Beispiele oder Seiten auf Englisch sind. DoxyGen wird benutzt.

[1] https://github.com/AlDanial/cloc, (Zugriff am 21.09.2020)
[2] https://github.com/arduino/ArduinoCore-avr, (Zugriff am 21.09.2020)
[3] https://www.arduino.cc/, (Zugriff am 21.09.2020)
[4] https://www.microchip.com/wwwproducts/en/ATmega328, (Zugriff am 21.09.2020)
[5] https://github.com/WiringProject/Wiring, (Zugriff am 21.09.2020)
[6] https://github.com/RT-Thread/rt-thread, (Zugriff am 21.09.2020)
[7] https://github.com/RT-Thread/rtthread-manual-doc, (Zugriff am 21.09.2020)
[8] https://github.com/RT-Thread/rtthread-specification, (Zugriff am 21.09.2020)

4.4 PX4 Autopilot Software

PX4[1] ist eine Autopilot-Software für Dronen. Ausführliche html Dokumentation, inklusive Architekturbeschreibung, Anleitungen und Beispiele sind auf der Webseite der Entwickler zu finden[2]. Das Projekt führt Continuous Integration mit ausführlichen Tests unter Benutzung von verschiedenen Tools durch. DoxyGen wird benutzt.

4.5 Asuswrt-Merlin

Asuswrt-Merlin[3] ist eine erweiterte Firmware für Router von ASUS. Eine etwas begrenzte Dokumentation ist auf der GitHub-Wiki Seite des Projekts zu finden[4]. Die Betonung dabei ist bei der Kompilierung und Benutzung der Software. Es befinden sich wenige bis keine Angaben zur Architektur und Entwicklung. Es sind auch keine Zeichen von der Benutzung einer CI feststellbar. DoxyGen wird benutzt.

Aufgrund der Übersichtlichkeit, wurden bei diesem Projekt nur die Dateien aus dem Pfad release/src analysiert.

4.6 Tasmota

Tasmota[5] ist eine alternative Firmware für ESP8266 Wireless-Modulen. Ausführliche Benutzerinformation mit Anleitungen, Tutorials und Beispiele ist vorhanden[6]. Entwicklerinformation ist aus dem Code zu entnehmen. Das Projekt benutzt Continuous Integration mit Travis und anderen Tools. Weil das Projekt auf die Plattform „PlatformIO“ aufbaut, kann die dort vorhandene Entwickler-Information zur Hilfe kommen[7]. DoxyGen wird benutzt.

4.7 Quantum Mechanical Keyboard Firmware

Quantum[8] ist eine alternative Firmware für mechanische Tastaturen, die Atmel AVR oder ARM Steuerung benutzen. Eine ausführliche html Dokumentation ist auf eigener Webseite vorhanden[9]. Dabei sind nicht nur Benutzeranleitungen mit Beispielen und Tutorials, sondern auch gut strukturierte und detaillierte Entwickler-Information angeboten. Continuous Integration mit Travis und Code-Rechtschreibung Prüfung (Lint) werden beim Zufügen von Code ausgeführt. DoxyGen wird benutzt.

[1] https://github.com/PX4/Firmware, (Zugriff am 21.09.2020)
[2] https://dev.px4.io/master/en/index.html, (Zugriff am 21.09.2020)
[3] https://github.com/RMerl/asuswrt-merlin.ng, (Zugriff am 21.09.2020)
[4] https://github.com/RMerl/asuswrt-merlin.ng/wiki, (Zugriff am 21.09.2020)
[5] https://github.com/arendst/Tasmota, (Zugriff am 21.09.2020)
[6] https://tasmota.github.io/docs/, (Zugriff am 21.09.2020)
[7] https://docs.platformio.org/en/latest/, (Zugriff am 21.09.2020)
[8] https://github.com/qmk/qmk_firmware, (Zugriff am 21.09.2020)
[9] https://docs.qmk.fm, (Zugriff am 21.09.2020)

4.8 Klipper

Klipper[1] ist eine Firmware für 3D-Drucker. Eine html/md Benutzer- und Entwicklerdokumentation ist in der Repository auffindbar und beinhaltet sehr detaillierte Informationen für beide interessierte Gruppen. Das Projekt benutzt Continuous Integration mit Travis. Keine Angabe der Benutzung von DoxyGen.

