Intertask Kommunikation in Echtzeitbetriebsystemen

Stefan Lindörfer
stefan.lindoerfer@stud-mail.uni-wuerzburg.de

Universität Würzburg
Institut für Informatik
Lehrstuhl V - Technische Informatik
Seminar: Embedded Systems
Prof. Dr. Reiner Kolla
Wintersemester 2020/21

Abstract Die Informationsübermittlung zwischen Tasks in Echtzeitbetriebsystemen ist von fundamentaler Bedeutung und kann je nach Anforderungsszenario auf verschiedene Arten erfolgen. In der vorliegenden Arbeit wird zunächst ein Überblick über die verschiedenen Möglichkeiten gegeben sowie darauffolgend jeweils vertieft auf die Mechaniken eingegangen und durch Beispiele demonstriert.

1 Einführung

Echtzeitbetriebsysteme (auch **R**eal **T**ime **O**perating **S**ystems, kurz RTOS genannt) sind aus der Informatik nicht mehr wegzudenken und begegnen uns oft unbewusst im Alltag in zahlreich verschiedenen Anwendungsbereichen. So werden sie etwa in modernen Automobilen als Betriebssysteme eingesetzt, zur Steuerung und Regelung industrieller Anlagen verwendet oder im Verkehrswesen (z.B. Schienenverkehr- oder Ampelsteuerungen) genutzt [vgl. 1, S. 157]. Auch als Betriebssysteme von Satelliten in der Raumfahrt sowie für den Einsatz in Flugzeugen sind sie geeignet [2].

Die Besonderheit dieser Art von Betriebssystemen ist die Fähigkeit, Echtzeit-Anforderungen umsetzen zu können. Das bedeutet (anders als bei Nicht-RTOS) als Softwareentwickler die Zusicherung seitens des Betriebssystems zu besitzen, dass eine bestimmte Aufgabe innerhalb eines vordefinierten Zeitfensters entweder ausgeführt und abgeschlossen (oder unterbrochen) wird. Betriebssysteme anderer Kategorien führen ihre Aufgaben meist schnellstmöglich aus und passen kein Zeitfenster ab, garantieren also keine Echtzeit, sondern setzen auf größtmögliche Performanz.

Aufgaben in Echtzeitbetriebssystemen sind in sogenannte Tasks (oder auch Threads) unterteilt, die vom Betriebssystem gekapselt voneinander verwaltet werden. Abhängig vom jeweiligen Szenario können Tasks nicht ausschließlich getrennt voneinander operieren und ihren jeweiligen Aufgaben nachgehen, sondern ein Informationsaustausch zwischen ihnen ist erforderlich: Um etwa einen Ablauf abzubilden, bei dem zuerst Task A und anschließend Task B ausgeführt werden soll, muss eine Information von A nach B übermittelt werden können, die den entsprechenden Prozessfortschritt anzeigt sodass B beginnen kann. Oder auch wenn Daten eines Tasks von einem anderen zur (synchronisierten) Weiterverarbeitung benötigt werden, ist ein entsprechender Kommunikationskanal erforderlich um die Daten zu transferieren.

Dieser gesamte interne Informationsaustausch wird als Intertask-Kommunikation bezeichnet. Es existieren mehrere allgemeine Möglichkeiten und Techniken, Informationen und Daten zwischen Tasks auszutauschen, abhängig vom jeweiligen Szenario und den Anwendungsanforderungen. Diese Techniken werden im Folgenden aufgezeigt und nacheinander erläutert.

1.1 Überblick

Grundsätzlich unterteilt man Intertask-Kommunikation in drei Kategorien [vgl. 3, S. 79]:

1. **Synchronisation und Koordination von Tasks ohne Datentransfer**
2. **Datentransfer zwischen Tasks ohne Synchronisation**
3. **Datentransfer zwischen Tasks mit Synchronisation**

Hier eventuell die Referenzen der jeweiligen Kapitel ab-

Punkt 1 enthält anders als (2) und (3) keinen Datentransfer und unterscheidet sich von diesen - wie bereits in 1 ausgeführt - dadurch, dass lediglich eine Information ausgetauscht bzw. vom Empfänger abgefragt wird, um Tasks zu synchronisieren oder einen Ablauf umzusetzen. Während bei (2) und (3) ein Datentransfer insofern stattfindet, als das Daten ausgetauscht und vom Empfänger für die Weiterverarbeitung genutzt werden, sie also nicht für eine Ablaufsteuerung verwendet werden.

