
Model Driven Software Engineering mit IBM Rational Rhapsody für Embedded Systems

Masterprojekt

Thomas Sauter
Matrikel-Nr.: 3122629
Studiengang: SYE\2

Hochschule Ulm
Graduate School
Studiengang Systems Engineering und Management

26. Juni 2017

Betreuer:
Prof. Dr. Marianne von Schwerin, Hochschule Ulm

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation	1
1.2	Aufgabenstellung	1
2	Entwicklungsumgebung	3
2.1	Keil MCB1760 Evaluation Board	3
2.2	ESP8266	4
2.3	Toolchain	4
2.3.1	IBM Rational Rhapsody	4
2.3.2	Willert Embedded UML RXF	5
2.3.3	Keil uVision	5
3	Implementierungen	7
3.1	Ethernet	7
3.1.1	Anforderungen	8
3.1.2	Architektur	9
3.1.3	Design und Coding	10
3.1.4	Konfigurieren des Keil Projekts	14
3.2	SD-Karte	19
3.2.1	Anforderungen	19
3.2.2	Architektur	20
3.2.3	Design und Coding	21
3.2.4	Konfigurieren des Keil Projekts	25
	Abkürzungsverzeichnis	31
	Literaturverzeichnis	33

1 Einleitung

1.1 Motivation

TODO

1.2 Aufgabenstellung

TODO

2 Entwicklungsumgebung

2.1 Keil MCB1760 Evaluation Board

Das Keil MCB1760 Evaluation Board enthält einen NXP LPC1768 Mikrocontroller basierend auf einem 100Mhz ARM 32-bit Cortex-M3 Mikroprozessor. Neben den wesentlichen Komponenten und Schnittstellen, welche in Abbildung 2.1 dargestellt sind, verfügt das Keil MCB1760 Evaluation Board über 512KB Flash und 64KB RAM On-Chip Memory.

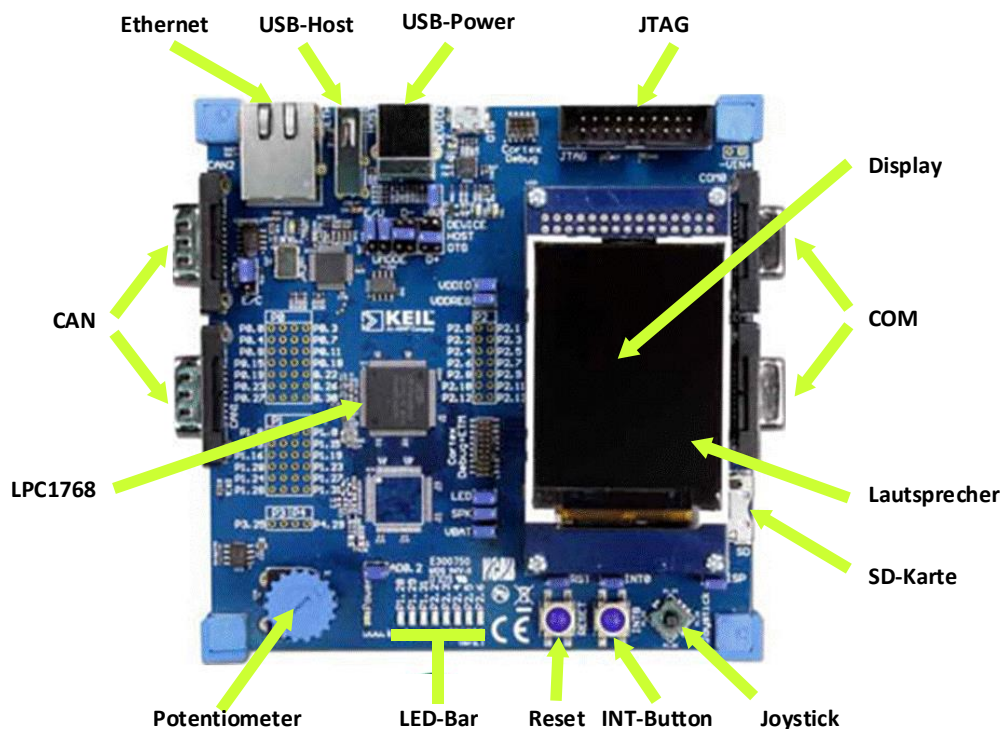


Abbildung 2.1: Komponenten des MCB1760 Evaluation Board.

In der Regel werden Evaluation Boards in früheren Entwicklungsphasen eingesetzt, um die Leistungsgrenzen der gewählten Architektur zu verifizieren. Im Rah-

men dieser Arbeit steht die Integration des Evaluation Boards zusammen mit der Toolchain im Vordergrund.

2.2 ESP8266

TODO

2.3 Toolchain

TODO

2.3.1 IBM Rational Rhapsody

Derzeit gibt es auf dem Markt eine Vielzahl an Software-Modellierungswerkzeuge, die sich im Wesentlichen durch ihre Funktionen und den daraus resultierenden Preis unterscheiden. Jedoch haben die meisten dieser Tools miteinander gemein, dass sie die Modellierungssprache UML unterstützen. So gibt es unter anderem einige kostenlose Tools wie StarUML oder Netbeans, die geringen Anforderungen durchaus genügen. Weitaus mächtiger sind etwa Enterprise Architect von SparxSystems und das in dieser Arbeit verwendete Rational Rhapsody von IBM. Mit Rational Rhapsody ist es möglich, neben UML-Modellierung weitere Aufgaben zu bearbeiten, die während der Softwareentwicklung anfallen. Beispielsweise können Anforderungen direkt spezifiziert oder auch aus DOORS NG importiert und zu der erstellten Architektur verlinkt werden. Ein Codegenerator übersetzt die Architektur in Quelltext. Dabei bietet der Codegenerator etliche Konfigurationsmöglichkeiten, um Layout und Syntax nach Belieben anzupassen. Das Spezifizieren und Ausführen von Tests in einer integrierten Testumgebung runden den Funktionsumfang zur Unterstützung eines Software Entwicklungsprozesses ab.

Rational Rhapsody wird in verschiedenen Versionen angeboten, die sich stark in ihrem Funktionsumfang unterscheiden. Grundlegende Funktionen, wie etwa das Erstellen von UML-Diagrammen und das Verlinken von Anforderungen ist mit allen Versionen möglich. Weitere Funktionalitäten wie grafikbasierte Simulation oder automatische Codegenerierung, unter anderem auch für Embedded Echtzeitsysteme, ist nur in den Premium-Versionen verfügbar (IBM, 2017).

Beim Generieren von Quelltext unterstützt Rational Rhapsody die Programmiersprachen C, C++, Java und C#. Dabei ist es wichtig, dass direkt beim Anlegen des Projekts die gewünschte Programmiersprache ausgewählt wird. Denn in der Folge startet Rational Rhapsody beim Öffnen der Rhapsody Projektdatei stets die passende Variante für die definierte Programmiersprache. Da die Implementierungen in dieser Arbeit in C++ erfolgen, ergibt sich somit die Version *IBM Rational Rhapsody Developer for C++*, die zudem den vollen Funktionsumfang beinhaltet.

2.3.2 Willert Embedded UML RXF

Der generierte Code aus Rhapsody eignet sich zunächst nicht zur Ausführung auf einem Target. Die UML-Notation ist viel leistungstärker und auf einer höheren Abstraktionsebene als jede höhere Programmiersprache. UML-Elemente wie asynchrone Kommunikation, aktive Klassen oder auch komplexe Zustände können nicht direkt in eine höhere Programmiersprache übersetzt werden.

Das Tool Embedded UML Real-time eXecution Framework (RXF) der Firma Willert bildet die Schnittstelle zwischen UML-Modell und einer Zielplattform bestehend aus Compiler, CPU und einem möglichen RTOS. Durch eine Abstraktionsschicht werden die gängigsten Echtzeit-Betriebssysteme auf dem Markt unterstützt. Das bedeutet, dass in UML definierte Timer oder Events unabhängig vom Betriebssystem verwendet werden können. Somit ist das Software-Design komplett losgelöst vom gewählten Target.

Bei der Codegenerierung unterstützt das RXF die beiden bekanntesten UML-Tools, Rhapsody und Sparx Enterprise Architect, sowie eine Vielzahl an IDEs. Um eine bestmögliche Integration zu gewährleisten, ist jede Variante des RXF auf die verwendete Toolchain zugeschnitten. Ein Vorteil davon ist, dass die Target IDE über das RXF mit Rhapsody verbunden ist und somit der Code aus dem UML-Modell direkt in die Target IDE generiert wird (Van der Heiden, 2016). Zur Unterscheidung der vielen verschiedenen Varianten hat die Firma Willert mit der Version 6 einen Produktcode eingeführt, welcher zur Identifikation der enthaltenen Komponenten dient. Das Schema ist in Tabelle 2.1 abgebildet.

In dieser Arbeit wurden die Varianten *RXF-Eval-Rpy-Cpp-ARM* in der Version 6.02 und *Rpy-CPP-CMSIS_Keil5_ARM_MCB1700-TD* in der Version 6.01 verwendet.

2.3.3 Keil uVision

Die IDE Keil uVision ist Teil des Keil Microcontroller Development Kit (MDK). Es vereint einen Projektmanager und eine Run-Time Environment (RTE), mit deren Hilfe vorgefertigte Software Pakete integriert werden können. Die Software Pakete

UML-Tool	Programmier- sprache	RTOS	Compiler	EvalBoard*	Erweiterungen**
<p>* Die EvalBoard Komponente ist kein Teil des Produkts. Sie sagt lediglich aus, mit welcher CPU Familie das Produkt verwendet werden kann.</p> <p>** Erweiterungen sind optional und können auch miteinander kombiniert werden. Mögliche Zusätze sind „TD“ für Embedded UML Target Debugger oder „Eval“ für eine RXF Evaluierungsversion.</p>					

Tabelle 2.1: Produktcode zur Identifikation der enthaltenen Komponenten (Römer, 2012).

können Bibliotheken, Module, Konfigurationsdateien, Templates und Dokumentation enthalten, welche bei der Inbetriebnahme des Targets unterstützen. Die Basisfunktionalitäten einer gewöhnlichen IDE, wie Quellcode-Editor und Debugger, sind ebenfalls enthalten (ARM Keil Group, 2017b).

In dieser Arbeit wird das Keil MDK in der Version 5 verwendet. Im Vergleich zum vorherigen Keil MDK in der Version 4, ist eine wesentliche Neuerung das Echtzeitbetriebssystem Cortex Microcontroller Software Interface Standard (CMSIS). Es löst das bisherige RTX Real-Time Library (RL-ARM) Echtzeitbetriebssystems ab und bringt die folgenden Vorteile mit sich (ARM Keil Group, 2014):

- Standardisierte API
- Basisfunktionen zur Unterstützung von UML oder Java
- Einfaches wiederverwenden von Software Komponenten durch einheitliche Funktionen
- CMSIS konforme Middleware kann einfach angepasst werden

3 Implementierungen

3.1 Ethernet

Dieses Kapitel beschreibt die Einbindung der Ethernet Schnittstelle des Keil MCB1760 Evaluation Boards. Ziel ist es, dass zwei Boards über ihre Ethernet Schnittstelle Daten austauschen können. In der Implementierung nach Steinmeyer und Pollithy (2015) wurde die gewünschte Funktionalität bereits umgesetzt, jedoch auf der Basis des RTX RL-ARM Echtzeitbetriebssystems und einer damit überholten Version der Keil MDK. Zudem soll die gesamte Implementierung in Rhapsody stattfinden, so dass der generierte Code in der IDE Keil uVision lediglich übersetzt und auf das Target geflasht werden muss.

