Intertask Kommunikation in Echtzeitbetriebssystemen

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

Seminar: Embedded Systems
bei Prof. Dr. Reiner Kolla
Wintersemester 2020/21

Abstract Die Informationsübermittlung zwischen Tasks in Echtzeitbetriebssystemen ist von fundamentaler Bedeutung und kann je nach Anforderungsszenario auf verschiedene Arten erfolgen. In der vorliegenden Arbeit wird ein Überblick über die verschiedenen Möglichkeiten der Koordinierung und Synchronisierung von Tasks gegeben und anschließend vertieft gezeigt, wie Daten zwischen Tasks ausgetauscht werden können.

1 Einführung

Echtzeitbetriebssysteme (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 Luftfahrzeugen sind sie geeignet [vgl. 2] [vgl. 3].

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 ggf. abgeschlossen (oder unterbrochen) wird. Betriebssysteme anderer Kategorien (z.B. Microsoft Windows) führen ihre Aufgaben meist schnellstmöglich bzw. ohne Vorhersagemöglichkeit aus und passen kein Zeitfenster ab, garantieren also keine Echtzeit, sondern setzen auf größtmögliche Performanz [vgl. 4, S. 317].

Aufgaben in Echtzeitbetriebssystemen sind in sogenannte Tasks (oder auch Threads) unterteilt, die vom Betriebssystem getrennt voneinander verwaltet werden. Abhängig vom jeweiligen Szenario können Tasks jedoch 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 zunächst eine Information von A nach B übermittelt werden können, die den entsprechenden Prozessfortschritt anzeigt, sodass B beginnen kann. Aber 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. 5, S. 79]:

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

Punkt 1 enthält anders als 2 und 3 keinen Datentransfer und unterscheidet sich von diesen - wie bereits in der Einleitung ausgeführt - dadurch, dass lediglich eine Information ausgetauscht bzw. vom Empfänger abgefragt wird, um Tasks zu synchronisieren oder einen Arbeitsablauf 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 zwingend für eine Ablaufsteuerung verwendet werden [vgl. 5, S. 80].

1.2 Begriffsunterscheidung

Für eine genauere Betrachtung der einzelnen Kategorien aus Kapitel 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.“* [5, S. 80]
- **Synchronisation**: *„Etwas verursachen, bewegen oder ausführen, genau zur exakten Zeit.“* [5, 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 [vgl. 5, S. 80]. 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 in dieser Thematik ihre klare Daseinsberechtigung und auch ihre bevorzugten Anwendungsgebiete in denen sie am besten zum Tragen kommen.

2 Task-Interaktion ohne Datentransfer

Müssen, wie schon in Kapitel 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 Synchronisation durchgeführt werden als auch zeit unkritisch als Koordination, siehe 1.2. Demzufolge wird bei den Methoden 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.

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. 5, S. 82]

Grundsätzlich ist ein Flag (dt. Flagge) ein Statusindikator, der einen bestimmten Zustand anzeigt. Im simpelsten Fall sind das 0 und 1. Für das Ziel einer Koordinierung, werden sogenannte Condition Flags verwendet. Ist Synchronisierung erforderlich, dass also der gesteuerte Prozess zeitkritisch auszuführen ist, können - je nach Anwendungsfall - Event Flags oder Signale verwendet werden. Alle drei Konzepte werden im Folgenden detaillierter erläutert und Implementierungsansätze aufgezeigt.

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 und fortlaufend in regelmäßigen Abständen

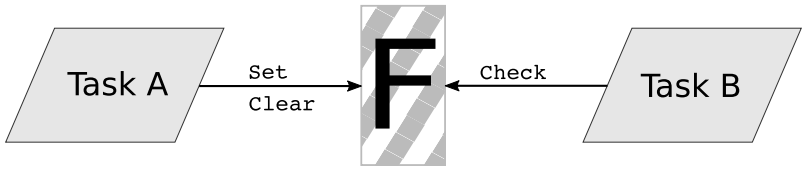


Abbildung 1: Einfache Benutzung von Condition Flags [vgl. 5, S. 83]

überprüft (`Check`), ob das zugehörige Flag gesetzt wurde. Im einfachsten Fall wird dafür eine globale Boolean-Variable eingesetzt, bei dieser *true* den Zustand ‚gesetzt' und *false* den Zustand ‚zurückgesetzt' repräsentieren kann. Ist es erforderlich, mehrere Zustände einzusetzen, können Aufzählungstypen wie Enums verwendet werden. Diese bieten auch den Vorteil der besseren Lesbarkeit des Quellcodes, da sofort verifizierbar ist, welcher Flag-Zustand ‚gesetzt' bzw. ‚zurückgesetzt' darstellt. Für die meisten Anforderungen ist dieses Vorgehen der Implementierung von Condition Flags absolut ausreichend [vgl. 5, S. 84].