Hier fehlt noch eine Quelle

1.2 Begriffsgrundlage

Für eine genauere Betrachtung der einzelnen Kategorien aus 1.1, müssen zunächst die beiden Begriffe Koordination und Synchronisation definiert und pragmatisch voneinander abgegrenzt werden:

- **Koordination**: *„Das Integrieren und Anpassen (einer Reihe von Teilen oder Prozessen), um eine reibungslose Beziehung zueinander herzustellen.“* [vgl. 3, S. 80]
- **Synchronisation**: *„Etwas verursachen, bewegen oder ausführen, genau zur exakten Zeit.“* [vgl. 3, S. 80]

Es fällt auf, dass die Definition der Koordination keinen Bezug zur Zeit beinhaltet. Der wesentliche Unterschied zwischen Koordinierung und Synchronisierung ist somit der Zeit-Faktor. Während mit einer Koordination ein theoretisch zeitunabhängiger, sequentieller Ablauf von Tasks angestrebt wird, meint Synchronisation dagegen das zeitliche Abgleichen von Vorgängen und legt damit verstärkt Fokus auf die Kerneigenschaft von Echtzeitbetriebssystemen. Dennoch haben beide Begriffe ihre klare Daseinsberechtigung und ihren verhältnismäßigen Anwendungsspielraum.

anderes Wort!

2 Task-Interaktion ohne Datentransfer

Müssen, wie schon in 1.1 erwähnt, keine Daten im eigentlichen Sinne zwischen Tasks transferiert werden, sondern nur ein Arbeitsablauf gesteuert werden, spricht man von Task-Interaktion ohne Datentransfer. Dies kann sowohl mit dem Ziel einer Task-Synchronisation durchgeführt werden als auch ohne den bereits festgestellten Zeit-Faktor mittels Koordination, siehe 1.2. Demzufolge wird bei den Möglichkeiten dieser Kategorie auch in diese beiden Fälle unterschieden. Tabelle 1 zeigt diese Unterscheidung auf und gibt gleichzeitig einen Überblick über die jeweils zu verwendeten Konstrukte dieser Kategorie. Für das Ziel einer Koordination, werden Condition Flags verwen-

Koordination	**Synchronisation**	
Condition Flags	Event Flags	Signale
Operationen:	*Operationen:*	*Operationen:*
`Set`	`Set`	`Wait`
`Clear`	`Clear`	`Send`
`Check`	`Check`	`Check`

Tabelle 1: Koordinierungs- und Synchronisationskonstrukte [vgl. 3, S. 82]

det. Ein Flag ist ein Statusindikator, der einen bestimmten Zustand anzeigt. Ist Synchronisierung erforderlich, dass also der gesteuerte Prozess zeitkritisch auszuführen ist, dann können - je nach Anwendungsfall - Event Flags oder Signale verwendet werden. Alle drei Konstrukte und deren Operationen werden im Folgenden genauer erläutert.

2.1 Task-Koordinierung mit Condition Flags

Die einfachste Möglichkeit der Koordination ist das Condition Flag. Tabelle 1 zeigt die auf Condition Flags anwendbaren Operationen `Set` (setzen), `Clear` (zurücksetzen) und `Check` (überprüfen).

Abbildung 1 zeigt exemplarisch, wie Condition Flags verwendet werden: Task A übernimmt in diesem Fall die Steuerung des Ablaufs, während Task B darauf wartet, ausgeführt zu werden

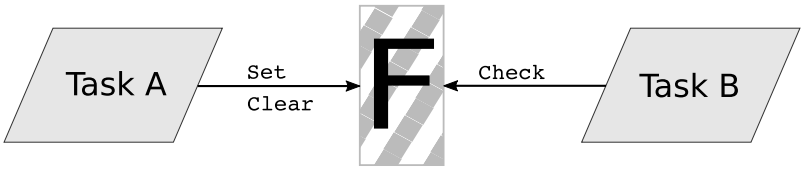


Abbildung 1: Einfache Benutzung von Condition Flags [vgl. 3, S. 83]

und fortlaufend überprüft (`Check`), ob das Flag gesetzt wurde. Im einfachsten Fall wird dafür eine globale Boolean-Variable eingesetzt, bei dieser *true* den Zustand `Set` und *false* den Zustand `Clear` repräsentieren kann. Ist es erforderlich, mehrere Zustände einzusetzen, können Aufzählungstypen (enums) verwendet werden. Diese bietet auch den Vorteil der besseren Lesbarkeit des Quellcodes, da sofort eindeutig verifizierbar ist, welcher Flag-Zustand `Set` bzw. `Clear` darstellt. Für die meisten Anforderungen ist dieses Vorgehen absolut ausreichend. [vgl. 3, S. 84]