Zur Implementierung und Demonstration der Kommunikation über Ethernet wurde eine Entwicklungsumgebung entsprechend der nachfolgenden Abbildung aufgebaut.

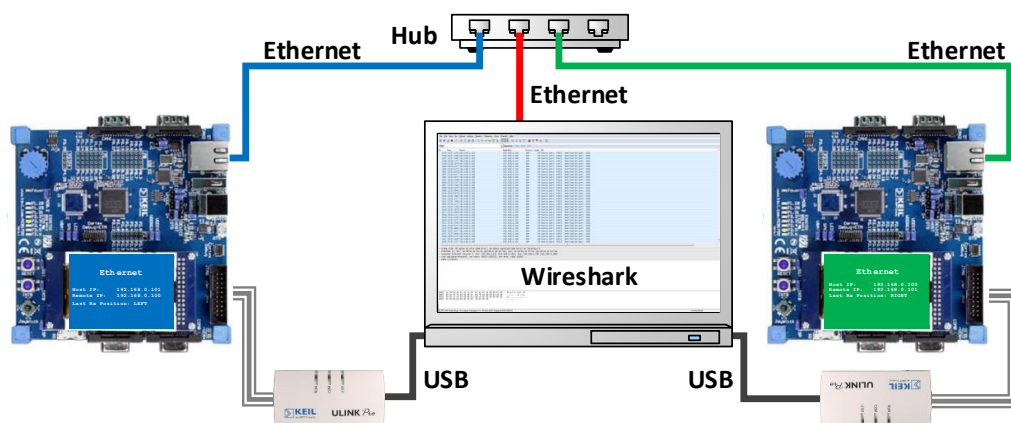


Abbildung 3.1: Demonstrator für die Kommunikation zwischen zwei MCB1760 Evaluation Boards über Ethernet

Wie in Abbildung 3.1 dargestellt, sind die beiden MCB1760 Evaluation Boards über Patchkabel mit einem Hub verbunden. Der Hub hat gegenüber einem Switch oder Router den Nachteil, dass er eine geringere nutzbare Bandbreite mit sich bringt.

Grund dafür ist, dass der Hub ein Datenpaket immer an jedes angeschlossene Gerät sendet, unabhängig davon, ob das Datenpaket an das Gerät adressiert wurde oder nicht. Jedoch unterstützt dieses Defizit bei der Implementierung, indem mit Hilfe eines PCs das Tool Wireshark den Datenverkehr zwischen den beiden MCB1760 Evaluation Boards abhört. Da die versendeten Datenpakete eine Größe von zwei Bytes haben, spielt die nutzbare Bandbreite im Rahmen dieser Arbeit keine Rolle.

3.1.1 Anforderungen

Zur Demonstration einer funktionsfähigen Ethernet-Kommunikation soll ein exemplarisches Szenario implementiert werden. So sollen LEDs durch den Joystick auf dem jeweils anderen Board ein- und ausgeschaltet werden. Dabei soll die Position des Joysticks angeben, welche LED die LED-Bar ein- oder ausschaltet. Des Weiteren soll das Display die zuletzt empfangene Position anzeigen. Damit die MCB1760 Evaluation Boards einfach zu identifizieren sind, soll das Display zudem die Host IP Adresse sowie die Target IP Adresse darstellen. Beim Senden sowie beim Empfangen eines Datenpakets soll der Transmitter bzw. der Receiver kurzzeitig eine LED blinken lassen. Die Zuordnung von Funktionen zu den LEDs der LED-Bar ist Abbildung 3.2 rechts dargestellt.

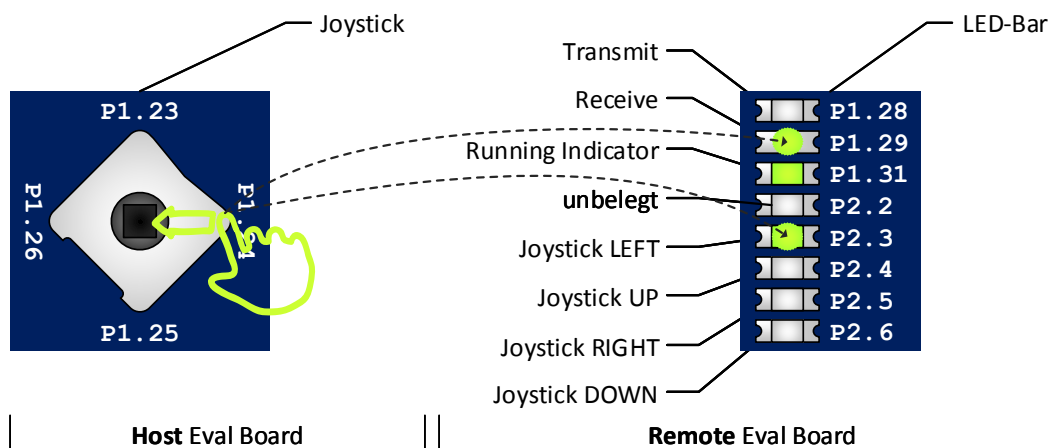


Abbildung 3.2: Beispiel mit Zuordnung der LEDs.

Zudem veranschaulicht Abbildung 3.2 das Funktionsprinzip. Auf dem Host MCB1760 Evaluation Board bewegt der Benutzer den Joystick in die linke Richtung. Das führt dazu, dass beim Remote MCB1760 Evaluation Board die Receiver LED P1.29 kurzzeitig aufleuchtet, sowie die LED P2.3 dauerhaft angeschaltet wird. Bewegt der Benutzer den Joystick erneut in die linke Richtung, erlischt die LED P2.3 wieder.

3.1.2 Architektur

Bei der Formulierung der Anforderungen in Kapitel 3.1.1 wurde darauf geachtet, dass diese in möglichst aktiver Form spezifiziert sind. Dadurch können die benötigten Klassen abgeleitet werden, welches sich im Folgenden durch die Ähnlichkeit von Subjekten oder Objekten zu den Klassennamen widerspiegelt.

Die Architektur der Ethernet-Kommunikation ist als Klassendiagramm in Abbildung 3.3 dargestellt. Zentrales Element ist die Basisklasse `EthernetController`, von ihr werden die beiden Klassen `EthernetTransmitter` und `EthernetReceiver` abgeleitet. Diese beiden Klassen arbeiten in separaten Tasks und sind für das Senden und Empfangen von Datenpaketen verantwortlich. Auf der linken Seite in Abbildung 3.3 sind die Klassen `RunningIndicatorLed` und `Joystick` abgebildet, welche ebenfalls in eigenen Tasks ausgeführt werden. Die Klasse `RunningIndicatorLed` lässt die LED P1.31 zyklisch blinken, mit einer Periodendauer von einer Sekunde. Sie dient zu Debugging zwecken und um unmittelbar zu erkennen, ob das Target läuft. Die Klasse `Joystick` pollt regelmäßig die Position des Joysticks. Auf der rechten Seite sind die Klassen `LedBar`, `Display` und `Led` zu finden. Dabei liegt zwischen den Klassen `LedBar` und `Led` eine Komposition mit der Multiplizität vier vor, wodurch der LED-Bar die LEDs zugeordnet sind, welche eine Joystick Position repräsentieren. Außerhalb des Pakets `DefaultPkg` sind Abhängigkeiten zu externen Bibliotheken in orange eingezeichnet. Die verwendeten Bibliotheken, `Network` und `Graphics Component`, stammen aus der MDK Middleware und vereinfachen das Verwenden dieser Peripheriegeräte. Mögliche Vorgehensweisen beim Einbinden externer Quellen beschreiben Matuschek und Van der Heiden (2015).

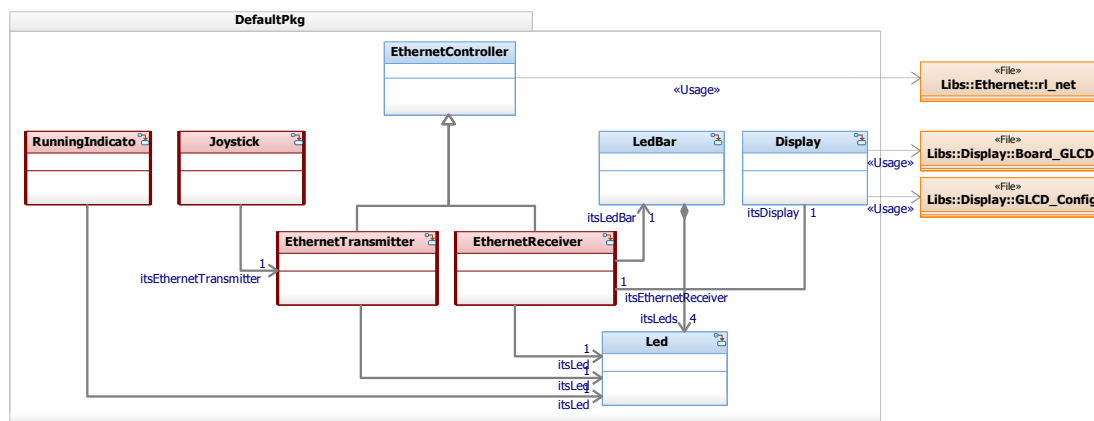


Abbildung 3.3: Klassendiagramm zur Ethernet-Kommunikation. Klassen, die in eigenen Tasks laufen sind rot eingezeichnet.

3.1.3 Design und Coding

In diesem Kapitel werden Attribute, Funktionen und Statecharts wichtiger Klassen im Detail vorgestellt.

Ethernet-Controller, Transmitter und Receiver

Der Ethernet-Controller basiert auf der MDK Middleware Network Component in der Version 7.4.1. Die Network Component beinhaltet eine Vielzahl an Services, Sockets (TCP, UDP und BSD), sowie eine Ethernet Schnittstelle inklusive eines IPv4/IPv6 Protocol Stacks. In dieser Arbeit wird der BSD Socket als Datagram Socket (UDP) zusammen mit dem IPv4 Protocol Stack verwendet. Der BSD Socket stellt eine API zur Verfügung, die das Aufbauen und Abhandeln einer Netzwerkkommunikation unterstützt. Ursprünglich wurden die BSD Sockets für unixnahe Betriebssysteme entwickelt. Mittlerweile sind sie in den POSIX Standard aufgenommen und wurden auch von Microsoft Windows übernommen. Ein Vorteil der BSD Sockets ist, dass mit geringem Konfigurationsaufwand zwischen Stream Sockets (TCP) und Datagram Sockets (UDP) gewechselt werden kann.