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

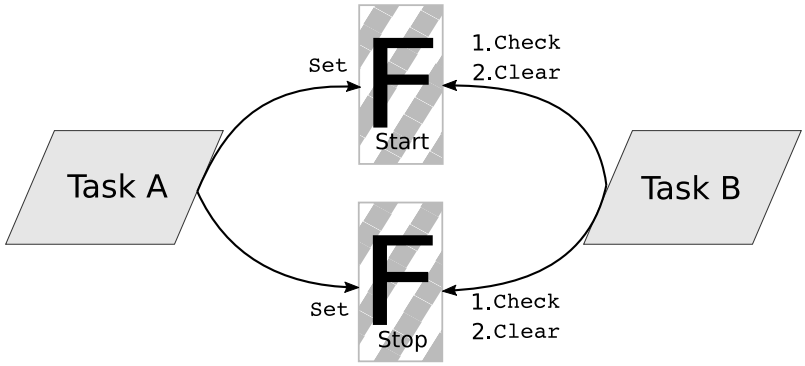


Abbildung 2: Verbesserte Benutzung von Condition Flags zur Koordination [vgl. 5, S. 84]

Für jeden Zustand, der gesetzt werden kann (hier: zwei), existiert jetzt ein eigenes Condition Flag. Task A übernimmt in diesem Fall ausschließlich das Setzen der Zustände, führt diese Operation jedoch nur durch, falls das entsprechende Flag vorher von Task B zurückgesetzt wurde [vgl. 5, S. 85]. Task B prüft im Unterschied zur Abbildung 1 jetzt zusätzlich den Zustand aller relevanten Flags (bezogen auf Abbildung 2 also beide) bevor auf einen `Set`-Zustand reagiert wird

[vgl. 5, 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 u.U. kein Fehlverhalten vom Empfänger-Task festgestellt werden, da keine Absicherungsinstanz existiert, mit der verglichen werden könnte. Nach einem erfolgreichem `Check` wird das ausgeführte Flag wieder von Task B zurückgesetzt. Umgesetzt wird diese Methode wieder mittels Enums, wie folgender Codeabschnitt für Abbildung 2 zeigen soll:

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

Diese Herangehensweise führt, wie bereits hervorgehoben, zu einer sichereren und zuverlässigeren Abwicklung der Koordination und besitzt auch die bereits festgestellte Eigenschaft der besseren Lesbarkeit, was nicht zu unterschätzen ist. An dieser Stelle sei noch erwähnt, dass eine Implementierung auf globaler Ebene ebenfalls Nachteile mit sich bringen kann (schlechtere Performanz, unkontrollierter Zugriff). In objektorientierten Programmiersprachen sollte deswegen eine Kapselung in einem separatem Objekt in Erwägung gezogen werden.

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 etwa in einem einfachen Fall Zustand A (Flag A) nicht gleichzeitig mit Zustand B (Flag B) gesetzt sein darf. Dabei werden einzelne Flags auf Wortbreite zusammengefasst [vgl. 5, S. 85] (auch eine geringere Breite ist möglich, z.B. durch Einsatz einer Variablen geringerer Breite), in der jedes Bit ein Flag repräsentieren kann. 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

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

$$((([b_{15}] \wedge ([b_{13}] \vee \neg[b_{11}])) \oplus [b_5]) \equiv 1) \tag{1}$$

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 aus Tabelle 2 angewendet wird, erhält man eine gefilterte Bitfolge:

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

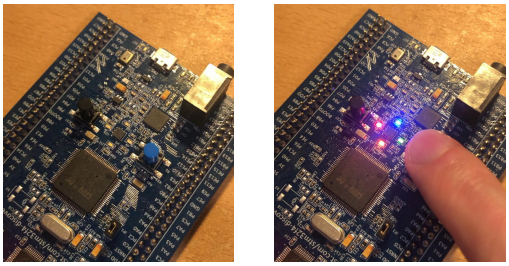
Das Ergebnis enthält für unmarkierte Bits stets eine 0 und für markierte eine 1, sofern das jeweilige Bit der Variablen gesetzt war, ansonsten ebenfalls eine 0. Dieses Resultat kann damit komfortabel für weitere Vergleiche herangezogen werden. Beispielsweise erfüllt 32768 bereits die oben angegebene Gleichung (1). Ein vollwertiger C++-Code für sogenanntes Bit Pattern, eine noch effizientere Möglichkeit der Analyse, ist unter [6] zu finden. Mit dieser Methode entfällt quasi der Filterungsprozess und der Variableninhalt kann sozusagen bereits vollständig in die Gleichung eingesetzt werden, während bei ersterer Methode, das Ergebnis nach dem Filtern noch untersucht werden muss. Für weitere ausführlichere Informationen zu diesem Thema sei auf die entsprechende Fachliteratur verwiesen.

