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, wie Tasks koordiniert und synchronisiert bzw. Daten zwischen ihnen ausgetauscht werden können sowie darauffolgend jeweils vertieft auf die einzelnen Mechaniken eingegangen und Implementierungsansätze aufgezeigt.

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**

Hier eventuell die Referenzen der jeweiligen Kapi-

3. **Datentransfer zwischen Tasks mit Synchronisation**

Punkt 1 enthält anders als (2) und (3) keinen Datentransfer und unterscheidet sich von diesen - wie bereits in Kapitel 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.

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]

Für das Ziel einer Koordination, werden Condition Flags verwendet. 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.

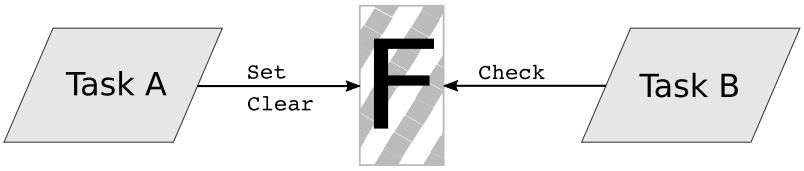


Abbildung 1: Einfache Benutzung von Condition Flags [vgl. 3, S. 83]

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 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 wie 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 Schreibfehler ergeben) oder wenn ein höheres Maß an Ausfallsicherheit gewünscht ist, empfiehlt sich eine Mehrfachabsicherung wie sie Abbildung 2 demonstriert:

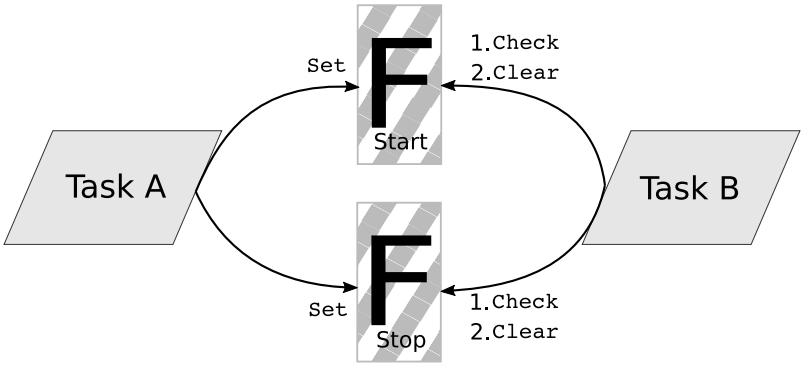


Abbildung 2: Verbesserte Benutzung von Flags zur Koordination [vgl. 3, S. 84]

Für jeden Zustand, der gesetzt werden kann (hier: Zwei), existiert ein eigenes Condition 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, da keine Vergleichsinstanz existiert. 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 bringen kann (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 letzte 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 in einem einfachen Fall etwa Zustand A (Flag A) nicht gleichzeitig mit Zustand B (Flag 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 beispielsweise daher 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 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 und kann damit komfortabel für weitere Vergleiche herangezogen werden. Beispielsweise erfüllt (die binäre Schreibweise von) 32768 bereits die oben angegebene Gleichung (1). Ein vollwerter C++-Code für sogenanntes Bit Pattern, eine noch effizientere Möglichkeit der Analyse, ist unter [4] zu finden. 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 Anforderungsprofile einfache und trotzdem leistungsstarke Lösungen an, um diesen Ablauf zu realisieren und die Kommunikation sicher und zuverlässig zu gestalten.

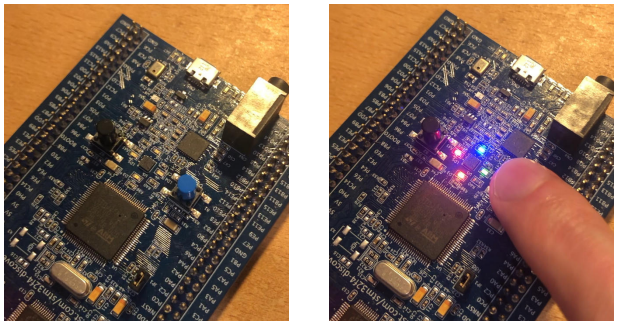
2.2 Task-Synchronisation über Event Flags

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

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

Interrupt **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 es in Abbildung 3 gezeigt wird: Erst mit gedrücktem bzw. gehaltenem blauen Button leuchten die vier LEDs auf (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 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 [5] oder weitere Fachliteratur empfohlen. Weiterhin spielt für die hier gewünschten Task-Synchronisierung, also 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 Zwei-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 Periode 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` (warten), `Send` (senden) und `Check` (überprüfen)) 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.