Die Klasse EthernetController mit ihren Attributen und Operationen ist in Abbildung 3.4 dargestellt. Zum Spezifizieren der IP-Adressen dienen die Attribute hostIpAddr und remoteIpAddr vom Typ String. Die IP-Adresse werden in Rhapsody über den Features Dialog der beiden Attribute initial festgelegt. Zum Flashen des zweiten Targets können die IP-Adressen einfach getauscht werden.

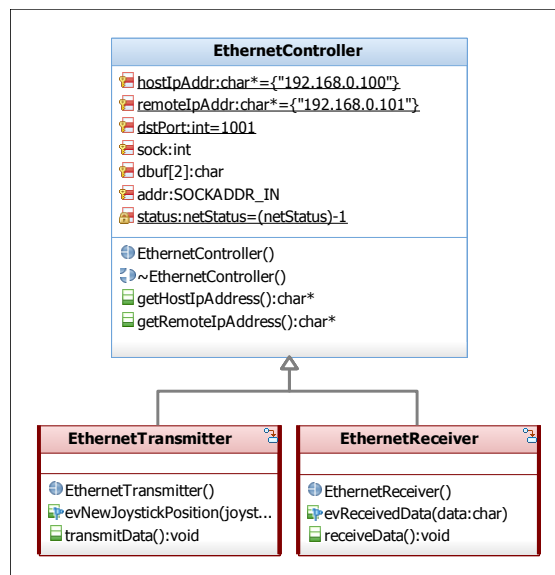


Abbildung 3.4: Basisklasse des Ethernet-Controllers mit abgeleiteten Klassen.

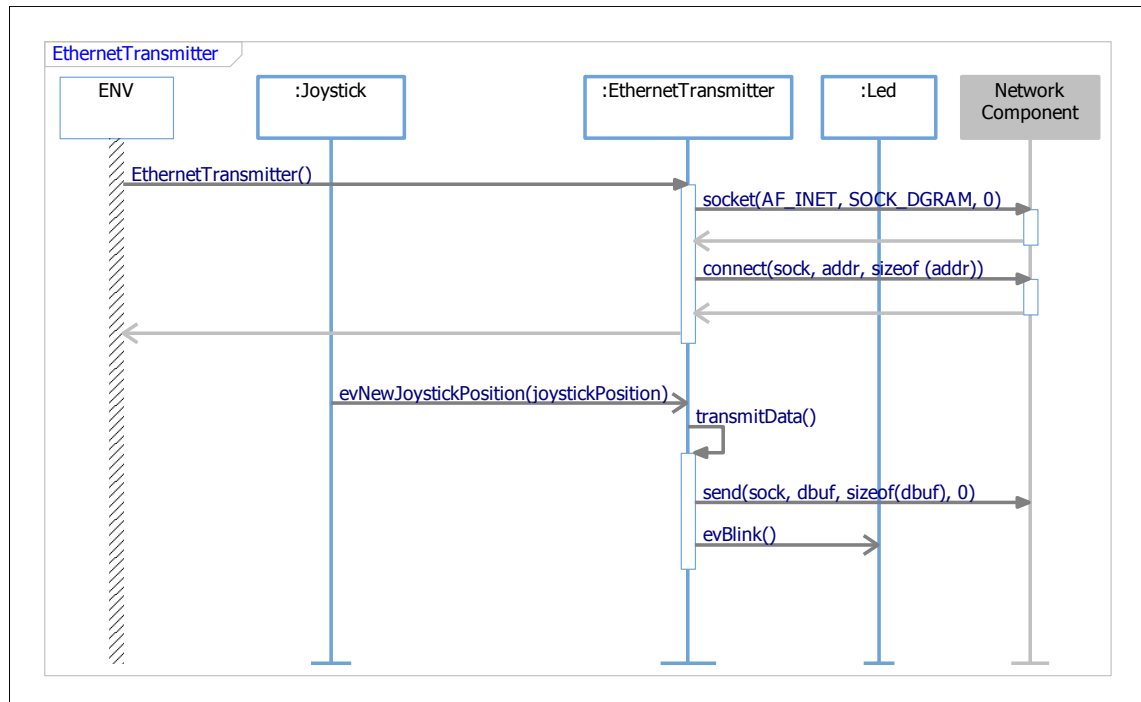
```
1 unsigned char buf[8];
2
3 if (status != netOK)
4 {
5     // Initialize the network component only once
6     status = netInitialize ();
7
8     // Set the host ip address once
9     netIP_aton (hostIpAddr, NET_ADDR_IP4, buf);
10    netIF_SetOption (
11        NET_IF_CLASS_ETH | 0,
12        netIF_OptionIP4_Address,
13        buf,
14        NET_ADDR_IP4_LEN);
15 }
```

Quelltext 3.1: Konstruktor des Ethernet-Controllers

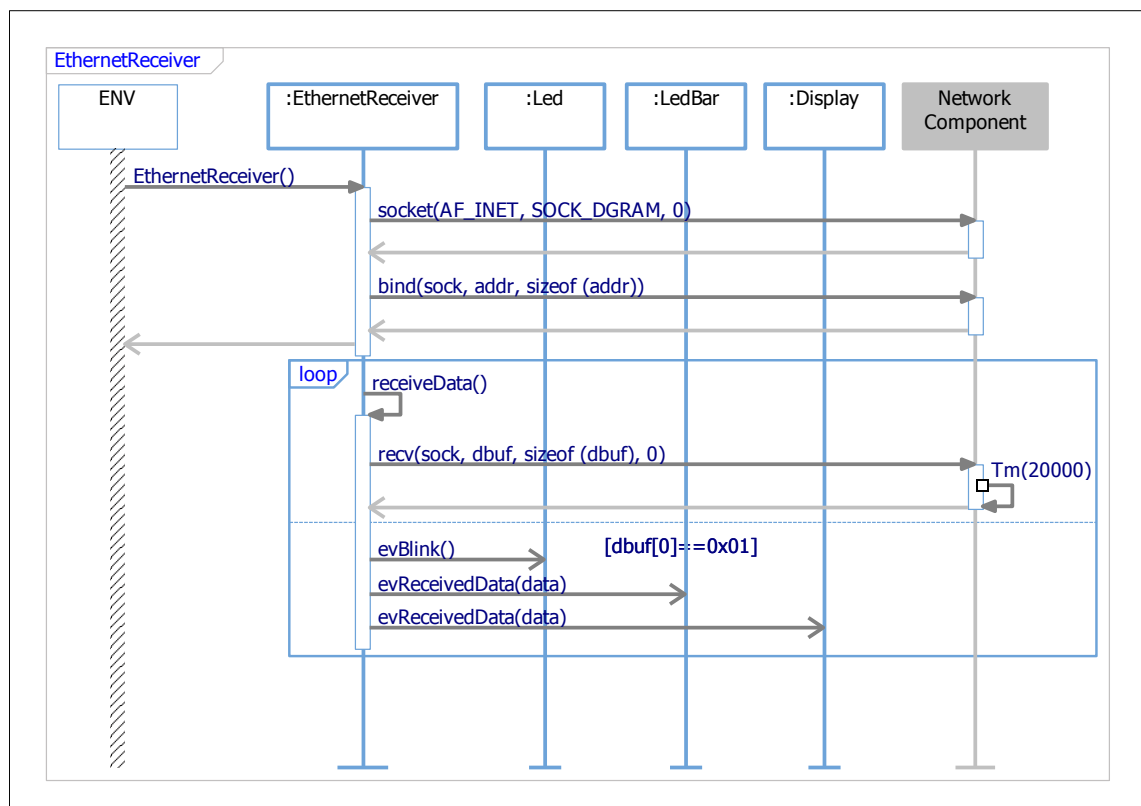
Quelltext 3.1 zeigt den Konstruktor des EthernetControllers. Der Konstruktor verwendet ausschließlich Funktionen der Network Component, was am Prefix `net` zu erkennen ist. In Zeile 6 wird die Funktion `netInitialize` aufgerufen. Diese Funktion muss bei Systemstart einmalig ausgeführt werden. Sie initialisiert Systemressourcen, Protokolle und zwei Tasks für den Network Core. Bei erfolgreicher Initialisierung wird dem Attribut `status` der Wert `netOK` zugewiesen. Mit der übergeordneten Abfrage wird sichergestellt, dass die Initialisierung auch nur einmalig durchgeführt wird, auch wenn der Konstruktor des EthernetControllers durch die beiden Instanzen der abgeleiteten Klassen zweimal durchlaufen wird. Die Funktion `netIP_aton` konvertiert eine IP-Adresse vom Typ `String` in eine binäre Form. Dadurch ist es anschließend möglich, die Host IP-Adresse dynamisch mit Hilfe der Funktion `netIF_SetOption` zu setzen. Somit ist die Konfiguration der Host IP-Adresse ebenfalls in Rhapsody möglich und benötigt keine manuelle Anpassung innerhalb der Keil Umgebung.

Das Sequenzdiagramm in Abbildung 3.5(a) stellt das Verhalten des EthernetTransmitters dar. Der EthernetTransmitter erstellt in seinem Konstruktor einen Socket vom Typ `Datagram Sockets (UDP)`, zu erkennen am Übergabeparameter `SOCK_DGRAM`. Dem Socket weist er durch Aufruf von `connect` die Remote IP-Adresse, also die IP-Adresse des anderen Endpunkts, zu. Aufgrund des gewählten Typ `Datagram Sockets (UDP)`, richtet der EthernetTransmitter zudem ein Adressfilter zwischen den Endpunkten ein. Im weiteren Ablauf reagiert der EthernetTransmitter auf das Event `evNewJoystickPosition` welches vom Joystick abgefeuert wird. Durch die Operation `transmitData` handelt der EthernetTransmitter seine Sendeaktivitäten ab. Die Funktion `send` sorgt dafür, dass die Daten im Buffer `dbuf` übertragen werden und dass das Event `evBlink` die Sende-LED P1.28 blinken lässt.

Der EthernetReceiver erstellt in seinem Konstruktor ebenfalls einen Socket



(a) Initialisierung und Sendebetrieb des Ethernet-Transmitters.



(b) Initialisierung des Ethernet-Receiver und Ablauf beim Empfangen von Daten.