In kritischeren Situationen jedoch, z.B. bedingt durch hohe Zugriffsraten auf ein Flag (es können sich Lese- oder Schreibfehlern ergeben) oder wenn ein höheres Maß an Ausfallsicherheit gewünscht ist, empfiehlt sich eine Mehrfachabsicherung wie sie Abbildung 2 demonstriert:

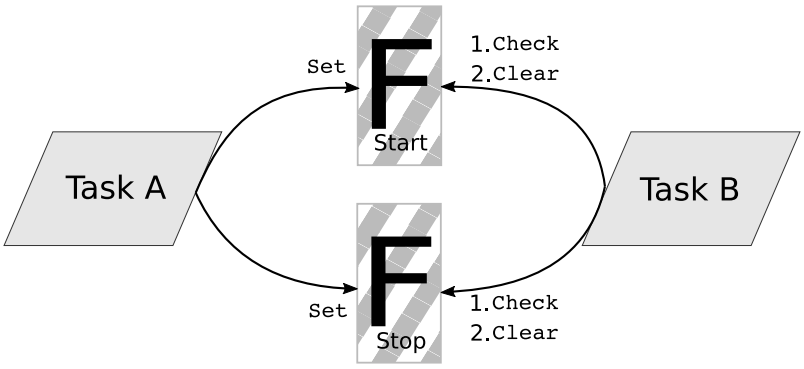


Abbildung 2: Verbesserte Benutzung von Flags zur Koordination [vgl. 3, S. 84]

Für jeden Zustand, der gesetzt werden kann (hier: Zwei), existiert ein eigenes Flag. Task A übernimmt in diesem Fall ausschließlich das Setzen der Zustände, führt dies jedoch nur durch, falls das entsprechende Flag vorher von Task B zurückgesetzt wurde [vgl. 3, S. 85]. Task B prüft jetzt zusätzlich den Zustand aller relevanten Flags (bezogen auf Abbildung 2 also alle beide) bevor auf einen `Set`-Zustand reagiert wird [vgl. 3, S. 85]. Wenn Unstimmigkeiten auftreten, weil z.B. das Prüfergebnis keinen eindeutig definierten Gesamtzustand ausweist, kann ein Fehlverhalten rechtzeitig abgefangen werden. Anders als dies bei Abbildung 1 möglich ist, denn dort kann kein Fehlverhalten vom Empfänger-Task festgestellt werden. Nach einem erfolgreichem `Check` wird das ausgeführte Flag wieder von B zurückgesetzt. Umgesetzt wird dies beispielsweise wieder mittels enums:

```
typedef enum {StartSet, StartClear} StartFlag;
typedef enum {StopSet, StopClear} StopFlag;
```

Diese Herangehensweise führt, wie bereits ausgeführt, zu einer sichereren und zuverlässigeren Abwicklung der Kommunikation und besitzt auch die Eigenschaft der besseren Lesbarkeit. An dieser Stelle wird noch erwähnt, dass eine Implementierung auf globaler Ebene ebenfalls Nachteile mit sich bringt (schlechtere Performanz, unkontrollierter Zugriff) und deswegen eine Kapselung in einem separatem Objekt eine Möglichkeit ist, diesen zu begegnen.

Hier vielleicht ein neues Unterkapitel schaffen?

Eine weitere Möglichkeit der Organisation mit Condition Flags sind Flag Gruppen. Diese bieten sich besonders dann an, wenn viele Zustände abgebildet werden sollen, die zudem logisch miteinander in Verbindung stehen können. Wenn beispielsweise im einfachen Fall etwa Zustand A nicht gleichzeitig mit Zustand B aktiv sein darf bevor Zustand C ausgelöst wird. Dabei werden einzelne

Flags auf Wortbreite zusammengefasst (auch eine geringere Breite ist möglich, z.B. durch Einsatz einer Variablen), in der jedes Bit ein Flag repräsentiert. Abbildung 2 zeigt dieses Konzept auf einer 2 Byte Variablen mit je 8 Bits.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Wert	1	0	0	1	0	1	1	1	1	0	0	1	0	1	1	0

Tabelle 2: Condition Flag Gruppe [vgl. 3, S. 85]