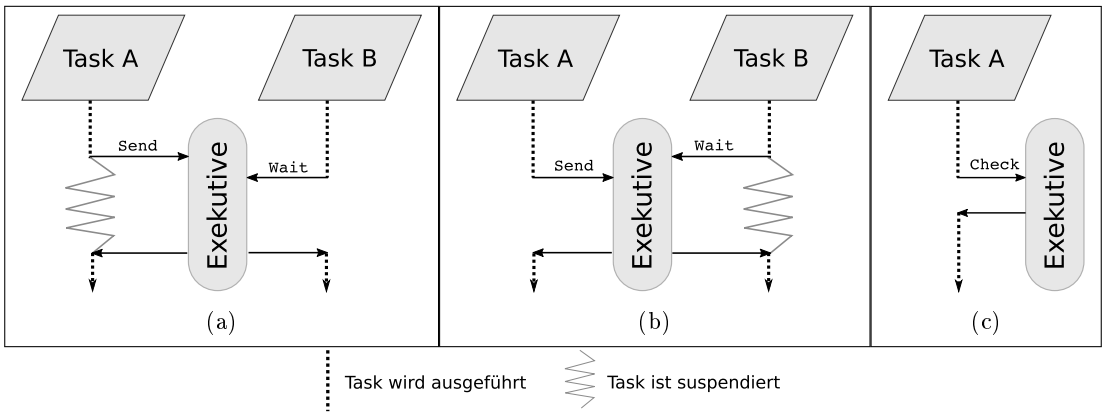


Abbildung 4: Task-Synchronisation mittels Signalen [vgl. 3, S. 90]

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 synchronisierte Signal zur Wiederaufnahme erhält. Beide starten dann synchron zum gleichen Zeitpunkt. Option 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.

Hier fehlt noch was: mehr ausführen!

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

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

/* Warte-Anfrage auf 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`: Sendender 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 der Task 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 gute Methode, um in zwei oder mehrere Richtungen zu synchronisieren, da hier jeder Task sowohl Sender als auch Empfänger des Synchronisationsprozesses sein kann. Diese Technik synchronisiert - anders als Event Flags - in beide (oder mehrere) Richtungen.

noch mehr hier! Nochmal Buch lesen!

3 Datentransfer ohne Synchronisation oder Koordination

Ist ein Datenaustausch zwischen Tasks erforderlich (zum Beispiel weil Daten zur Weiterverarbeitung benötigt werden), aber nicht an Koordinierungs- und Synchronisationskriterien gebunden, wird dies über verschiedene Datenverarbeitungsmechanismen bewerkstelligt. Diese lassen sich primär zwei Kategorien zuordnen und unterliegen also weder einer zeitlichen Beschränkung noch einem Ablauf. Sie sind deswegen unkomplizierter in der Benutzung und leichter zu implementieren.

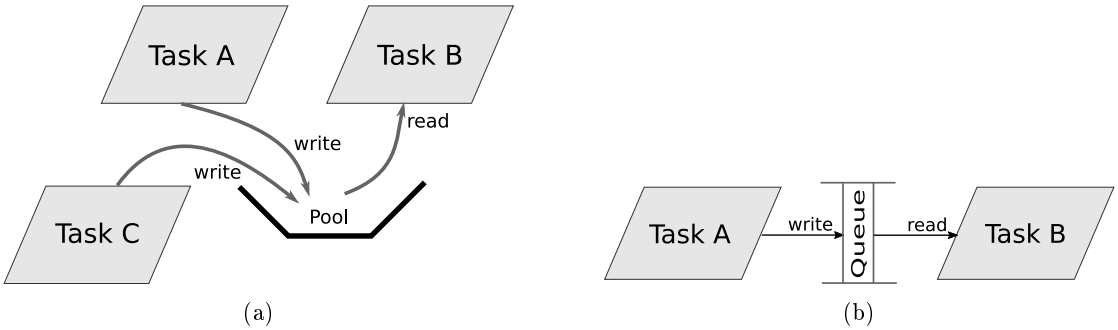


Abbildung 5: Datentransfer zwischen Tasks [vgl. 3, S. 95]