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

2.2 Einseitige Task-Synchronisation über Event Flags

Event Flags übernehmen viele Eigenschaften von Condition Flags wie sie in Kapitel 2.1 vorgestellt wurden. Auf sie sind, wie in Tabelle 1 dargestellt, auch die gleichen Operationen anwendbar: `Set` (setzen), `Clear` (zurücksetzen) und `Check` (überprüfen). Event Flags können anders als Condition Flags trotzdem zur Task-Synchronisation eingesetzt werden. Allerdings ist die erreichte Synchronisierung - wie noch gezeigt wird - auf eine Richtung beschränkt [vgl. 5, S. 87]. Die im Vergleich zu Condition Flags neu hinzukommende Komponente ist das Ereignis (Event).

Ereignisse können asynchron 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 [7]. 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. 5, 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 (siehe Abb. 3b).



(a) suspendierte ISR (b) aktive ISR

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

Der schematische Ablauf mit Event Flags ist mit Abbildung 1 durchaus vergleichbar, nur dass Task A durch eine ISR ersetzt wird und eine Eingabe in Form des ausgelösten Ereignisses bekommt bevor das Event Flag gesetzt (oder rückgesetzt) wird [vgl. 5, S. 87]. Als weiterführende Lektüre zum Thema Interrupt und Interrupt Service Routine wird [7] oder weitere Fachliteratur empfohlen. Dabei muss es sich nicht zwingend um ein Hardware-Interrupt handeln, auch auf andere, benutzerdefinierte Ereignisse kann so reagiert werden.

Weiterhin spielt für die hier gewünschten Task-Synchronisierung, also der Zeitraum, in dem der Empfänger Task B das Event Flag überprüft (`Check`), eine größere und wichtigere Rolle: Für eine dynamische Reaktion auf das Ereignis ist ein entsprechend kleines, periodisches Zeitfenster erforderlich. In Abbildung 3 dargestellter Demonstration prüft Task B beispielsweise alle 500 Millisekunden ob das zugehörige Event Flag gesetzt ist. Ein darüber hinaus durchgeführter Versuch mit einer zwei-Sekunden-Periode funktionierte prinzipiell ebenfalls, zeigte aber kein flüssiges Verhalten von Aktion und Reaktion und kann deswegen für dieses Szenario ungeeignet sein. Hier muss von Anwendungsfall zu Anwendungsfall seperat entschieden werden.

Um das Betriebsystem zu entlasten, wird Task B nach jeder negativen Überprüfung bis zur nächsten Periode suspendiert, womit Ressourcen für andere Aufgaben frei werden.

Die Anwendung der unter Kapitel 2.1 vorgestellten Techniken zu Condition Flags (z.B. Flag Gruppen) sind auch auf das Konstrukt der Event Flags übertragbar [vgl. 5, S. 87–88]. Somit ist sowohl eine zuverlässigere und sicherere Implementierung wie auch für eine größere und ggf. auch verknüpftere Anzahl Event Flags vorhanden.

Wie eingangs bereits erwähnt, funktioniert diese Synchronisierung lediglich in eine Richtung: Die Interrupt Service Routine wird durch das vordefinierte Event geweckt und setzt das entsprechende Flag, welches vom Empfänger-Task im Rahmen seiner regelmäßigen Überprüfung registriert wird. Ist die Überprü-

fungsperiode entsprechend den Anforderungen gewählt, wird eine Synchronisierung von Ereignis und Reaktion bzw. ISR und Task B erreicht. Sollen zwei oder mehrere Tasks synchronisiert werden, bei der jeder beteiligte Task sowohl Sender als auch Empfänger sein kann, ist eine andere Methodik zu wählen. Diese wird im nächsten Abschnitt vorgestellt.

2.3 Beiderseitige Task-Synchronisation über Signale

Mit der bisher vorgestellten Technik, der Event Flags, lässt sich bereits eine einseitige Synchronisation implementieren. Bei einer Problemstellung aber, bei der zwei oder 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, hilft dieses Konstrukt allerdings nicht weiter. Dafür gibt es das Konzept der Signale.