4.9 Analyse und Schlussfolgerungen

Die Ergebnisse sind in der folgenden Tabelle zusammengefasst:

Merkmal/Projekt	Arduino AVR Core	RT-Thread	PX4 Autopilot	Asuswrt-Merlin	Tasmota	Quantum	Klipper
Benutzer-Dokumentation	o.A.	HTML/MD, zahlreich	HTML, zahlreich	HTML/MD, zahlreich	HTML, zahlreich	HTML, zahlreich	HTML/MD, zahlreich
Externe Entwickler-Dokumentation	o.A.	HTML/MD, zahlreich	HTML, zahlreich	o.A.	o.A.	HTML, zahlreich	HTML/MD, zahlreich
DoxyGen	o.A.	ja	ja	ja	ja	ja	o.A.
Continuos Integration	o.A.	Travis	mehrere Tools	o.A.	Travis u.a.	Travis mit Linter	Travis
Kommentare/ Code	0,39	0,43	0,32	0,22	0,21	0,37	0,10
Commits ohne Dokumentation	15	16	15	11	13	7	8

Tabelle 2: Benutzung von Dokumentation in Open-Source Projekte - Übersicht

Daraus lassen sich folgende Schlüsse ziehen:

- Alle Projekte, bis auf Arduino AVR zeigen ähnliche Eigenschaften. Arduino AVR ist auch das kleinste Projekt in der Gruppe mit unter 300 Code-Dateien (siehe Anhang 1). Eventuell ist genau das der Grund, warum das Projekt so stark auffällt. Bei der kleinen Größe werden keine Schwierigkeiten beim Verständnis vom sonst gut kommentiertem Code erwartet;
- DoxyGen und Continuous Integration sind praktisch die Regel. Continuous Integration gibt zusätzliche Sicherheit in der Qualität des Codes, wenn die Tests erfolgreich abschließen. DoxyGen ist eine Hilfe bei der Arbeit mit dem Code, um Information über Funktionen oder benötigte Parameter zu bekommen;

[1] https://github.com/KevinOConnor/klipper, (Zugriff am 21.09.2020)

- Das Projekt Klipper fällt auf mit besonders wenigen Kommentare im Code. Dafür aber, auch wenn das Projekt das zweitkleinste aus der Gruppe ist, ist eine sehr ausführliche externe Dokumentation für die Entwickler vorhanden. Sie gleicht das schlechte Kommentare/Code Verhältnis gut aus;

- Die zweite von der Untersuchung eingeführte Metrik - Commits ohne Dokumentation - ist besonders klein bei Quantum und Klipper. Interessant ist das gerade Klipper sehr wenige Kommentare im Code insgesamt hat. Eventuell sind genau die zur Zeit der Untersuchung analysierten Commits nicht typisch für die übliche Code-Struktur. Die Metriken bei Quantum stimmen überein - der Code ist ausführlich kommentiert genauso wie die letzten Commits;

- Eine andere Feststellung aus den Ergebnissen ist die Betonung auf Benutzerdokumentation, was logisch erscheint. Es wird von den Benutzern nicht erwartet, dass sie sich zuerst mit dem Code auseinandersetzen, nur um das Programm zu benutzen.

Aufgrund der Tatsache, dass Open-Source Projekte ausschließlich aus freiwilligen Mitgliedern bestehen, ist eine formale Organisation und Planung schwer denkbar. Somit ist die agile Entwicklung praktisch die Regel, wie das auch bei der Untersuchung festgestellt wurde.

Das Fehlen von Formalitäten bewirkt sich auf die Vielfalt von Ansätzen bei der Erstellung von Code und Dokumentation. Welche Segmente wichtig sind, und wie die Information dazu dargestellt wird, ist mehr oder weniger vom persönlichen Geschmack der Entwickler abhängig.