Abbildung 5 zeigt die Unterscheidung auf die hier angewendeten zwei Prinzipien „zufälliger Zugriff "5a und „sequentieller Zugriff "5b. Während bei ersterem eine Vielzahl an Tasks den gleichen Datentyp verwenden können, erfolgt bei zweiterem eine Eins-zu-eins-Weiterleitung (von Task zu Task). Damit besitzt jeder Datentyp seine Daseinsberechtigung und wird im folgenden genauer vorgestellt.

3.1 Pools

Pools sind per Definition ein Read-Write Random Access Data Store [vgl. 3, S. 94]. Das bedeutet: Ein Pool besitzt eine variable Größe, ist also theoretisch nicht auf eine bestimmte, feste Anzahl an möglichen Speicherelementen beschränkt, kann aber nur vordefinierte Objekte speichern (die die gleiche Speichergröße besitzen). Es ist also vereinfacht ausgedrückt beispielsweise nicht möglich, einen Integer und einen String in einem Pool zu 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 enthält. Der Pool *fifoList* kann nur Objekte dieses Typs aufnehmen und verwalten.

Eine weitere Eigenschaft von Pools ist die (Daten-)Kapselung: Außerhalb existierender Code bzw. Tasks bekommt lediglich Zugriff auf die Lese- und Schreib-Methoden des Pools, haben aber keinerlei Einfluss auf die interne Organisation oder Informationen über den Speicherort der Daten und können diese auch nicht ohne direkte Kommunikation mit öffentlichen Methoden des Pools lesen oder ändern.

Abbildung 6 zeigt den schematischen Aufbau dieses Datentyps: Über eine Schnittstelle (Interface), die mindestens die o.g. (R/W) Methoden bereitstellt, erfolgt die Kommunikation bzw. die Datenübermittlung in beide Richtungen. Um Schreib- oder Lesefehler zu vermeiden, verwenden Pools intern einen sogenannten Mutex, also einen Sicherungsmechanismus, der keinen mehrfachen Zugriff auf Daten erlaubt. 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, vergleichbar mit einer Schrankenabfertigung im

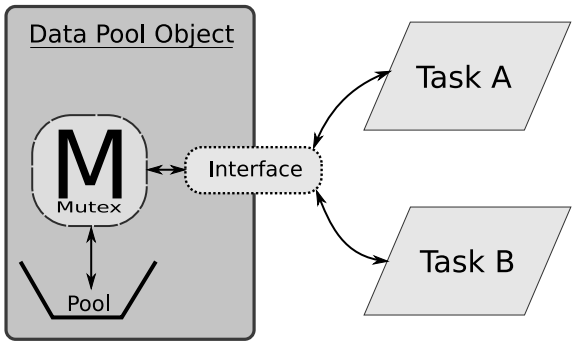


Abbildung 6: Ein mit Mutex geschützter Pool [vgl. 3, S. 95]

Straßenverkehr.

Allgemein bekannte Speicherorganisationskonzepte für Pools sind etwa FIFO (First In First Out) und LIFO (Last In First Out) Listen sowie Heaps.

hier noch ordentlich abschließen

3.2 Warteschlangen

Warteschlangen (Queues) eigenen sich ebenfalls für die Datenübertragung zwischen Tasks, unterscheiden sich - bezogen auf diesen Anwendungsbereich - aber in diversen Punkten von den bereits vorgestellten Pools. Zunächst besitzen Warteschlangen hier eine feste Größe. Sobald sie im Code deklariert werden, sind sie in ihrer Größe nicht mehr veränderbar. Des weiteren sind sie am ehesten für die Task-zu-Task-Datenübertragung konzipiert und bildlich vergleichbar, mit einem Einbahnstraßen-Verbindungstunnel: Ein Task legt Daten in der Warteschlange ab, ein anderer liest diese daraus (5b). Daraus ergeben sich die beiden Funktionen `Dequeue` und `Enqueue`, die entweder ein Objekt aus der Wartschlange zurückgeben oder ihr ein neues hinzufügen.
Abbildung 7 verdeutlicht diese Funktionsweise einer Warteschlange. Intern werden gekapselt zwei