Abbildung 3.5: Sequenzdiagramme des Ethernet-Transmitters und -Receivers.

vom Typ Datagram Sockets (UDP) und bindet diesen mit der Funktion `bind` an die Remote IP-Adresse sowie an den Ziel-Port. Der Ziel-Port `dstPort` kann für beide Targets gleich bleiben, er definiert an welchem Port auf eingehende Datenpakete gehorcht wird. Anschließend durchläuft der `EthernetTransmitter` eine Endlosschleife. In dieser ruft er seine Operation `receiveData` auf, welche das Empfangen von Daten behandelt. Die Funktion `recv` empfängt eingehende Daten auf dem zuvor spezifizierten Port. Wenn der Network Core erkennt, dass ein Betriebssystem im Einsatz ist, betreibt der Network Core die Funktion `(recv)` automatisch im Blocking Mode. Dadurch ist es zwingend erforderlich, dass die Klasse `EthernetReceiver` in einem separaten Task ausgeführt wird. Der Blocking Mode ist zudem mit einem Timeout verbunden, der bei Ablauf in den Errorcode `BSD_ERROR_TIMEOUT` resultiert. Nach Ablauf der vorkonfigurierten Zeit von 20 Sekunden, oder falls zuvor Daten Empfangen wurden, ruft der `EthernetReceiver` die Funktion `(recv)` erneut auf. Hat der `(EthernetReceiver)` Daten Empfangen, werden diese auf ihre Gültigkeit überprüft. Dazu dient ein minimales Protokoll, dessen erstes Byte signalisiert, dass ein passendes Datenpaket vorhanden ist. Entspricht das erste Byte dem Wert `0x00`, feuert der `EthernetReceiver` das Event `evBlink` an seine LED, damit die Empfangs-LED P1.29 blinkt. Zudem sendet er das Event `evReceivedData` mit dem Inhalt des zweiten Bytes an die LED-Bar, sowie an das Display. Das Verhalten des `EthernetReceiver` ist in Abbildung 3.5(b) dargestellt.

Somit gibt es je Target einen `EthernetTransmitter` und einen `EthernetReceiver`, die in eigenen Tasks ihre Routinen durchlaufen.

Joystick

Der Joystick verfügt in Summe über sechs verschiedene Richtungen, von denen zunächst die vier Richtungen links, rechts, oben und unten von Interesse sind. Das Abtasten und Auslesen der Position des Joysticks erfolgte in Anlehnung an von Schwerin und Normann (2017). Dabei wurde die Auswertung der Joystick Position um ein Filter ergänzt, damit nur relevante und neue Positionen via Ethernet übertragen werden. Ursache dafür ist, dass der Joystick nach der Betätigung, in eine der zuvor aufgezählten Richtung, wieder in die zentrale Position zurück kehrt. Dadurch nimmt der Joystick eine für ihn neue Position ein und würde ohne Filter das Senden eines Datenpakets triggern. Quelltext 3.2 zeigt die Implementierung des Filters. Bewegt sich der Joystick in die mittige Position, wird nie ein Event abgefeuert. Hat der Joystick eine andere Richtung eingenommen, feuert die Klasse `Joystick` das Event `evNewJoystickPosition` an den `EthernetTransmitter`.

Die Klasse `Joystick` hält die Position für eine Dauer von 100 Millisekunden, liest im Anschluss daran den aktuellen Wert aus dem entsprechenden Register aus und filtert diesen. Dieses Sample-and-Hold-Verhalten wurde im zugehörigen Statechart modelliert. Die Halte-Dauer kann im Konstruktor der Klasse `Joystick` angepasst werden.

```
1 int position = (LPC_GPIO1->FIOPIN >> 20) & Joystick_Mask ;
2
3 // Bit 4-7 contain the position information
4 position = position >> 3;
5
6 if (position == Joystick_CENTER)
7 {
8     // If the Joystick got back to center only update
9     lastPosition = position;
10 }
11 else if (position != lastPosition)
12 {
13     lastPosition = position;
14     FIRE( this->itsEthernetTransmitter, evNewJoystickPosition(
15         position));
16 }
```

Quelltext 3.2: Filter zur Auswertung der Joystick Position


LED-Bar und Display

Die LED-Bar und das Display sind die Empfänger des Events `evReceivedData`, welches der Ethernet-Receiver mit dem Inhalt des zweiten Bytes als Parameter abfeuert. Die Klasse `LedBar` ordnet den empfangenen Parameter der entsprechenden LED zu und ruft die Operation `toggleLed` der Klasse `Led` auf. Die Klasse `Display` nutzt ebenfalls den empfangenen Parameter und zeigt damit die zuletzt empfangene Richtung an.

3.1.4 Konfigurieren des Keil Projekts

Als Basis für die Implementierung wurde das Blinky-Beispielprojekt verwendet, welches im RXF *Rpy_CPP_CMSIS_Keil5_ARM_MCB1700_TD* von Willert enthalten ist. Zwar ist das Keil Projekt lediglich rudimentär konfiguriert, stellt aber die funktionsfähige Einbindung von Rhapsody sicher. Im Folgenden werden die wichtigsten Anpassungen in der Konfiguration gegenübergestellt.

Manage Run-Time Environment

Mittels des Konfigurationsassistenten  *Manage Run-Time Environment* ist es möglich, Software Komponenten einem Keil Projekt hinzuzufügen. Mit dem Ziel eine Ethernet Kommunikation aufzubauen, wird zunächst der Software Pack *Keil::MDK-Middleware* in der Version 7.4.1 (2017-04-21) dem Projekt hinzugefügt. Diese Paket beinhaltet unter anderem die Network Component in der Version 7.5.0 (2017-04-21). Zum Betreiben der Network Component wird der *ARM::CMSIS:CORE* in der Version

5.0.1 vorausgesetzt. Die dazu notwendige Änderung im Ethernet Projekt gegenüber dem Blinky Projekt ist in Tabelle 3.1 rot gekennzeichnet.

Die Network Component beinhaltet eine Vielzahl an Komponenten, die dem Keil Projekt hinzugefügt werden können. Für den vorliegenden Fall der Ethernet-Kommunikation sind die Komponenten und die benötigten CMSIS Treiber gemäß Tabelle 3.1 zu wählen. Notwendige Anpassungen zur Verwendung des Displays sind orange markiert.

Software Component	Blinky		Ethernet	
	Sel. Variant	Version	Sel. Variant	Version
Board Support	MCB1700		MCB1700	
Graphic LCD (API)		1.0.0		1.0.0
Graphic LCD	<input type="checkbox"/>	1.0.0	<input checked="" type="checkbox"/>	5.0.1
CMSIS				
CORE	<input checked="" type="checkbox"/>	4.3.0	<input checked="" type="checkbox"/>	5.0.1
CMSIS Driver				
Ethernet MAC (API)		2.01		2.1.0
Ethernet MAC	<input type="checkbox"/>	2.9	<input checked="" type="checkbox"/>	2.9.0
Ethernet PHY (API)		2.00		2.1.0
DP83848C	<input type="checkbox"/>	6.1	<input checked="" type="checkbox"/>	6.1.0
SPI (API)		2.01		2.2.0
SPI	<input type="checkbox"/>	2.1	<input type="checkbox"/>	2.1.0
SSP	<input type="checkbox"/>	2.5	<input checked="" type="checkbox"/>	2.7.0
Device				
GPDMA	<input type="checkbox"/>	1.2	<input checked="" type="checkbox"/>	1.2.0
GPIO	<input type="checkbox"/>	1.1	<input checked="" type="checkbox"/>	1.1.0
PIN	<input type="checkbox"/>	1.0	<input checked="" type="checkbox"/>	1.0.0
Startup	<input checked="" type="checkbox"/>	1.0.0	<input checked="" type="checkbox"/>	1.0.0
Network				
CORE	<input type="checkbox"/>	MDK-Pro 7.4.0	<input checked="" type="checkbox"/>	MDK-Pro 7.5.0
Legacy API	<input type="checkbox"/>	IPv4/IPv6 Release 7.4.0	<input type="checkbox"/>	IPv4/IPv6 Release 7.5.0
Interface				
ETH	0	7.4.0	1	7.5.0
PPP	<input type="checkbox"/>	7.4.0	<input type="checkbox"/>	7.5.0
SLIP	<input type="checkbox"/>	7.4.0	<input type="checkbox"/>	7.5.0
Service				
Socket				
BSD	<input type="checkbox"/>	7.4.0	<input checked="" type="checkbox"/>	7.5.0
TCP	<input type="checkbox"/>	7.4.0	<input checked="" type="checkbox"/>	7.5.0
UDP	<input type="checkbox"/>	7.4.0	<input checked="" type="checkbox"/>	7.5.0

Tabelle 3.1: Heraufsetzen des CMSIS:CORE (rot), benötigte Komponenten der Network Component und deren Abhängigkeiten (gelb), sowie die Komponenten zum Betreiben des Displays (orange).

Target Options

Mit der Aufnahme der Network Component in das Keil Projekt, steigt der erforderliche RAM-Speicherbedarf der Applikation auf über 41KB (RW-data=352 Bytes + ZI-data=41176 Bytes). Damit wird die vorkonfigurierte RAM-Speichergröße von 32KB überschritten. Jedoch verfügt das Keil MCB1760 Evaluation Board über insgesamt 64KB RAM On-Chip Memory, so dass die weiteren 32KB RAM mit Hilfe des Scatter Files adressiert werden müssen. Das Scatter File (.sct) befindet sich im Flash-Ordner des Keil-Projekts. Allerdings bietet Keil die Möglichkeit, dass Scatter File über die Bedienoberfläche zu generieren, so dass das Scatter File nicht direkt editiert werden muss. Über die *Target Options* im Reiter *Target* können im Panel *Read/Write Memory Areas* die zweiten 32KB RAM-Speicher aktiviert werden. Die Startadresse der zweiten Speicherbank ist mit 0x2007C000 anzugeben, die Größe des Speichers von 32KB ebenfalls als hexadezimaler Wert mit 0x8000.

CMSIS Configuration

Bei der CMSIS Configuration geht es primär um das Konfigurieren des CMSIS RTX Kernel. Dabei wird die Datei *RTX_Conf_CM.c*, die Teil der CMSIS Component ist, editiert. Keil bietet den Komfort, die Datei über den integrierten *Configuration Wizard* zu bearbeiten. Der erste Parameter *Number of concurrent running user threads* im Abschnitt *Thread Configuration* gibt die Anzahl der Tasks an, die zur gleichen Zeit laufen. Die Tasks mit der Ursache CMSIS-RTOS in Tabelle Tabelle 3.2 sind bei jeder Applikation standardmäßig aktiv, die das CMSIS-RTOS verwenden.

Das Willert RXF bringt durch sein kleines Onboard-Betriebssystem mit *WST_Monitor_receiveTask* einen weiteren Task mit sich. Zudem beansprucht die Network Component bei der Initialisierung die beiden Tasks *netCore_Thread* und *netETH_Thread*. Zusätzlich kommen noch die Tasks hinzu, die durch die Applikation an sich gefordert sind. Gemäß der in rot gekennzeichneten Klassen in Abbildung 3.3

Task Name	Priorität	Ursache
osTimerThread	1	CMSIS-RTOS
main	2	CMSIS-RTOS
os_idle_demon	255	CMSIS-RTOS
WST_Monitor_receiveTask	3	Willert RXF
netCore_Thread	4	Network Component
netETH_Thread	5	Network Component
RunningIndicator	6	Ethernet Applikation
EthernetReceiver	7	Ethernet Applikation
EthernetTransmitter	8	Ethernet Applikation
Joystick	9	Ethernet Applikation

Tabelle 3.2: Verwendete Task für die Ethernet-Kommunikation und deren Ursache.

sind das die Tasks *RunningIndicator*, *EthernetReceiver*, *EthernetTransmitter* und *Joystick*. In Summe ergeben sich somit zehn Tasks. Da nach ARM Keil Group (2017d) der Task *os_idle_demon* nicht in die Anzahl der gleichzeitig laufenden Tasks mit eingeht, wird der Parameter *Number of concurrent running user threads* auf den Wert neun gesetzt.