Die wenige oder komplett fehlende Entwicklerdokumentation ist leider ein oft auftretendes Problem. Das ist auch einer der Hauptgründe, warum sich neue Leute nicht oder schwierig an Open-Source Projekten beteiligen können[1];

[1] GitHub Inc. (2017), Internetquelle

5 Zusammenfassung

Der Ansatz, den Code und die Dokumentation als eine untrennbare Gesamtheit zu betrachten, kann aus Entwickler- oder sogar aus Firmensicht negativ betrachtet werden. Die Erstellung von Dokumentation kann die Entwicklung vom Code nur bremsen und die Produktivität, gemessen in Zeilen Code, sinkt. Diese Betrachtung gilt aber nur in sehr kurzer Sicht. Der Preis wird bei der nächsten Fehlersuche oder Änderung gezahlt, die zusammen mit der Termineinhaltung die größte Herausforderungen beim Embedded Prozess sind[1]. Die Wiederverwendung von Code, die durchaus die Regel ist[2], kann stark davon leiden, dass die Dokumentation fehlt oder unverständlich ist.

Die Software-Entwicklung folgt meistens dem agilen Prozess-Modell, bei dem keine festen Vorschriften über die Art und dem Umfang der erzeugten Dokumente angegeben sind. Typische Dokumentenartefakte sind die Anforderungsspezifikation, Architekturbeschreibungen und weitere für die Entwicklung wichtige Dokumente, sowie Benutzer-orientierte Dokumente wie Anleitungen, Tutorials und andere. Bis auf kleine Projekte sind normalerweise mehrere Dokumente vom selben Typ notwendig, die verschiedene Sichten oder Abstraktionsebenen darstellen. Die Dokumentation ist stark vom Projekt, der Firma und den konkreten Umgebungsgegebenheiten abhängig.

Um konkrete Umsetzungen von Dokumentation bei eingebetteter Software in der Praxis zu analysieren wurden sieben populäre Open-Source Projekte ausgewählt, aufgrund des leichten Zugangs zu deren Code. Die Ergebnisse zeigen einen grundsätzlichen Mangel an Dokumentation für Entwickler und überwiegende Betonung auf diese für die Benutzer. Selbst für die Benutzer ist nicht immer ausreichend Information vorhanden, was einen starken Kritikpunkt darstellt. Fast alle Projekte setzen automatische Tools für Tests und Generierung von Dokumentation ein, die leider ungenügend sind um ein komplettes Bild von der Software zu präsentieren.

Die Betonung der vorliegenden Arbeit lag bei agilen Open-Source Projekten. Diese sind nicht unbedingt ein vollständiges Bild von allem, was in der Software-Welt geschieht. Weitere Untersuchung von klassischen Projekten ist notwendig, um Überblick über andere in der Praxis benutzte Verfahren zu bekommen, insbesondere solche, die mehr formelle Modelle benutzen.

Der Skript in der Untersuchung, der die Lizenzinformation aus den Quellendateien löscht, setzt voraus, dass der erste mehrzeilige Kommentar in der Datei diese Information beinhaltet. Dieses Vorgehen kann nicht alle möglichen Dateien beabsichtigen und der Lizenzheader kann auch anders formatiert werden. Ein anderes Problem ist, dass manchmal ganze Code-Abschnitte auskommentiert sind, damit sie nicht kompiliert werden. Dies kann die

[1] Vgl. Aspencore (2019), S.37
[2] Vgl. Aspencore (2019), S.34

vorgestellte Prozedur auch nicht berücksichtigen. Die vorgestellten Zahlen sollen dafür als Best-Case Szenario interpretiert werden. Die Quantität der Kommentare ist kein Zeichen für ihre Qualität.