Abbildung 7: Prinzip einer FIFO-Warteschlange mit fester Größe [vgl. 3, S. 97]

Indizes (in Form von vorzeichenlosen Integern oder Zeigern) gespeichert, die zwei wichtige Punkte innerhalb der (FIFO-)Warteschlange markieren: Das Feld, an dem das nächste Element eingefügt wird und das Feld, des Objekts, das als nächstes entnommen wird. Außerdem wird ein Zähler gespeichert, der Auskunft über die Anzahl Elemente in der Warteschlange gibt. Damit gilt es, zwei Fälle gesondert zu behandeln: Beiden Fällen ist gemein, dass beide Indizes auf das gleiche Feld zeigen.

schauen ob cooling im buch auch diese unterscheidung in logische macht

- Ist der Zähler bei 0, die Warteschlange ist also leer, darf kein weiteres Element mehr entnommen werden können.
- Ist der Zähler größer 0, die Warteschlange somit vollständig befüllt und es muss - um weiterhin eine FIFO-Warteschlange zu sein -, das älteste Element im Falle eines Hinzufügens überschrieben werden.

Eine mögliche Version der Implementierung ist, diesen Datentyp als Ring Buffer umzusetzen:

4 Task-Synchronisation mit Datentransfer

hier gehts auch noch weiter!

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. Die Daten müssen also in einem definierten Zeitfenster beim Empfänger eingehen. Meistens wird dies so umformuliert, als das die Daten schnellstmöglich beim Empfänger-Task verfügbar sein müssen. Dies ist die höchste Kunst der Datenübermittlung zwischen Tasks und eine Kerneigenschaft von Echtzeitbetriebssystemen. Die für diese Aufgabe gedachte Datenstruktur heißt Mailbox. Sie kann auf verschiedene Arten funktionieren, welche nun vorgestellt werden.

hier bin ich auch noch nicht glücklich

mailbox wird aus signale und datenstruktur zusammengesetzt! ausformulieren!

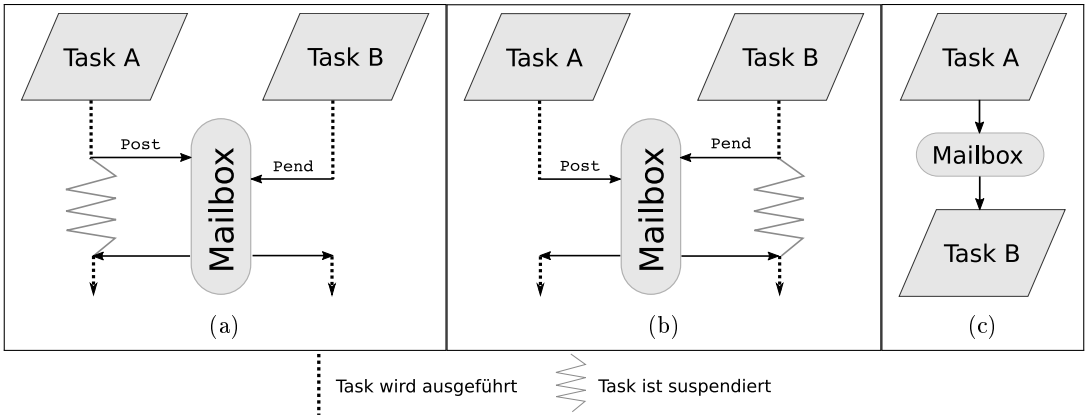


Abbildung 8: Datentransfer mit Task-Synchronisation [vgl. 3, S. 99]

5 Zusammenfassung

Task-Kommunikation ist ein unabdingbares Instrument in Echtzeitbetriebssystemen und in beinahe jeder Anwendung auf diesem Gebiet erforderlich. Es wurde aufgezeigt, für welche Fälle welche Art der Kommunikation am besten ist und wie diese best möglichst umgesetzt wird.

So erreicht man Task-Koordination, die Steuerung eines Ablaufs, am besten mit Condition Flags, eine einseitige-Synchronisierung mit Event Flags und kann beiderseite Synchronisierung über Signale umsetzen. Weiterhin wurden verschiedene Möglichkeiten gezeigt, wie diese Methoden implementiert werden können - abhängig von der erforderlichen Entwicklungsstrategie.

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

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

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

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

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