Unter den weiteren Parametern im Abschnitt *Thread Configuration* ist vor allem der Parameter *Total stack size [bytes] for threads with user-provided stack size* von Interesse. Dieser muss im Vergleich zum Blinky Projekt um 1024 Bytes für den Task *netCore_Thread* erhöht werden, sowie um weitere 512 Bytes für den Task *netEth_Thread* (ARM Keil Group, 2017c). Die Anpassungen für die Ethernet-Kommunikation in der Datei *RTX_Conf_CM.c* gegenüber dem Blinky Projekt sind in Tabelle 3.3 aufgelistet.

Option	Blinky	Ethernet
	Value	Value
<input type="checkbox"/> Thread Configuration		
— Number of concurrent running user threads	6	9 (+3)
— Default Thread stack size [bytes]	200	200
— Main Thread stack size [bytes]	1024	1024
— Number of threads with user-provided stack size	5	5
— Total stack size [bytes] for threads with user-provided stack size	4096	5632 (+1536)
— Check for stack overflow	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
— Processor mode for thread execution	Privileged mode	Privileged mode
<input type="checkbox"/> RTX Kernel Timer Tick Configuration		
<input type="checkbox"/> System Configuration		

Tabelle 3.3: Anpassungen in der RTX Configuration *RTX_Conf_CM.c*.

Device Configuration

Die Datei *startup_LPC17xx.s (Startup)* bildet zusammen mit der Datei *startup_LPC17xx.c (Startup)* den Start-up Code, welcher direkt nach einem RESET des Targets ausgeführt wird. Die beiden Dateien sind Teil der Device Component und können ebenfalls über den *Configuration Wizard* bearbeitet werden. Von größerem Interesse ist die Datei *startup_LPC17xx.s (Startup)* bei der Zuordnung von Speicher erfolgt. In der Datei *startup_LPC17xx.c (Startup)* geht es im Wesentlichen um die Clock Konfiguration, welche aber bei ihrer Standardeinstellung belassen wird.

Nach ARM Keil Group (2017c) ist bei Verwendung des Ethernet Cores eine Vergrößerung der Stack Size um 512 Bytes erforderlich. Eine Anpassung des Heaps ist nicht erforderlich, da in dieser Implementierung die Security Komponente nicht verwendet wird. Die Änderung im Startup File ist in Tabelle 3.4 dargestellt.

3 Implementierungen

Option	Blinky	Ethernet
	Value	Value
<input type="checkbox"/> Stack Configuration		
└─ Stack Size (in Bytes)	0x0000 0200	0x0000 0400 (+512)
<input type="checkbox"/> Heap Configuration		
└─ Heap Size (in Bytes)	0x0000 1000	0x0000 1000

Tabelle 3.4: Anpassungen in der Startup Configuration *startup_LPC17xx.s* (Startup) für die Ethernet-Kommunikation.

Ethernet Network Configuration

Die Ethernet-Konfiguration erfolgt über die Konfigurationsdateien, die zur Network Component gehören. Dabei ist in Konfigurationsdatei *Net_Config_ETH_0.h* (Interface:ETH) eine Anpassung vorzunehmen, welche die Option *Dynamic Host Configuration* betrifft. Wenn diese Option aktiviert ist, werden IP-Adresse, Netzmaske und Standardgateway automatisch von einem DHCP Server bezogen. Da der Demonstrationsaufbau über keinen DHCP Server verfügt und die IP-Adressen statisch vergeben werden, muss diese Option deaktiviert werden. Tabelle 3.5 zeigt, dass die IP-Adresse in dieser Datei auch manuell konfiguriert werden kann. Jedoch wird diese Option zur Laufzeit überschrieben, da die Implementierungen beschrieben in Kapitel 3.1.3 die IP-Adresse festlegen.

Option	Blinky	Ethernet
	Value	Value
<input type="checkbox"/> Ethernet Network Interface 0		
└─ Connected to hardware via Driver ETH#	0	0
└─ MAC Address	1E-30-6C-A2-45-5E	1E-30-6C-A2-45-5E
<input type="checkbox"/> IPv4	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
└─ IP Address	192.168.0.100	192.168.0.100
└─ Subnet Mask	255.255.255.0	255.255.255.0
└─ Default Gateway	192.168.0.254	192.168.0.254
└─ Primary DNS Server	8.8.8.8	8.8.8.8
└─ Secondary DNS Server	8.8.4.4	8.8.4.4
<input checked="" type="checkbox"/> IP Fragmentation	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> ARP Address Resolution		
<input checked="" type="checkbox"/> IGMP Group Management	<input type="checkbox"/>	<input type="checkbox"/>
└─ NetBIOS Name Service	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> Dynamic Host Configuration	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> IPv6	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> OS Resource Settings		

Tabelle 3.5: Anpassungen in der Ethernet Network Configuration *Net_Config_ETH_0.h*.

3.2 SD-Karte

Im Fokus dieses Kapitels steht die Inbetriebnahme des SD-Karten Slots des Keil MCB1760 Evaluation Boards. Der Schacht für die microSD-Karte befindet sich unterhalb des Displays, wie Abbildung 2.1 zu entnehmen ist. Das Ziel der Inbetriebnahme ist es, beispielhaft Daten auf die SD-Karte zu schreiben, lesen und diese wieder zu löschen. Wie auch schon bei der Ethernet-Applikation in Kapitel 3.1 soll die Implementierung in Rational Rhapsody erfolgen und die IDE Keil uVision lediglich zum Übersetzen und Flashen dienen.

3.2.1 Anforderungen

Ein beispielhaftes Szenario soll die funktionsfähige Inbetriebnahme der SD-Karte demonstrieren. Dabei soll der Joystick bei Betätigung in eine Richtung seine neue Position auf der SD-Karte aufzeichnen. Unter einer Positionsänderung des Joysticks sind die vier Richtungen links, rechts, oben und unten zu verstehen. Wenn der Joystick nach einer Richtungsänderung zurück in die zentrale Position geht, wird das nicht als Positionsänderung gewertet. Die SD-Karte soll sämtliche Aktivitäten und ihren Status über das Display ausgeben. Abbildung 3.6 veranschaulicht das Funktionsprinzip.

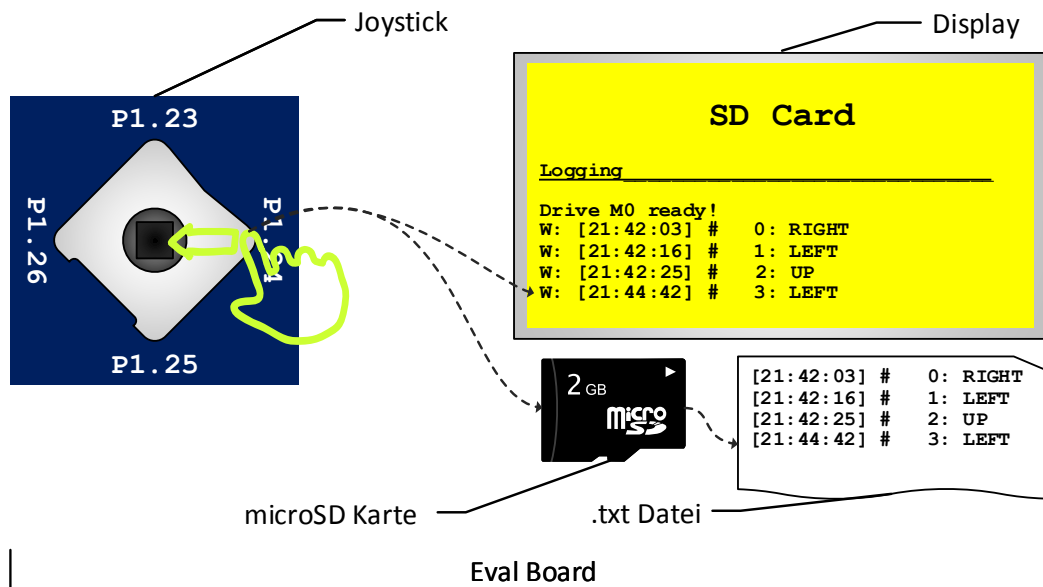


Abbildung 3.6: Beispielhaftes Logging der Joystick Position auf dem Display und auf der SD-Karte.

Zusätzlich ist es möglich, auf den Joystick zu drücken, was im Folgenden als Richtung *SELECT* bezeichnet wird. Die Richtung *SELECT* soll eine besondere Funktion im Beispielszenario einnehmen. Drückt der Benutzer den Joystick zweimal hintereinander, dann soll die SD-Karte die angelegte Log-Datei löschen. Bei erstmaligem Drücken soll die SD-Karte dem Benutzer über das Display mitteilen, dass die SD-Karte durch erneutes Drücken des Joysticks die angelegte Log-Datei löscht. Wählt der Benutzer jedoch eine andere Richtung, soll die SD-Karte wieder in den Log-Status zurückkehren. Ebenso soll die SD-Karte eine minimale Fehlerbehandlung enthalten und dem Benutzer über das Display entsprechende Fehlermeldungen anzeigen.

Zur Verifizierung der Lesefunktion soll die SD-Karte überprüfen, ob bereits eine Log-Datei durch vorherige Verwendung angelegt wurde. Wenn das der Fall ist, soll die SD-Karte die letzte Zeile der Log-Datei auslesen, die fortlaufende Nummer extrahieren und den Zähler auf diesen Wert setzen.

3.2.2 Architektur

Das Klassendiagramm in Abbildung 3.7 stellt die Architektur des SD-Karten Loggings dar. Im Fokus des Diagramms steht die Klasse `SDCard`. Sie besitzt eine gerichtete Assoziation zur Klasse `Clock` und zur Klasse `Display`. Dadurch kann die Klasse `SDCard` einen Zeitstempel abfragen und den aktuellen Status auf dem Display ausgeben.