Die in Tabelle 1 angegebenen Operationen `Wait` (warten), `Send` (senden) und `Check` (überprüfen) lassen bereits einen anderen Umgang als bei den bisher vorgestellten Flags aus Kapitel 2 erahnen: Eine zentrale Stelle (diese kann etwa das Betriebssystem bereitstellen) übernimmt die Signalverwaltung. Bei ihr registrieren sich die beteiligten Tasks sobald eine der o.g. Operationen ausgeführt wird (vgl. Abb. 4) und diese übernimmt auch die Steuerung dieses Konstrukts. Damit ist das Signal keinem spezifischem Task zugeordnet und auch keine direkt von diesem ausgehende Synchronisation [vgl. 5, S. 91]. Genau dieser Umstand ermöglicht eine beiderseitige/mehrseitige Synchronisierung. Abbildung 4 stellt die drei auftretenden Möglichkeiten vor.

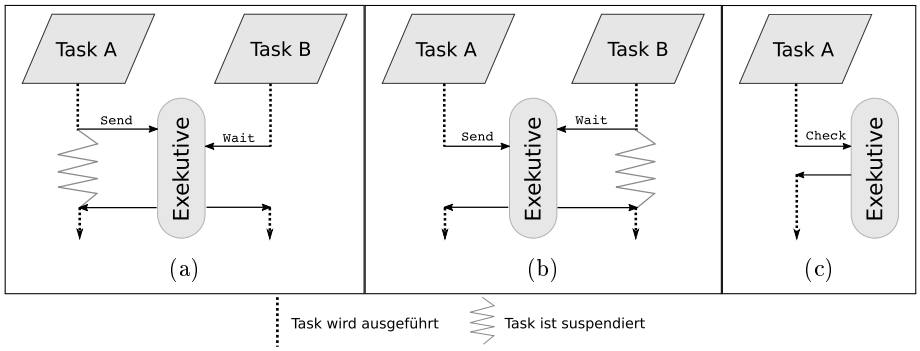


Abbildung 4: Task-Synchronisation mittels Signalen [vgl. 5, S. 90]

Zunächst sollen in den Abbildungen 4a und 4b 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 Signal zur Wiederaufnahme erhält. Beide starten dann synchron zum gleichen Zeitpunkt [vgl. 5, S. 90–91]. Der sichtbare zeitliche Abstand bevor beide Tasks erneut starten, nachdem sie zur Synchronisierung gemeldet sind, kann darin begründet sein, dass zuerst ein Zeitfenster abgewartet werden soll oder ein Jitter (unvorhersehbare Verzögerung) auftritt, der den Start verzögert.

Abbildung 4c zeigt, wie ein einzelner Task synchronisiert wird, ohne bekanntes Mitspiel anderer Tasks: Eine Überprüfung des Signalstatus erfolgt und ggf. eine Suspendierung oder eine sofortige Wiederaufnahme. Hierbei kann die Signalverwaltung (Exekutive) auch eine interne Logik besitzen und zum Timing verschiedener einzelner oder mehrerer Tasks benutzt werden [vgl. 5, S. 91].

Die Umsetzung von Signalen erfolgt normalerweise über Funktionen [vgl. 5, S. 91], wie im folgenden mit der Programmiersprache C skizziert:

```
/* Sendestatus auf das Signal */
void Send(SyncSignal SignalName);

/* Warte-Anfrage an Signal */
void Wait(SyncSignal SignalName);

/* Prueft ob ein Task wartet */
typedef enum {false, true} bool; // Def. fuer C hilfreich
bool Check(SyncSignal SignalName);
```

Zusammengefasst ist dabei folgendes, aus Abbildung 4 abgeleitetes Verhalten in den Funktionen zu implementieren um Signale vollständig umzusetzen:

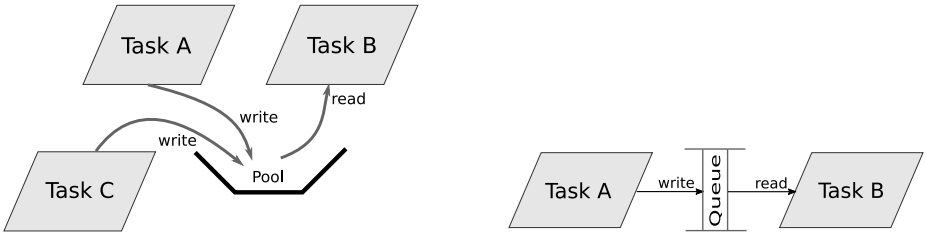
- `Send`: Ausführender Task wird suspendiert, sofern kein anderer Task bereits eine `Wait`-Anfrage gestellt hat.
- `Wait`: Wird das Signal nicht bereits gesendet, wenn diese Anfrage gestellt wird, wird betreffender Task ebenfalls suspendiert. Andererseits wird der „sendende“ Task reaktiviert und beide Tasks werden synchron wieder ausgeführt.
- `Check`: Gibt *true* zurück, wenn das Signal gesendet wird, ansonsten *false*.