Die andere Metrik bei der Untersuchung - die Anzahl der Commits die nur aus Code ohne Kommentare bestehen - ist auch nicht zwingend maßgebend für die Bedeutung der Dokumentation im Team. Es sind durchaus Situationen denkbar, in denen ein diszipliniertes und gut organisiertes Team 20 Commits nacheinander aufnimmt, die nichts mit Dokumentation zu tun haben, und trotzdem bleibt die Information aktuell und ausreichend. Zusätzliches Problem dieses Merkmals ist, dass manche Commits sehr klein sind und dadurch nur das Verhältnis verändern.

Die vorgestellten Metriken sollten dafür nur als Einschätzung dienen und liefern keine endgültige Aussage über den Umgang mit Dokumentation bei den vorgestellten Projekten.

Eine Untersuchung auf die Aktualität der externen Dokumentation, wenn solche vorhanden ist, wurde nicht durchgeführt.

Auf die Darstellung von Werkzeugen bis auf die kurze Betrachtung von DoxyGen wurde auch verzichtet.

Literaturverzeichnis

Aspencore (2019)
2019 Embedded Markets Study: Integrating IoT and Advanced Technology Designs, Application Development and Processing Environments, https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf (Zugriff am 21.09.2020)

Barr, M.; Massa, A. (2006)
Programming embedded systems: with C and GNU development tools, Second Edition, Beijing et al.

Bass, L.; Clements, P.; Kazman, R. (2015)
Software architecture in practice, Upper Saddle River, NJ et al.

Beck, K. et al. (2001)
Manifest für Agile Softwareentwicklung, https://agilemanifesto.org/iso/de/manifesto.html (Zugriff am 21.09.2020)

Behrens, J. et al. (2006)
Architekturbeschreibung, in: Reussner, R.; Hasselbring, W.(Hrsg): Handbuch der Software-Architektur, 2. überarbeitete und erweiterte Auflage, S. 33–68, Heidelberg

Fowler, M. (2006)
Continuous Integration, https://martinfowler.com/articles/continuousIntegration.html (Zugriff am 21.09.2020)

Gessler, R. (2014)
Entwicklung Eingebetteter Systeme: Vergleich von Entwicklungsprozessen für FPGA- und Mikroprozessor-Systeme Entwurf auf Systemebene, Wiesbaden

GitHub Inc. (2017)
Open Source Survey, https://opensourcesurvey.org/2017/ (Zugriff am 21.09.2020)

Goll, J. (2011)
Methoden und Architekturen der Softwaretechnik, Wiesbaden

Kothes, L. (2011)
Grundlagen der Technischen Dokumentation: Anleitungen verständlich und normgerecht erstellen, Heidelberg et al.

Lee, E. A.; Seshia, S. A. (2016)
Introduction to embedded systems: A cyber-physical systems approach, Second Edition, o.O.

Nygard, M. (2011)
Documenting Architecture Decisions, https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions (Zugriff am 21.09.2020)

Sommerville, I. (2011)
Software engineering, 9th Edition, Boston et al.

Wieczorrek, H. W.; Mertens, P. (2011)
Management von IT-Projekten: von der Planung zur Realisierung, 4., überarbeitete und erweiterte Auflage, Heidelberg et al.

Anhang

Anhang 1: Code-Struktur bei den untersuchten Projekten

Language	files	blank	comment	code
C/C++ Header	171	4788	14212	35079
C	94	6616	11356	29696
C++	21	735	621	3553
SUM:	286	12139	26189	68328

Abbildung 1: Code-Struktur in Arduino AVR Core

Language	files	blank	comment	code
C/C++ Header	7061	803273	2100202	5056024
C	6047	474585	1088749	2425961
C++	20	1622	1255	5921
SUM:	13128	1279480	3190206	7487906