In Abbildung 3.7 links dargestellt sind beiden Klassen `Joystick` und `RunningIndicatorLed`, welche in eigenen Threads laufen. Dabei verfügt die Klasse `Joystick` über eine gerichtete Assoziation zur Klasse `SDCard`. Somit kann die Klasse `Joystick`,

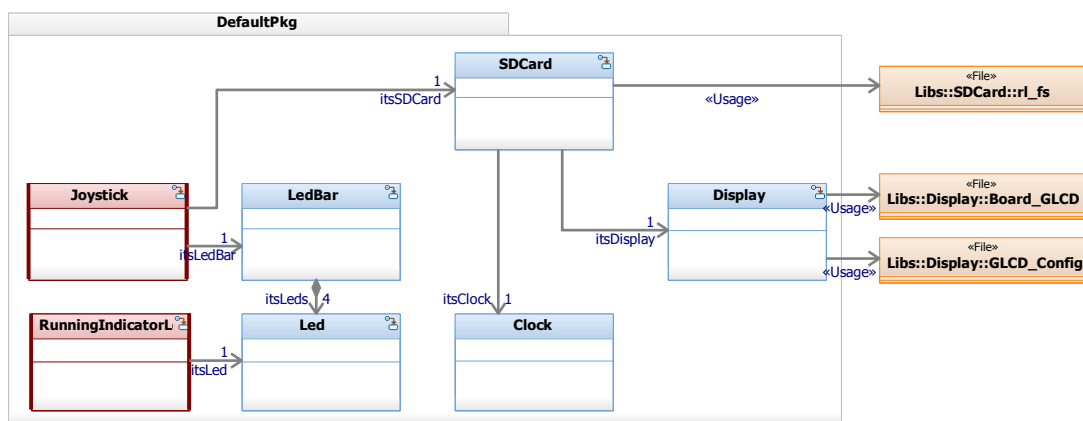


Abbildung 3.7: Klassendiagramm zum Logging der Joystick Positionen auf der SD-Karte. Klassen, die in eigenen Threads laufen sind rot eingezeichnet.

welche in einem festgelegten Intervall die Joystick Position pollt, die Klasse `SDCard` über Positionsänderungen informieren. Die Klasse `Joystick` verfügt über eine weitere Assoziation zur Klasse `LedBar`, wodurch die Joystick Position mit Hilfe von vier LEDs visualisiert wird. Wie in den vorherigen Implementierungen lässt die Klasse `RunningIndicatorLed` die LED P1.31 zyklisch blinken und dient lediglich zu Debugging zwecken. Die in Abbildung 3.7 rechts in orange dargestellten Dateien außerhalb des Pakets `DefaultPkg` stellen Abhängigkeiten zu externen Bibliotheken dar. Die verwendeten Bibliotheken stammen aus der MDK Middleware und vereinfachen das Betreiben der File System und Graphics Component.

3.2.3 Design und Coding

Dieses Kapitel behandelt Attribute, Funktionen und Statecharts wichtiger Klassen.

SD-Karte

Die Implementierung der SD-Karte erfolgt durch die MDK Middleware File System Component in der Version 6.9.8. Die File System Component unterstützt eine Vielzahl an Speichern und Speichergeräten, welchen jeweils ein Laufwerksbuchstabe zugewiesen ist. Der Laufwerksbuchstabe wird an Systemroutinen übergeben, wo er zur Initialisierung eines Dateisystems verwendet wird. Das Dateisystem, wie beispielsweise File Allocation Table (FAT) File System oder Embedded File System (EFS) wird in Abhängigkeit vom Laufwerk festgelegt. Die Klasse `SDCard` mit ihren Attributen und Operationen ist in Abbildung 3.8 dargestellt. Dabei legt das Attribut `drive` den Laufwerksbuchstaben „M0:“ und das damit verbundene Dateisystem FAT fest.

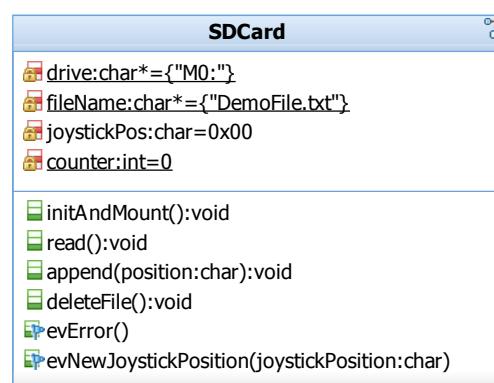


Abbildung 3.8: Klassendiagramm der SD-Karte.

Initialisierung: Das Sequenzdiagramm in Abbildung 3.9 stellt den Ablauf der Operation `initAndMount` dar. Dabei erfolgt das Initialisieren und Mounten des Dateisystems über die Systemroutinen `finit` und `fmount`. Außerdem sind die entsprechenden Statusmeldungen an das Display im Erfolgsfall (grün) und Fehlerfall (rot) eingezeichnet.

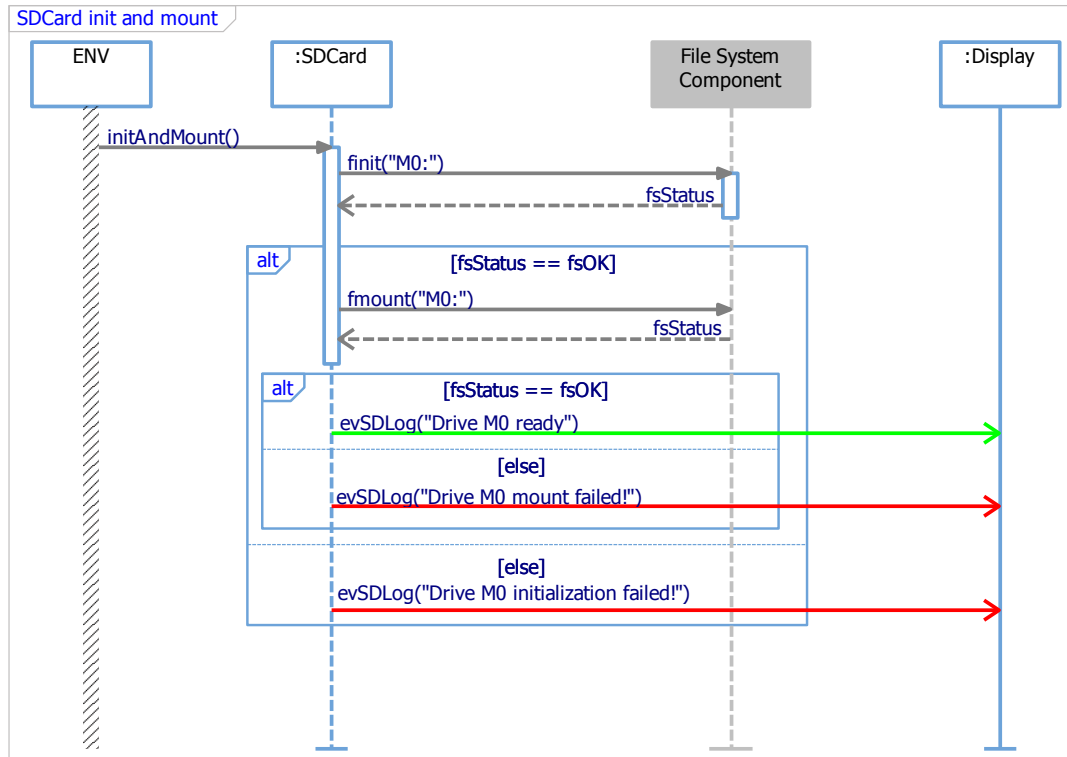


Abbildung 3.9: Initialisieren und Mounten der SD-Karte.

Lesen: Leseoperationen erfolgen mit Hilfe der Standard Input und Output Library `stdio.h`. Die Library `stdio.h` benutzt Datenströme (Stream) zur Kommunikation mit physikalischen Peripheriegeräten wie Tastatur und Drucker, oder auch zur Kommunikation mit jeder anderen Art von Dateien, welche das System unterstützt. Dabei werden die Streams immer identisch angewendet, unabhängig davon, ob mit einem Gerät oder einer Datei kommuniziert wird. Durch die Funktion `fopen` wird ein Stream geöffnet und mit einem Gerät oder einer Datei verbunden. Bei erfolgreichem Öffnen des Streams gibt die Funktion `fopen` einen Zeiger auf ein `FILE`-Objekt zurück. Dieser Zeiger beinhaltet alle Informationen des Datenstroms und muss bei jeder weiteren Dateioperation als Parameter übergeben werden.

Der für dieses Szenario implementierte Lese-Mechanismus ist als Sequenzdiagramm in Abbildung 3.10 abgebildet. Nach dem Öffnen des Streams wird mit der Funktion `fgets` Zeile für Zeile aus dem Logfile ausgelesen, bis das Ende der Datei erreicht ist.

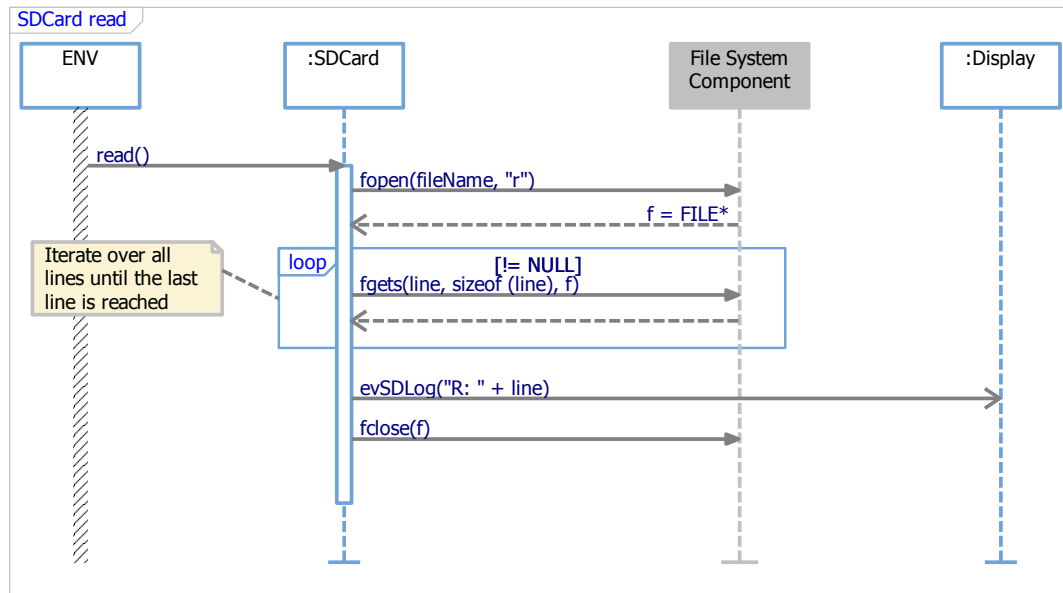


Abbildung 3.10: Lesen der SD-Karte.

Da alle Zeilen des Logfiles das gleiche Format haben, kann aus der letzten Zeile die fortlaufende Nummer extrahiert und zum Setzen des Zählers benutzt werden (siehe Abbildung 3.6). Den entsprechenden Ausschnitt aus der Operation `read` zeigt Quelltext 3.3. Dabei wird zunächst der Teil der Zeichenkette ab dem `#`-Symbol extrahiert. Danach wird mit der Funktion `scanf` der Dezimalwert ausgelesen und in `counter` geschrieben.

Im Anschluss wird das Event `evSDLog` mit der Bezeichnung „R:“ für *Read* und der letzten gelesenen Zeile als Parameter an das Display gefeuert. Nach erfolgreicher Dateioperation muss der Datenstrom wieder mittels `fclose` geschlossen werden.

```

1 // Set counter to last value
2 substring = strstr(line, "# ");
3 if (substring != NULL)
4 {
5     /* Found "# " in row. Extract decimal */
6     sscanf(substring + 3, "%d", &counter);
7 }
  
```

Quelltext 3.3: Extrahieren des aktuellen Zählstandes aus dem Logfile.