Somit sind Signale eine Möglichkeit, um in zwei oder mehrere Richtungen zu synchronisieren, da hier jeder Task sowohl Sender als auch Empfänger des Synchronisationsprozesses sein kann [vgl. 5, S. 91]. Diese Technik wird allerdings von eher wenigen Echtzeitbetriebssystemen als Möglichkeit zur Synchronisierung angeboten [vgl. 5, S. 91].

3 Datentransfer ohne Synchronisation oder Koordination

Ist ein regelmäßiger Datenaustausch zwischen Tasks erforderlich (z.B. weil Daten zur Weiterverarbeitung benötigt werden), die aber nicht zur Koordination verwendet und ebenfalls nicht an zeitliche Kriterien gebunden sind, wird dies über verschiedene Datenverarbeitungsmechanismen bewerkstelligt. Diese lassen

sich primär in zwei Kategorien unterteilen. Wie noch gezeigt wird, basieren sie auf allgemein bekannten Datenstrukturen.



(a) Direkter Zugriff (b) Sequentieller Zugriff

Abbildung 5: Datentransfer zwischen Tasks [vgl. 5, S. 95]

Abbildung 5 zeigt die Unterscheidung der hier aufgeführten zwei Prinzipien „direkter Zugriff“ (5a) und „sequentieller Zugriff“ (5b). Während bei ersterem eine Vielzahl an Tasks die gleiche Datenstruktur verwenden können, erfolgt bei zweiterem eine Eins-zu-Eins-Weiterleitung. Damit besitzt jeder Datentyp auch seine Daseinsberechtigung und wird im folgenden genauer vorgestellt.

3.1 Pools

Pools sind per Definition ein „Read-Write Random Access Data Store“ [5, S. 94]. Das bedeutet: Pools bieten einen direkten Zugriff, erlauben also den Zugang auf alle darin befindlichen Elemente. Außerdem besitzen sie eine variable Größe, ist also theoretisch nicht auf eine bestimmte, feste Anzahl an möglichen beinhalteten Elementen beschränkt. Er kann aber nur vordefinierte Objekte vom gleichen Typ speichern. Folgende Codedarstellung soll dieses Prinzip in C++ verdeutlichen:

```
struct Data {
        char* id;
        int value;
};

Pool<Data> fifoList;
```

Ein Typ *Data* wird deklariert, welcher einen *char*-Zeiger sowie einen Integer (Ganzzahl) enthält. *fifoList* kann nach der Deklarierung nur Objekte dieses Typs aufnehmen und verwalten. Wie noch ausgeführt wird, bieten Pools allerdings - abhängig von der Implementierung - den Vorteil, dass sie nicht in einem zusammenhängenden Speicherblock untergebracht sein müssen. Hier sei der Vollständigkeit halber noch angemerkt, dass das Objekt, welches vom *char*-Zeiger referenziert wird, nicht im Speicherbereich des Pools liegt. Dessen Speicheradresse und der Integer allerdings schon.

Eine weitere Eigenschaft eines Pools ist damit die (Daten-)Kapselung: Außerhalb existierender Code bzw. Tasks erhalten lediglich Zugriff auf die Lese- und Schreib-Methoden des Pools, haben aber keinerlei Einfluss auf die interne Organisation der Daten oder Informationen über den Speicherort dieser.

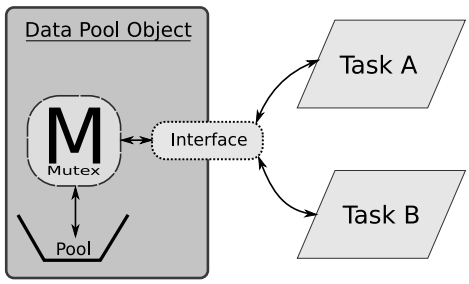


Abbildung 6: Ein mit Mutex geschützter Pool [vgl. 5, S. 95]

Diese Kapselung soll in Abbildung 6 dargestellt werden, sie zeigt außerdem den schematischen Aufbau dieses Datentyps: Über eine Schnittstelle (Interface), die mindestens die o.g. (Lese-/Schreib-)Methoden bereitstellt, erfolgt die Kommunikation bzw. die Datenübermittlung in beide Richtungen. Um Schreib- oder Lesefehler zu vermeiden, können Pools intern einen sogenannten Mutex, also einen Sicherungsmechanismus verwenden, der keinen mehrfach Zugriff auf Daten erlaubt. Abbildung 6 zeigt dessen Position und Funktionsweise: Tasks die während eines gerade stattfindenden Zugriffs ebenfalls Daten erlangen möchten, werden suspendiert und erhalten nach der internen Freigabe durch den Mutex nacheinander anschließend Zugriff auf die Daten. Dies ist vergleichbar mit einer Schrankenabfertigung im Straßenverkehr [vgl. 5, S. 95].