Wird der Wert der Variablen gelesen und als Integer (Ganzzahl) interpretiert (Abbildung 2 - Wert: 38806), kann er zudem bequem und einfach angepasst werden. Um etwa mehrere Flags, also Einzelbits, gleichzeitig zu setzen oder rückzusetzen, kann Bitmanipulation verwendet werden. Auch die Anwendung logischer Operatoren (AND, OR, XOR, NOT, $XNOR$) ist auf die einzelnen Bits möglich. Mit solchen Flag Gruppen kann daher beispielsweise leicht die logische Aussage

$$((([15] \wedge ([13] \vee \neg[11])) \oplus [5]) \equiv 1) \tag{1}$$

implementiert und evaluiert werden. Indem eine Bitmaske, die die relevanten Bits (hier: 15, 13, 11 und 5) markiert, mit dem AND-Operator auf den Variableninhalt angewendet wird erhält man eine aussagekräftigere Bitfolge:

```
  1001011110010110  (38806)
& 1010100000100000  (43040)
--------------------
  1000000000000000  (32768)
```

Das Ergebnis enthält für unmarkierte Bits stets eine 0 und für markierte eine 1, sofern das jeweilige Bit der Variablen gesetzt war, ansonsten ebenfalls eine 0 und kann damit komfortabel für weitere Vergleiche herangezogen werden. Beispielsweise erfüllt (die binäre Schreibweise von) 32768 bereits die oben angegebene Gleichung (1). Für ausführlichere Informationen zu diesem Thema sei auf die entsprechende Fachliteratur verwiesen.

Vielleicht hier noch Bit Pattern erwähnen und Link zu leftiCus einbauen

Zusammengefasst sind Condition-Flags ein gutes Konstrukt um einen Ablauf von Tasks zu koordinieren. Es bieten sich für verschiedenste Anforderungsprofile einfache und trotzdem leistungsstarke Lösungen an, um diesen Ablauf zu realisieren und die Kommunikation sicher und zuverlässig zu gestalten.

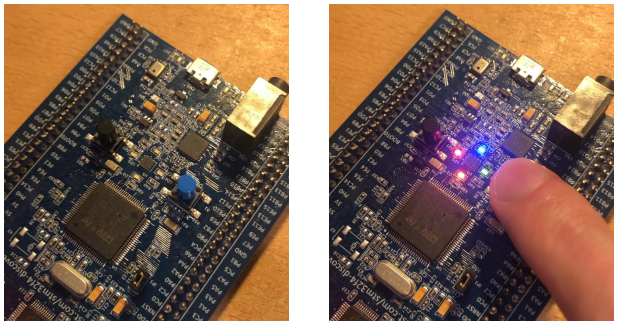
2.2 Task-Synchronisation über Event Flags

Event Flags übernehmen viele Eigenschaften von Condition Flags wie sie in 2.1 vorgestellt wurden. Auf sie sind wie in Tabelle 1 dargestellt, auch die gleichen Operationen anwendbar: `Set`, `Clear` und `Check`. Event Flags können anders als Condition Flags trotzdem zur Task-Synchronisierung eingesetzt werden. Allerdings ist dies - wie noch gezeigt wird - auf eine Flussrichtung beschränkt. Die im Vergleich zu Condition Flags neu hinzukommende Komponente ist das Ereignis (Event).

aperiodisch hat er mit Schwingungen zu tun. Eventuell anderes Wort für "nicht-periodisch" finden

Ereignisse können aperiodisch und unvorhersehbar auftreten und - je nach Szenario - muss entsprechend schnell auf sie reagiert werden können, das entsprechende koordinierte Verhalten also möglichst synchron zum Zeitpunkt des Events ausgelöst werden. Um dies zu erreichen, wird eine **I**nterrupt **S**ervice **R**outine (ISR) eingesetzt, die auf das Eintreten eines Events wartet und - so lange keines eingetreten ist - sich in einem suspendierten Zustand befindet [vgl. 3, S. 87]. Ein Beispiel für ein solches Ereignis kann ein Hardware-Interrupt sein wie er in Abbildung 3 gezeigt wird: Erst mit gedrücktem bzw. gehaltenem blauen Button leuchten die vier LEDs auf (3b).