Schreiben: Wie die Leseoperationen bedienen sich auch die Schreiboperationen bei der Standard Input und Output Library `stdio.h`. Gleichmaßen gilt es zunächst einen Stream zu Öffnen um einen Zeiger auf ein FILE-Objekt zu erhalten. Im Gegensatz zur Leseoperation, welche die Funktion `fopen` mit dem Parameter „r“ für *Read mode* aufruft, wird bei der Schreiboperation der Parameter „a“ für *Append mode* verwendet. Der *Append mode* öffnet eine bestehende Datei und fügt dieser Inhalte hinzu. Falls die Datei noch nicht besteht, wird eine leere Datei angelegt. Wenn eine Datei im *Append mode* geöffnet wird, werden die neuen Inhalte stets am Ende der Datei hinzugefügt.

Der implementierte Schreib-Mechanismus ist in Abbildung 3.11 als Sequenzdiagramm dargestellt. Nach dem Öffnen des Datenstroms wird die Zeichenkette `str` durch die Funktion `fprintf` in den Stream geschrieben. Die Funktion `fflush` sorgt letztendlich dafür, dass die Schreiboperation ausgeführt wird. Das bedeutet, dass die zugehörigen Buffer geleert und deren Inhalt in die entsprechende Datei

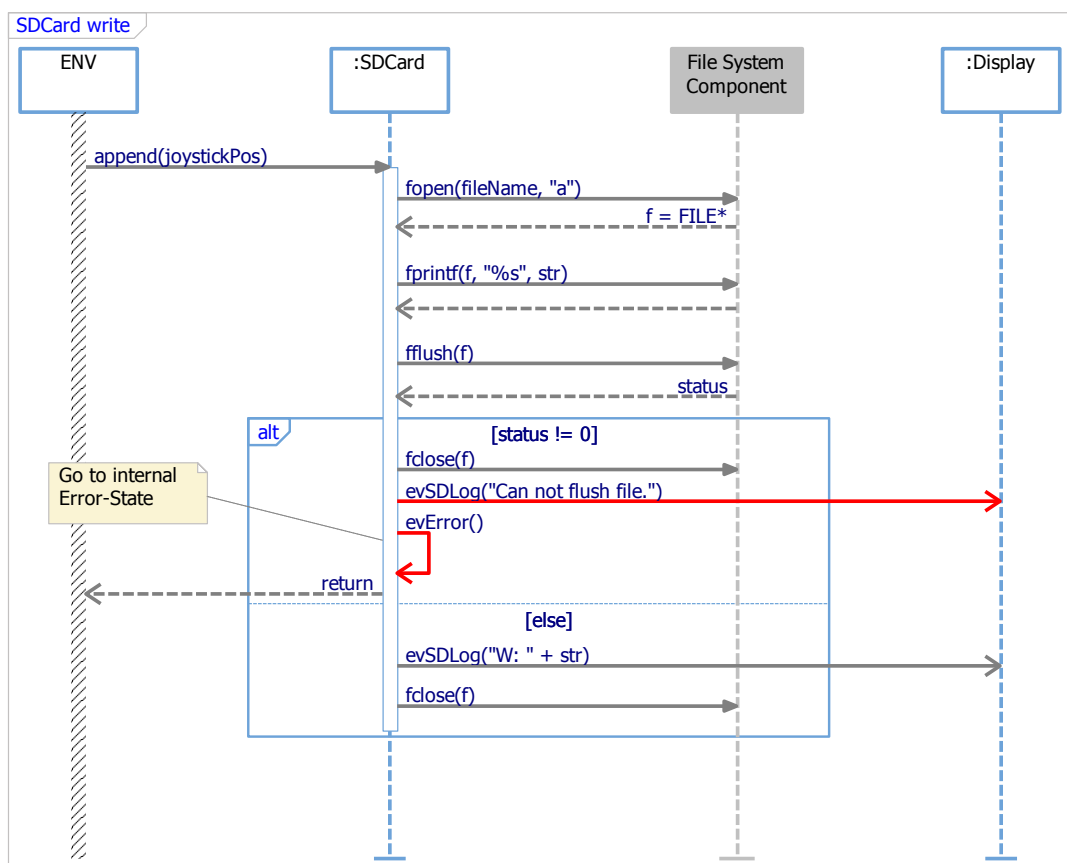


Abbildung 3.11: Schreiben auf die SD-Karte.

geschrieben wird. Sollte der Rückgabewert der Funktion `fflush` ungleich Null sein, so ist ein Fehler aufgetreten. Die Klasse `SDCard` feuert in diesem Fall das Event `evSDLog` mit einer entsprechenden Fehlermeldung an das Display, sowie das Event `evError` an sich selbst (siehe rote Markierungen in Abbildung 3.11). Durch das Event `evError` geht die Klasse `SDCard` in einen internen Fehlerstatus. Bei einer erfolgreichen Schreiboperation feuert die Klasse `SDCard` ebenfalls das Event `evSDLog` an das Display. In diesem Fall beinhaltet der Parameter des Events `evSDLog` den Bezeichner „W:“ für *Write* sowie die aktuell geschriebene Zeile. Zuletzt muss der Datenstrom durch die Funktion `fclose` wieder geschlossen werden.

Datei anlegen: Das Anlegen einer Datei geschieht implizit beim Aufruf der Funktion `fopen`. Dabei ist der Übergabeparameter für den Zugriffsmodus entscheidend. Mit „a“ für *Append mode* wird nur dann eine leere Datei angelegt, wenn diese noch nicht existiert. Mit dem Parameter „w“ für *Write mode* wird bei jedem Aufruf eine leere Datei angelegt. Eine existierende Datei wird dabei überschrieben.

Datei löschen: Zum Löschen einer Datei dient die Funktion `fdelete`. Als Parameter wird dieser Funktion der Name einer zu löschenden Datei oder eines Verzeichnisses übergeben. Mit einem weiteren Parameter „/S“ wird spezifiziert, ob bei Angabe eines Verzeichnisses alle Dateien und Unterverzeichnisse ebenfalls gelöscht werden sollen. Die Funktion `fdelete` liefert einen Rückgabewert vom Typ `fsStatus` der eine detaillierte Fehlerbehandlung ermöglicht.

Joystick


Clock

Display

3.2.4 Konfigurieren des Keil Projekts

Wie in auch in den vorherigen Implementierungen wurde das Blinky-Beispielprojekt aus dem RXF *Rpy_CPP_CMSIS_Keil5_ARM_MCB1700_TD* von Willert verwendet. Dadurch ist das Zusammenspiel von Rational Rhapsody, dem Willert RXF und der IDE Keil uVision sichergestellt. Nachfolgend werden die wichtigsten Anpassungen in der Konfiguration dargestellt.

Manage Run-Time Environment

Zur Verwendung der SD-Karte ist es nötig dem Keil Projekt, mit Hilfe des Konfigurationsassistenten  *Manage Run-Time Environment*, Software Komponenten hinzuzufügen. Zuerst wird der Software Pack *Keil::MDK-Middleware* in der Version 7.4.1 (2017-04-21) dem Projekt hinzugefügt. In diesem Paket ist die File System

Component in der Version 6.9.8 (2017-04-21) enthalten. Zur Verwendung der File System Component wird der *ARM::CMSIS:CORE* in der Version 5.0.1 benötigt. Zwar erstellt die File System Component keinen eigenen Thread, greift aber dennoch auf die Schnittstellen des Betriebssystems zu. Das geschieht beispielsweise dann, wenn eine Datei geöffnet wird und mittels Mutex ein exklusiver Zugriff auf die Datei gewährleistet wird. Die entsprechende Anpassung in der Run-Time Environment ist in Tabelle 3.6 rot markiert.

Die File System Component unterstützt sowohl das Betreiben rudimentärer Flash-Speicher-Chips in NAND- oder NOR-Architektur, als auch den Datenaustausch mit einem USB Massenspeicher oder einer SD Speicherkarte. Die benötigten Komponenten der File System Component für den Einsatz der SD Speicherkarte sind in Tabelle 3.6 gelb gekennzeichnet.

CMSIS Configuration

Die CMSIS Configuration behandelt die Einstellung des CMSIS RTX Kernel. Zwar legt die File System Component keine neuen Threads an, benötigt aber dennoch das CMSIS-RTOS zum Betrieb. Beispielsweise werden die Mechanismen zum Erstellen eines Mutex verwendet. Die Anforderungen an den CMSIS RTX Kernel werden in der Datei *RTX_Conf_CM.c* gepflegt, welche Teil der CMSIS Component ist. Diese Datei kann über den *Configuration Wizard* von Keil editiert werden. Im Abschnitt *Thread Configuration* in der Datei *RTX_Conf_CM.c* spezifiziert der erste Parameter *Number of concurrently running user threads* die Anzahl nebenläufiger Tasks. Die Tasks der SD-Karten Applikation sind in Tabelle 3.7 aufgelistet. Dabei sind die Tasks mit der Ursache CMSIS-RTOS und Willert RXF identisch zu den Tasks in Tabelle 3.2 aus der Ethernet-Applikation.

Gemäß ARM Keil Group (2017d) geht der Task *os_idle_demon* nicht in die Anzahl der gleichzeitig laufenden Tasks mit ein, weshalb der Parameter *Number of concurrent running user threads* auf den Wert fünf gesetzt wird. Dementsprechend ist auch der Parameter *Number of threads with user-provided stack size* auf den Wert vier zu setzen, da hier der main-Task nicht mitgezählt wird.

Abhängig vom gewählten Drive Type gibt ARM Keil Group (2017a) vor, wie die Parameter bezüglich der *Default*, *Main* und *Total Thread stack size* zu konfigurieren sind. Für den hier gewählten Drive Type *Memory Card drive 0* in Kombination mit dem CMSIS Treiber für SPI sind keine weiteren Anpassung nötig. Die Änderungen in der Datei *RTX_Conf_CM.c* gegenüber dem Blinky-Projekt sind in Tabelle 3.8 zusammengefasst.