Allgemein bekannte Datenstrukturen die sich Pools zuzuordnen lassen, sind etwa FIFO (**F**irst **I**n **F**irst **O**ut) und LIFO (**L**ast **I**n **F**irst **O**ut) Listen (basierend auf dem Prinzip verketteter Listen) sowie Heaps in verschiedenen Ausführungen. Gerade verkettete Listen bieten den bereits erwähnten Vorteil, dass ein Pool nicht auf einem zusammenhängenden Speicherblock liegen muss, sondern die einzelnen Listenelemente - durch Zeiger miteinander verbunden - verteilt im Speicher liegen können. Das macht Pools zu einer für das Speichermanagement flexibleren Datenstruktur, da erstens nicht die gesamte Speichermenge von Anfang an bereitgestellt werden muss und das Bereitstellen eines größeren Speicherblockes auch eine Herausforderung des Speichermanagements sein kann [vgl. 5, S. 96].

3.2 Warteschlangen

Warteschlangen (Queues) eigenen sich ebenfalls für die Datenübertragung zwischen Tasks, unterscheiden sich aber in diversen Punkten von den bereits vorge-

stellten Pools. Zunächst besitzen Warteschlangen in diesem Kontext eine feste Größe: Sobald sie im Code deklariert werden, sind sie in ihrer Größe bzw. in der Anzahl Elemente, die sie aufnehmen können, nicht mehr veränderbar. Hier muss deshalb eine vernünftige Größe gewählt werden, da schließlich nicht unendlich viel Speicher zur Verfügung steht. Des weiteren sind sie am ehesten für die Task-zu-Task-Datenübertragung konzipiert und vergleichbar mit einem einspurigem Verbindungstunnel (siehe Abb. 5b) [vgl. 5, S. 96]: Ein Task legt Daten in der Warteschlange ab, ein anderer liest diese daraus. Damit ergeben sich die beiden Funktionen einer Warteschlange: `Dequeue` und `Enqueue`, die entweder ein Objekt aus der Warteschlange zurückgeben oder ihr ein neues hinzufügen.

Intern werden - neben einem N-großen Feld (z.B. Array) - zwei Zeiger gespeichert, die wichtige Punkte innerhalb der Warteschlange markieren: Das Feld, an dem das nächste Element eingefügt wird (Abbildung 7: schwarzer Pfeil) und das Feld, des Objekts, das als nächstes entnommen wird (7: weißer Pfeil; bei FIFO zugleich auch das älteste Element). Außerdem wird ein Zähler gespeichert, der Auskunft über die Anzahl Elemente in der Warteschlange gibt.

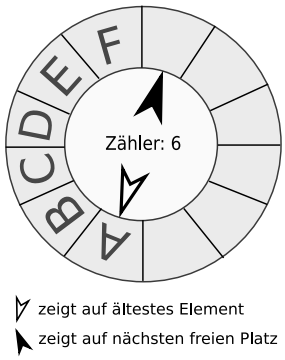


Abbildung 7: Prinzip einer FIFO-Warteschlange als Ring Buffer dargestellt [vgl. 5, S. 97]

Hier wird nun beispielhaft die Umsetzung als Ring Buffer (auch Circular Buffer) nach dem FIFO Prinzip gezeigt: Zur Initialisierung zeigen beide Zeiger auf das erste Feld des internen Arrays. Neue Elemente werden mit `Enqueue` der Warteschlange im Uhrzeigersinn hinzugefügt und der schwarze Zeiger jeweils angepasst (siehe Abb. 7). Wird `Dequeue` aufgerufen, wird das älteste Element der Warteschlange entnommen und der weiße Zeiger entsprechend angepasst.

Werden nach und nach mehr Element eingefügt als insgesamt Plätze vorhanden sind, wird jedes Mal das älteste Element der Warteschlange überschrieben und auch die Zeiger entsprechend angepasst.

Damit entsteht eine relativ einfach zu implementierende Datenstruktur für die Task-zu-Task-Kommunikation mit der verschiedene Organisationsprinzipien (FIFO/LIFO) leicht umgesetzt werden können.