Der schematische Ablauf mit Event Flags ist mit Abbildung 1 durchaus vergleichbar, nur dass jetzt Task A durch eine ISR ersetzt wird und eine Eingabe in Form des ausgelösten Ereignisses bekommt bevor das Event Flag gesetzt wird [vgl. 3, S. 87]. Als weiterführende Lektüre zum Thema Interrupt und Interrupt Service Routine sei [4] oder weitere Fachliteratur empfohlen. Weiterhin spielt



(a) suspendierte ISR (b) aktive ISR

Abbildung 3: Beispiel von Event Flags für Interrupt-Management auf STM32F4 Discovery-Board

für die hier gewünschten Task-Synchronisierung der Zeitraum, in dem der Empfänger - hier Task B - das Event Flag überprüft eine größere und wichtigere Rolle: Für eine dynamische Reaktion auf ein Ereignis ist ein entsprechend kleines, periodisches Zeitfenster erforderlich. In der in Abbildung 3 dargestellte Demonstration prüft Task B beispielsweise alle 500 Millisekunden ob das zugehörige Event Flag gesetzt ist (also der Button gedrückt wird). Ein weiter mit einer 2 Sekunden-Periode durchgeführter Versuch funktionierte ebenfalls, zeigte aber kein flüssiges Verhalten und kann für dieses Szenario ungeeignet sein. Hier muss von Anwendungsfall zu Anwendungsfall verhältnismäßig entschieden werden. Um das Betriebsystem zu entlasten, wird Task B nach jeder negativen Überprüfung bis zur nächsten Iteration suspendiert, womit Ressourcen für andere Aufgaben frei werden.

Die Anwendung der unter 2.1 vorgestellten Techniken zu Condition Flags (z.B. Flag Gruppen) sind auch auf das Konstrukt der Event Flags übertragbar. Somit ist sowohl eine verbesserte und sichere Implementierung wie auch eine für eine größere und ggf. auch verknüpfte Anzahl Event Flags vorhanden. [vgl. 3, S. 87–88]

Wie eingangs bereits erwähnt, funktioniert diese Synchronisierung lediglich in eine Richtung: Die Interrupt Service Routine wird durch das vordefinierte Event (Interrupt) geweckt und setzt das entsprechende Flag, was vom Empfänger-Task im Rahmen seiner regelmäßigen Überprüfung registriert wird. Ist die Prüfungsperiode entsprechend den Anforderungen gewählt, wird eine mindestens angenäherte Synchronisierung von Ereignis und Reaktion bzw. ISR und Empfänger-Task erreicht. Sollen zwei oder mehrere Tasks synchronisiert werden, bei der jeder beteiligte Task sowohl Sender als auch Empfänger sein kann, ist eine andere Methodik anzuwenden, die im nächsten Abschnitt vorgestellt wird.

2.3 Task-Synchronisation über Signale

Mit den bisher vorgestellten Techniken - Condition Flag und Event Flag - lässt sich bereits eine Koordination bzw. eine Einbahnstraßen-Synchronisation implementieren. Bei einer Problemstellung, bei der etwa mehrere Tasks nach und nach in eine Art Wartestellung gehen (etwa weil ihre Aufgaben erfüllt sind) und alle zu einem gleichen (synchronen) Zeitpunkt erneut starten sollen, helfen diese Konstrukte allerdings nicht weiter. Hierfür gibt es das Konzept der Signale.

Die in Tabelle 1 angegebenen Operationen (`Wait`, `Send` und `Check`) lassen bereits einen anderen Umgang als bei Flags erahnen: Eine zentrale Stelle (diese kann sowohl das Betriebssystem selbst bereitstellen oder auch ein eigenes konzipiertes Objekt) übernimmt die Signalverwaltung. Bei ihr registrieren sich die beteiligten Tasks zunächst, wenn eine der o.g. Operationen ausgeführt wird und diese übernimmt auch die Steuerung des Signals. Abbildung 4 stellt die drei auftretenden Möglichkeiten vor.

Zunächst sollen bei (a) und (b) zwei Tasks synchronisiert werden. Beide Darstellungen unterscheiden sich lediglich in der Reihenfolge, welche Operation zuerst auf ein Signal gewirkt wird: Der Task, welcher sich zuerst mit einer Operation meldet, wird suspendiert bis er zusammen mit dem anderen Task das synchronisierte Signal zur Wiederaufnahme erhält. Beide starten dann synchron. Option (c) zeigt, wie ein einzelner Task synchronisiert wird, ohne bekanntes Mitspiel anderer