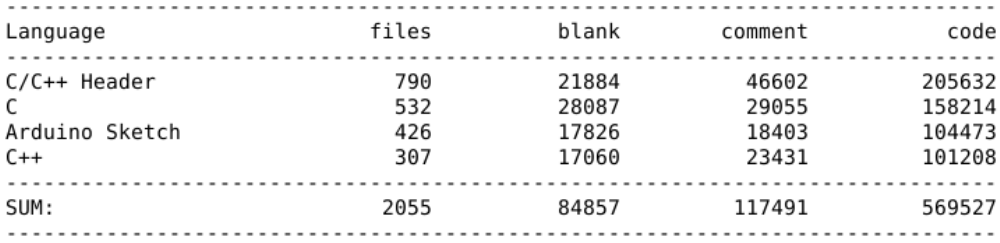
Abbildung 2: Code-Struktur in RT-Thread

Language	files	blank	comment	code
C++	797	47453	21666	145650
C/C++ Header	796	21870	26273	61974
C	329	12080	26875	27083
SUM:	1922	81403	74814	234707

Abbildung 3: Code-Struktur in PX4

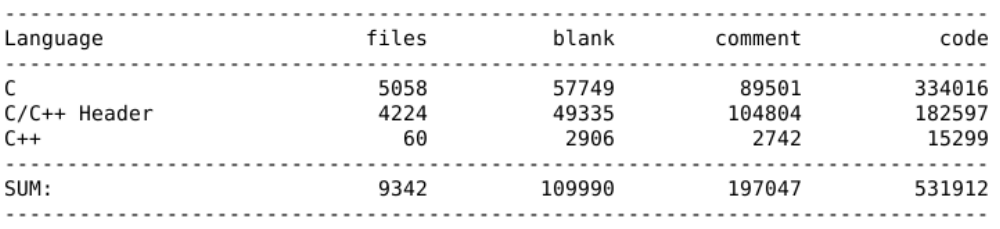
Language	files	blank	comment	code
C	26628	2027183	2126561	11344653
C/C++ Header	13656	300853	697282	1548091
C++	635	28036	22758	125879
SUM:	40919	2356072	2846601	13018623

Abbildung 4: Code-Struktur in Asuswrt-Merlin



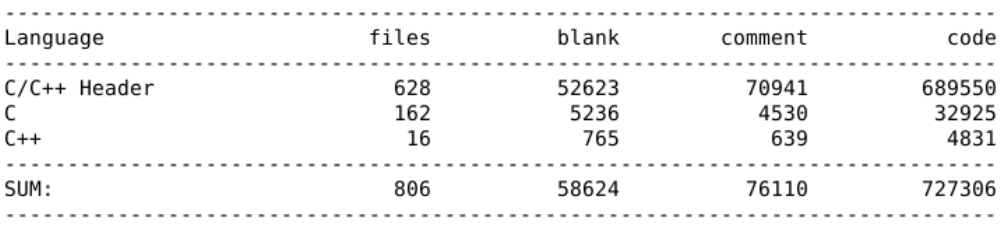
Language	files	blank	comment	code
C/C++ Header	790	21884	46602	205632
C	532	28087	29055	158214
Arduino Sketch	426	17826	18403	104473
C++	307	17060	23431	101208
SUM:	2055	84857	117491	569527

Abbildung 5: Code-Struktur in Tasmota



Language	files	blank	comment	code
C	5058	57749	89501	334016
C/C++ Header	4224	49335	104804	182597
C++	60	2906	2742	15299
SUM:	9342	109990	197047	531912

Abbildung 6: Code-Struktur in Quantum



Language	files	blank	comment	code
C/C++ Header	628	52623	70941	689550
C	162	5236	4530	32925
C++	16	765	639	4831
SUM:	806	58624	76110	727306

Abbildung 7: Code-Struktur in Klipper

Anhang 2: Version der untersuchten Projekte zum Zeitpunkt der Untersuchung

Projekt	Commit-Nummer
Arduino AVR Core	3055c1e
RT-Thread	de0bb6f
PX4 Autopilot	9116acc
Asuswrt-Merlin	7848c48
Tasmota	515ed22
Quantum	3d3c2e1
Klipper	063f9a2

Tabelle 3: Letzter Commit zum Zeitpunkt der Untersuchung