4 Task-Synchronisation mit Datentransfer

Daten, die mit einer zeitlichen Priorität zwischen Tasks ausgetauscht werden müssen, also genau zum richtigen Zeitpunkt (oder schnellstmöglich) beim Empfänger bereit stehen müssen, werden ebenfalls mit einer Datenstruktur transferiert. An diese besteht jedoch ein höherer Anspruch, da sie die synchrone Auslieferung sicherstellen und auch Daten verwalten muss [vgl. 5, S. 99]. Die Daten müssen also in einem vordefinierten Zeitfenster beim Empfänger eingehen. Dies kann auch so gedeutet werden, als das die Daten schnellstmöglich beim Empfänger-Task verfügbar sein müssen. Es ist die höchste Kunst der Datenübermittlung zwischen Tasks und eine Kerneigenschaft von Echtzeitbetriebssystemen. Die für diese Aufgabe zugedachte Datenstruktur heißt Mailbox. Sie kann auf verschiedene Arten funktionieren, welche nun aufgezeigt werden.

4.1 Mailbox

Für die synchronisierte Datenübermittlung in Echtzeitbetriebssystemen gibt es einige Datenstrukturen, die unter dem Begriff Mailbox zusammengefasst werden. Eine Mailbox wird grundsätzlich mit folgenden beiden Befehlen bzw. entsprechenden Funktionalitäten ausgestattet: `Post` und `Pend`. Die Schemata 8a und 8b ähneln nicht zufällig denen aus Kapitel 2.3 (Signale). Im Grunde lässt sich eine Mailbox bereits mithilfe einer Datenstruktur, wie sie in Kapitel 3 vorgestellt wurden und einem Signal vollständig umsetzen [vgl. 5, S. 99].

Demzufolge sind die Kommandos einer Mailbox in diesem Fall auch von denen der Signale abgeleitet: `Post` bedeutet hier, dass eine Benachrichtigung an die Mailbox geschickt wird, die von einem anderen Task entgegengenommen werden kann. Diese Aufnahmebereitschaft zeigt der Empfänger-Task mit dem Kommando `Pend` an (ähnlich zu `Wait`). Hat sich der Empfänger-Task noch nicht bei der Mailbox zur Abholung gemeldet, wird der Sender suspendiert. Auch umgekehrt ist dies möglich, wie Abbildung 8b zeigt: Hier wird zuerst nach einer Nachricht gefragt und bis diese vorliegt, wird der entsprechende Task suspendiert. Besteht letztendlich eine synchrone Verbindung zweier Tasks über die Mailbox, wird die betreffende Nachricht ausgetauscht. Meistens wird ein Zeiger übergeben um den Ressourceneinsatz zu minimieren [vgl. 5, S. 99]. Abbildung 8c zeigt eine andere Herangehensweise zur synchronen Datenübermittlung: Während bei 8a und 8b eine synchrone Übermittlung das Ziel ist, soll hier eine schnellstmögliche Datenübermittlung erreicht werden. Diese wird teilweise in Echtzeitbetriebssystemen nach dem Publisher / Subscriber-Prinzip umgesetzt: Eine öffentliche Datenstruktur, etwa ein sogenanntes Topic, vergleichbar mit einem Kanal, wird eingerichtet. In dieses Topic können Tasks Daten schreiben (publishen). Genauso können andere Tasks diese Datenstruktur abonnieren (subscriben) und erhalten bei neu

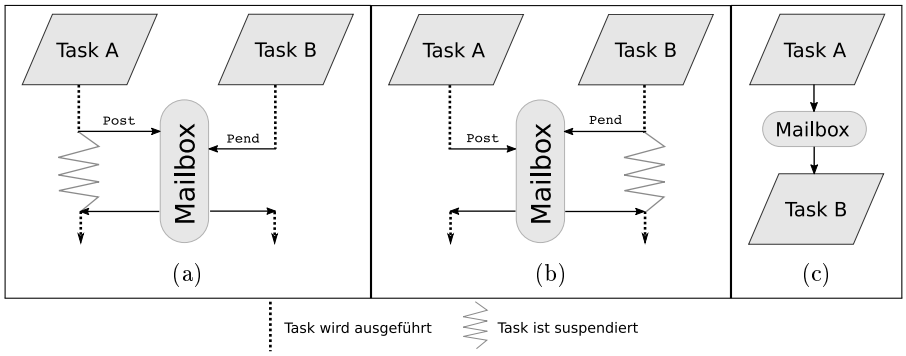


Abbildung 8: Datentransfer mit Task-Synchronisation [vgl. 5, S. 99]

hinzugefügten Daten eine Benachrichtigung. Dies kann z.B. als Ereignis umgesetzt werden (indem z.B. dann eine vordefinierte Funktion aufgerufen wird). Damit liegt keine direkte Verbindung der Tasks vor, sondern es werden nur Benachrichtigungen über die Datenstruktur selbst ausgetauscht.