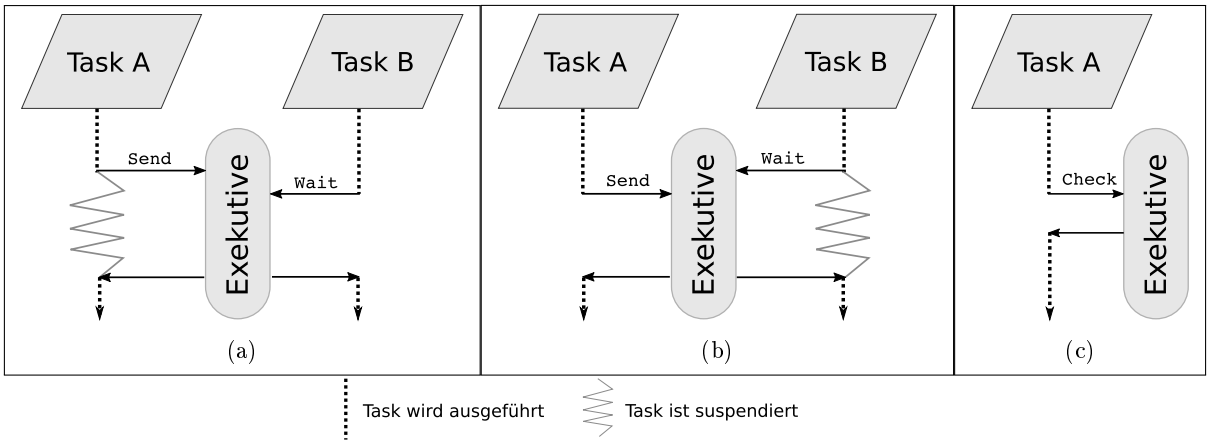


Abbildung 4: Task-Synchronisation mittels Signalen [vgl. 3, S. 90]

Tasks: Eine Überprüfung des Signalstatus erfolgt und ggf. eine Suspendierung oder eine sofortige Wiederaufnahme. Hierbei kann das Signal (Exekutive) auch eine interne Logik besitzen und zum Timing verschiedener einzelner oder mehrerer Tasks benutzt werden.

Hier fehlt noch was: mehr ausführen!

Die Umsetzung von Signalen erfolgt meist über Funktionen, wie im folgenden mit C beispielhaft skizziert:

```
// Sendestatus an Signal; wird suspendiert wenn niemand darauf wartet
void Send(SyncSignal SignalName);
// Wartet auf Signal; kein Sendestatus - suspendiert, sonst - fortsetzen
void Wait(SyncSignal SignalName);

// Prueft ob ein Task wartet; 'true' wenn Signal den Sendestatus besitzt
typedef enum {false, true} bool;
bool Check(SyncSignal SignalName);
```

noch mehr hier! Nochmal Buch lesen!

Diese Technik synchronisiert - anders als Event Flags - in beide (mehrere) Richtungen.

3 Datentransfer ohne Synchronisation oder Koordination

Ist ein Datenaustausch zwischen Tasks erforderlich, aber nicht an Koordinierungs- und Synchronisationskriterien gebunden, unterliegt also weder einer zeitlichen Vorgabe noch einem Ablauf, werden verschiedene Datenstrukturen zum Austausch zwischen Tasks verwendet.

3.1 Überblick

3.2 Pools

3.3 Queues

4 Task-Synchronisation mit Datentransfer

Daten, die mit einer zeitlichen Priorität zwischen Tasks ausgetauscht werden, also genau zum richtigen Zeitpunkt beim Empfänger bereit stehen müssen, werden ebenfalls mit einer Datenstruktur transferiert. An diese besteht jedoch ein höherer Anspruch, da sie die Synchronisation sicherstellen muss.

4.1 Mailbox

5 Zusammenfassung

Literatur

[1] Marco Winzker. *Elektronik für Entscheider: Grundwissen für Wirtschaft und Technik*. Vieweg, Wiesbaden, 2008.

[2] Lehrstuhl VIII Universität Würzburg Institute für Informatik. *Rodos - Lehrstuhl für Informatik VIII*. Aufgerufen am 31.12.2020. o. D. URL: https://www.informatik.uni-wuerzburg.de/aerospaceinfo/wissenschaftforschung/rodos/.

[3] Jim Cooling. *Real-time Operating Systems: Book 1 - The Theory*. Independently Published, 2017.

[4] *Interrupt - Mikrocontroller.net*. Aufgerufen am 02.01.2021. o. D. URL: https://www.mikrocontroller.net/articles/Interrupt.