Der Vollständigkeit halber wird an dieser Stelle noch auf die Anforderungen bezüglich Mutexes eingegangen, auch wenn im Rahmen dieser Implementierung keine Anpassung nötig sind. Sollen mehr als drei Dateien gleichzeitig geöffnet und bearbeitet werden, muss die Anzahl der Mutexobjekte angepasst werden. Diese Änderung kann nicht über den *Configuration Wizard* erfolgen, sondern muss direkt

Software Component	Blinky		SD Card	
	Sel. Variant	Version	Sel. Variant	Version
Board Support	MCB1700	1.0.0	MCB1700	1.0.0
Graphic LCD (API)		1.0.0		1.0.0
Graphic LCD	<input type="checkbox"/>	1.0.0	<input checked="" type="checkbox"/>	5.0.1
CMSIS	<input checked="" type="checkbox"/>	4.3.0	<input checked="" type="checkbox"/>	5.0.1
CORE				
CMSIS Driver				
SPI (API)		2.01		2.2.0
SPI	<input type="checkbox"/>	2.1	<input type="checkbox"/>	2.1.0
SSP	<input type="checkbox"/>	2.5	<input checked="" type="checkbox"/>	2.7.0
Compiler	ARM Compiler	1.2.0	ARM Compiler	1.2.0
Event Recorder	DAP	1.1.0	DAP	1.1.0
I/O				
File	<input type="checkbox"/> File System	1.2.0	<input checked="" type="checkbox"/> File System	1.2.0
STDERR	<input type="checkbox"/> User	1.2.0	<input type="checkbox"/> User	1.2.0
STDIN	<input type="checkbox"/> User	1.2.0	<input type="checkbox"/> User	1.2.0
STDOUT	<input type="checkbox"/> User	1.2.0	<input type="checkbox"/> User	1.2.0
TTY	<input type="checkbox"/> User	1.2.0	<input type="checkbox"/> User	1.2.0
Device				
GPDMA	<input type="checkbox"/>	1.2	<input checked="" type="checkbox"/>	1.2.0
GPIO	<input type="checkbox"/>	1.1	<input checked="" type="checkbox"/>	1.1.0
PIN	<input type="checkbox"/>	1.0	<input checked="" type="checkbox"/>	1.0.0
Startup	<input checked="" type="checkbox"/>	1.0.0	<input checked="" type="checkbox"/>	1.0.0
File System	MDK-Pro	6.9.8	MDK-Pro	6.9.8
CORE	SFN	6.9.8	LFN	6.9.8
Drive				
Memory Card	0	6.9.8	1	6.9.8
NAND	0	6.9.8	0	6.9.8
NOR	0	6.9.8	0	6.9.8
RAM	<input type="checkbox"/>	6.9.8	<input type="checkbox"/>	6.9.8
USB	0	6.9.8	0	6.9.8

Tabelle 3.6: Heraufsetzen des CMSIS:CORE (rot), benötigte Komponenten der File System Component und deren Abhängigkeiten (gelb), sowie die Komponenten zum Betreiben des Displays (orange).

Task Name	Priorität	Ursache
osTimerThread	1	CMSIS-RTOS
main	2	CMSIS-RTOS
os_idle_demon	255	CMSIS-RTOS
WST_Monitor_receiveTask	3	Willert RXF
RunningIndicator	4	SD-Karte Applikation
Joystick	5	SD-Karte Applikation

Tabelle 3.7: Verwendete Task für den Betrieb der SD-Karte und deren Ursache.

Option	Blinky	SD-Karte
	Value	Value
<input type="checkbox"/> Thread Configuration		
— Number of concurrent running user threads	6	5 (-1)
— Default Thread stack size [bytes]	200	200
— Main Thread stack size [bytes]	1024	1024
— Number of threads with user-provided stack size	5	4 (-1)
— Total stack size [bytes] for threads with user-provided stack size	4096	5632 (+1536)
— Check for stack overflow	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
— Processor mode for thread execution	Privileged mode	Privileged mode
<input type="checkbox"/> RTX Kernel Timer Tick Configuration		
<input type="checkbox"/> System Configuration		

Tabelle 3.8: Anpassungen in der RTX Configuration *RTX_Conf_CM.c* für die SD-Karte.

in der Datei *RTX_Conf_CM.c* manuell durchgeführt werden. Dabei ist der Parameter *OS_MUTEXCNT* zu editieren, welcher standardmäßig auf den Wert acht gesetzt ist. Der Parameter *OS_MUTEXCNT* setzt sich dabei folgendermaßen zusammen: 2 (internal stdio operations) + 3 (stdin, stdout und stderr file streams) + 3 für jede geöffnete Datei.

Device Configuration

Die Dateien *startup_LPC17xx.s* (*Startup*), *startup_LPC17xx.c* (*Startup*) und *RTE_Device.h* (*Startup*) implementieren den Start-up Code, welcher direkt nach einem RESET des Targets ausgeführt wird. Alle drei Dateien gehören zur Device Component und können ebenfalls über den *Configuration Wizard* bearbeitet werden. Von größerem Interesse ist zunächst die Datei *startup_LPC17xx.s* (*Startup*) bei der festgelegt wird, wie viel Speicher für Stack und Heap zu reservieren sind.

Bei Verwendung der File System Component ist eine Vergrößerung der Stack Size um 512 Bytes nötig. Eine Anpassung des Heaps ist ebenfalls erforderlich und hängt dabei mit der Anzahl gleichzeitig geöffneter Dateien zusammen. Für jede geöffnete Datei müssen 512+96 Bytes reserviert werden. In diesem Fall soll nur eine Datei gleichzeitig geöffnet werden (ARM Keil Group, 2017a). Wie Tabelle 3.9 zu entnehmen ist, wurde der Heap insgesamt um 4096 Bytes vergrößert. Die Differenz von 3488 Bytes kommt für den Betrieb des Logging-Mechanismus auf dem Display zum Einsatz. Dieser Mechanismus basiert auf der Klasse *OMString*, welche Teil des Rhapsody OXF ist und zusätzlichen Heap benötigt.

Die Datei *RTE_Device.h* (*Startup*) muss ebenfalls angepasst werden. In ihr erfolgt die Pin Konfiguration für die verschiedenen Schnittstellen des Keil MCB1760 Eval

Option	Blinky	SD-Karte
	Value	Value
<input type="checkbox"/> Stack Configuration		
└─Stack Size (in Bytes)	0x0000 0200	0x0000 0400 (+512)
<input type="checkbox"/> Heap Configuration		
└─Heap Size (in Bytes)	0x0000 1000	0x0000 2000 (+4096)

Tabelle 3.9: Anpassungen in der Startup Configuration *startup_LPC17xx.s* (*Startup*) für die SD-Karte.

Boards. Da in dieser Implementierung die SD-Karte mittels SPI angesprochen wird, müssen die entsprechenden Pins dem SPI Treiber *Driver_SPI0* zugewiesen werden. Die Zuordnung der Pins zu den jeweiligen SPI-Leitungen ist in Tabelle 3.10 abgebildet. Der SPI Treiber *Driver_SPI1* ist ebenfalls aktiviert, er ist zum Betrieb des Displays nötig.

Option	Blinky	SD-Karte
	Value	Value
<input checked="" type="checkbox"/> USB Controller [Driver_USBD and Driver_USBH]	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> ENET (Ethernet Interface) [Driver_ETH_MAC0]	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> I2C0 (Inter-integrated Circuit Interface 0) [Driver_I2C0]	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> I2C1 (Inter-integrated Circuit Interface 1) [Driver_I2C1]	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> I2C2 (Inter-integrated Circuit Interface 2) [Driver_I2C2]	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> UART0 (Universal asynchronous receiver transmitter)	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> UART1 (Universal asynchronous receiver transmitter)	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> UART2 (Universal asynchronous receiver transmitter)	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> UART3 (Universal asynchronous receiver transmitter)	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> CAN1 Controller [Driver_CAN1]	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> CAN2 Controller [Driver_CAN2]	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/> SSP0 (Synchronous Serial Port 0) [Driver_SPI0]	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<input type="checkbox"/> Pin Configuration		
└─SSP0_SSEL		P0_16
└─SSP0_SCK		P0_15
└─SSP0_MISO		P0_17
└─SSP0_MOSI		P0_18
<input type="checkbox"/> DMA		
<input checked="" type="checkbox"/> Tx	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> Rx	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/> SSP1 (Synchronous Serial Port 1) [Driver_SPI1]	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> SPI (Serial Peripheral Interface) [Driver_SPI2]	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> I2S0 (Integrated Interchip Sound 0) [Driver_SAI0]	<input type="checkbox"/>	<input type="checkbox"/>

Tabelle 3.10: Anpassungen in der File System Configuration *FS_Config_MC_0.h* (*Drive:Memory Card*).

File System Configuration

Die Konfiguration des File Systems geschieht über die Dateien *FS_Config.c (CORE)* und *FS_Config_MC_0.h (Drive:Memory Card)*. Durch die zuvor erfolgte Konfiguration der *Run-Time Environment* wurden die beiden Dateien der File System Component hinzugefügt. Die Datei *FS_Config.c (CORE)* kann bei den Standardeinstellungen belassen werden, wohingegen bei der Datei *FS_Config_MC_0.h (Drive:Memory Card)* eine wichtige Änderung vorzunehmen ist. Dies betrifft den Parameter *Memory Card Interface Mode*, welcher von *Native* auf *SPI* umgestellt wird.

Option	Blinky	SD-Karte
	Value	Value
<input checked="" type="checkbox"/> Memory Card Drive 0		
— Connected to hardware via Driver_MCI#	0	0
— Connected to hardware via Driver_SPI#	0	0
— Memory Card Interface Mode	Native	SPI
— Drive Cache Size	4 KB	4 KB
<input checked="" type="checkbox"/> Locate Drive Cache and Drive Buffer	<input type="checkbox"/>	<input type="checkbox"/>
— Filename Cache Size	0	0
— Use FAT Journal	<input type="checkbox"/>	<input type="checkbox"/>

Tabelle 3.11: Anpassungen in der File System Configuration *FS_Config_MC_0.h (Drive:Memory Card)*.

Abkürzungsverzeichnis

<i>CMSIS</i>	Cortex Microcontroller Software Interface Standard
<i>CPU</i>	Central Processing Unit
<i>IDE</i>	Integrated Development Environment
<i>MDK</i>	Microcontroller Development Kit
<i>RL – ARM</i> ...	Real-Time Library for ARM microprocessors
<i>RTE</i>	Run-Time-Environment
<i>RTOS</i>	Real Time Operating System
<i>RXF</i>	Real-time eXecution Framework

Literaturverzeichnis

- ARM Keil Group (2014). *Migrate RTX to CMSIS-RTOS*. Application Note 264.
- ARM Keil Group (2017a). *File System Component: Resource Requirements*. URL: http://www.keil.com/pack/doc/mw/FileSystem/html/fs_resource_requirements.html (besucht am 24.06.2017).
- ARM Keil Group (2017b). *MDK Microcontroller Development Kit*. URL: <http://www2.keil.com/mdk5/> (besucht am 14.05.2017).
- ARM Keil Group (2017c). *Network Component: MDK Middleware for IPv4 and IPv6 Networking*. URL: http://www.keil.com/pack/doc/mw/Network/html/nw_resource_requirements.html (besucht am 23.05.2017).
- ARM Keil Group (2017d). *Real-Time Operating System: API and RTX Reference Implementation*. URL: <http://www.keil.com/pack/doc/CMSIS/RTOS/html/threadConfig.html> (besucht am 23.05.2017).
- IBM (2017). *Rational Rhapsody family*. URL: <http://www-03.ibm.com/software/products/en/ratirhapfami> (besucht am 25.05.2017).
- Matuschek, Marco und Walter Van der Heiden (2015). *external sources*. Handout. Willert Software Tools GmbH.
- Römer, Eike (2012). *RXF Migration Guide*. Application Note. Willert Software Tools GmbH.
- Steinmeyer, Timo und Stefan Pollithy (2015). *Codegenerierung aus UML Aktivitätsdiagrammen und Implementierung einer Ethernet Schnittstelle für Embedded Systems*. Master-Projektarbeit. Hochschule Ulm.
- Van der Heiden, Walter (2016). *Modeling Embedded Systems*. Datenblatt. Willert Software Tools GmbH.
- von Schwerin, Marianne und Norbert Normann (2017). *Laboratory Guide - Embedded Systems*. Guide. Hochschule Ulm.