Um eine grobe Vorstellung von der Dauer einer synchronen Datenübermittlung in einem Echtzeitbetriebssystem zu vermitteln, wurde ein Messversuch unternommen: Dabei wurde auf Grundlage des Betriebssystems Xubuntu das Echtzeitbetriebssystem RODOS [3] verwendet. Das ausgeführte Programm verwendete die Datenstruktur eines Topics (siehe oben) und nutzte das Publisher / Subscriber-Prinzip um Daten zwischen zwei Tasks synchron auszutauschen. Dabei wurde von Task A die momentane Systemzeit in das Topic veröffentlicht. Task B erhielt diese und bildete die Differenz zur aktualisierten Systemzeit, womit die Übertragungsdauer sichtbar wird. Bei 23 aufgenommenen Messwerten lies sich somit ein Durchschnittswert von

$$t_{Dauer} = (544 \pm 26)\ ns \tag{2}$$

ermitteln. Dies ist selbstverständlich keine allgemeingültige Aussage über die Dauer von Datenübermittlungen zumal es sich hierbei nur um geringe Datengrößen handelt, es vermittelt aber einen Eindruck der zeitlichen Größenordnung (Nanosekunden), wie schnell sich Daten über eine Mailbox austauschen lassen.

Mit der hier entwickelten Datenstruktur einer Mailbox und deren Abwandlungen können so relativ leicht Daten synchron zwischen Tasks ausgetauscht werden. Zum Beispiel über eine Mailbox die u.a. auf Signalen basiert oder mittels des Publisher / Subscriber-Prinzips.

5 Zusammenfassung

Task-Kommunikation ist ein unabdingbares Instrument in Echtzeitbetriebssystemen und in beinahe jeder Anwendung auf diesem Gebiet erforderlich. Es wurde

abhängig von verschiedenen Szenarien aufgezeigt, welche Art der Kommunikation jeweils am besten ist und wie diese umgesetzt werden könnte.

So erreicht man Task-Koordination - die Steuerung eines Ablaufes - am besten mit Condition Flags, eine einseitige-Synchronisierung mit Event Flags und kann beiderseits Synchronisierung über Signale erreichen. Weiterhin wurden verschiedene Möglichkeiten gezeigt, wie diese Methoden implementiert werden können - abhängig von den entsprechenden Anwendungsanforderungen.

Auch Daten von einem Task zu anderen (oder mehreren) zu übermitteln ist eine wichtige Aufgabe und kann wahlweise mit der Datenstruktur eines Pools oder einer Warteschlange bewerkstelligt werden. Echtzeitbetriebssysteme bieten oft von Haus aus entsprechende Typen an, wobei eine eigene Implementierung aufgrund der geringen Komplexität nicht allzu schwierig ist.

Die Königsdisziplin, eine synchronisierte Datenübertragung zwischen zwei oder mehreren Tasks, erfolgt über das Konstrukt der Mailbox und ist in einigen Echtzeitbetriebssystemen als Publisher / Subscriber-Prinzip umgesetzt, welches den Empfänger automatisch bei Eintreffen neuer Daten benachrichtigt, sodass diese zum richtigen Zeitpunkt oder schnellstmöglich weiterverarbeitet werden können.

Mit dem in dieser Arbeit vorgestelltem Handwerkszeug an Techniken ist das Thema der Intertask-Kommunikation in Echtzeitbetriebssystemen übersichtlich zusammengefasst und jedes größere Anwendungsszenario abgedeckt.

Literatur

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

[2] Planetary Transportation Systems. *Our Partners - University of Wuerzburg*. Aufgerufen am 31.12.2020. 2017. URL: `https://www.pts.space/partners/universitat-wurzburg/`.

[3] Universität Würzburg - Lehrstuhl für Informatik VIII - Informationstechnik für Luft- und Raumfahrt. *RODOS*. Aufgerufen am 31.12.2020. o.D. URL: `https://www.informatik.uni-wuerzburg.de/aerospaceinfo/wissenschaftforschung/rodos/`.

[4] Uwe Brinkschulte Heinz Wörn. *Echtzeitsysteme*. Springer-Verlag, 2005.

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

[6] Jason Turner. *High Performance Bit Pattern*. Aufgerufen am 03.01.2021. Juli 2020. URL: `https://github.com/lefticus/cpp_weekly/commits/master/HighPerfBitPattern`.

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