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
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 zunächst ein Überblick über die verschiedenen Möglichkeiten der Koordinierung und Synchronisierung von Tasks gegeben und gezeigt, wie Daten zwischen Tasks ausgetauscht werden können. Anschließend wird jeweils vertieft auf die einzelnen Mechaniken eingegangen und Implementierungsansätze aufgezeigt.

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 Anwendungsgebieten. 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 ggf. abgeschlossen (oder unterbrochen) wird. Betriebssysteme anderer Kategorien führen ihre Aufgaben meist schnellstmöglich bzw. ohne Vorhersagemöglichkeit aus und passen kein Zeitfenster ab, garantieren also keine Echtzeit, sondern setzen auf größtmögliche Performanz [vgl. 3, 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. 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. 4, 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 Kapitel 1 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. 4, S. 80].

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.“* [4, S. 80]
- **Synchronisation**: *„Etwas verursachen, bewegen oder ausführen, genau zur exakten Zeit.“* [4, 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. 4, 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 ihre bevorzugten Anwendungsbereiche in denen sie zum Tragen kommen.

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. 4, S. 82]

Grundsätzlich ist ein Flag (Flagge) ein Statusindikator, der einen bestimmten Zustand anzeigt. Im simpelsten Fall sind das 0 und 1. Für das Ziel einer Koordination, werden sogenannte Condition Flags verwendet. 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 detaillierter erläutert.

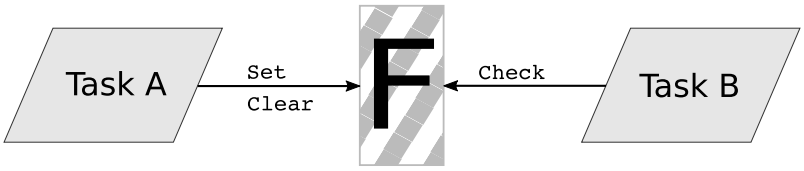


Abbildung 1: Einfache Benutzung von Condition Flags [vgl. 4, S. 83]

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 in regelmäßigen Abständen 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 bieten auch den Vorteil der besseren Lesbarkeit des Quellcodes, da sofort eindeutig ablesbar ist, welcher Flag-Zustand `Set` bzw. `Clear` darstellt. Für die meisten Anforderungen ist dieses Vorgehen der Implementierung von Condition Flags absolut ausreichend [vgl. 4, 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:

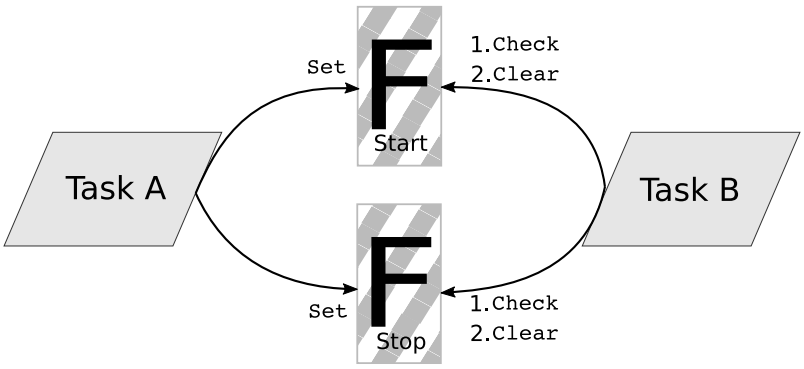


Abbildung 2: Verbesserte Benutzung von Condition Flags zur Koordination [vgl. 4, S. 84]

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. 4, 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. 4, 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 kann. Nach einem erfolgreichem `Check` wird das ausgeführte Flag wieder von B zurückgesetzt. Umgesetzt wird dies wieder mittels Enums, wie folgender Codeabschnitt für Abbildung 2 zeigt:

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

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

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) aktiv sein darf. 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 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 [vgl. 4, 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

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

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. Dieses Resultat kann damit komfortabel für weitere Vergleiche herangezogen werden. Beispielsweise erfüllt 32768 bereits die oben angegebene Gleichung (1). Ein vollwerter C++-Code für sogenanntes Bit Pattern, eine noch effizientere Möglichkeit der Analyse, ist unter [5] zu finden. Mit dieser Methode kann der Variableninhalt sozusagen bereits vollständig in die Gleichung eingesetzt werden, während bei erster, das Ergebnis nach dem Filtern noch untersucht werden muss. Für weitere ausführlichere Informationen zu diesem Thema sei auf die entsprechende Fachliteratur verwiesen.

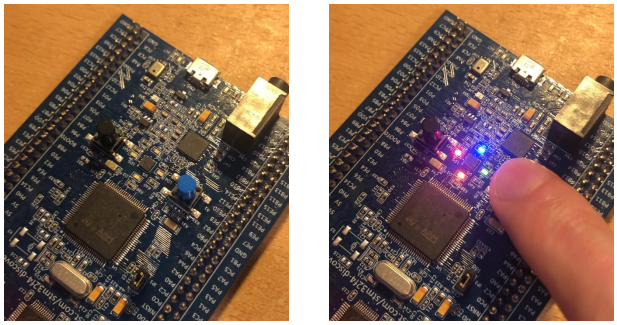
Zusammengefasst sind Condition-Flags ein gutes Konstrukt um einen Ablauf von Tasks zu koordinieren. Es bieten sich für verschiedenste Anforderungen einfache und trotzdem leistungsstarke Lösungen an, um einen Ablauf zu realisieren und die Kommunikation sicher und zuverlässig zu gestalten.

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` (setzen), `Clear` (zurücksetzen) und `Check` (überprüfen). Event Flags können anders als Condition Flags

trotzdem zur Task-Synchronisierung eingesetzt werden. Allerdings ist die erreichte Synchronisierung - wie noch gezeigt wird - auf eine Richtung beschränkt. 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. 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. 4, 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. 4, S. 87]. Als weiterführende Lektüre zum Thema Interrupt und Interrupt Service Routine wird [6] oder weitere Fachliteratur empfohlen. Dabei muss es sich nicht zwingend um ein Hardware-Interrupt handeln, auch andere, benutzerdefinierte Ereignisse können hier verwendet werden. Weiterhin spielt für die hier gewünschten Task-Synchronisierung, also der Zeitraum, in dem der Empfänger - hier Task B - das Event Flag überprüft (`Check`), 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. 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 [vgl. 4, S. 87–88]. Somit ist sowohl eine verbesserte und sicherere Implementierung wie auch für eine größere und ggf. auch verknüpfte Anzahl Event Flags vorhanden.

Wie eingangs bereits erwähnt, funktioniert diese Synchronisierung lediglich in eine Richtung: Die Interrupt Service Routine wird durch das vordefinierte Event (z.B. Interrupt) geweckt und setzt das entsprechende Flag, was vom Empfänger-Task im Rahmen seiner regelmäßigen Überprüfung registriert wird. Ist die Überprüfungsperiode entsprechend den Anforderungen gewählt, wird eine mindestens angenäherte 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 Task-Synchronisation über Signale

Mit der bisher vorgestellten Technik, der Event Flags, lässt sich bereits 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, hilft dieses Konstrukt 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 den bisher vorgestellten 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 sobald eine der o.g. Operationen ausgeführt wird und diese übernimmt auch die Steuerung des Signals. Damit ist das Signal keinem spezifischem Task zugeordnet und auch keine von diesem ausgehende Synchronisierung [vgl. 4, S. 91] Abbildung 4 stellt die drei auftretenden Möglichkeiten vor.

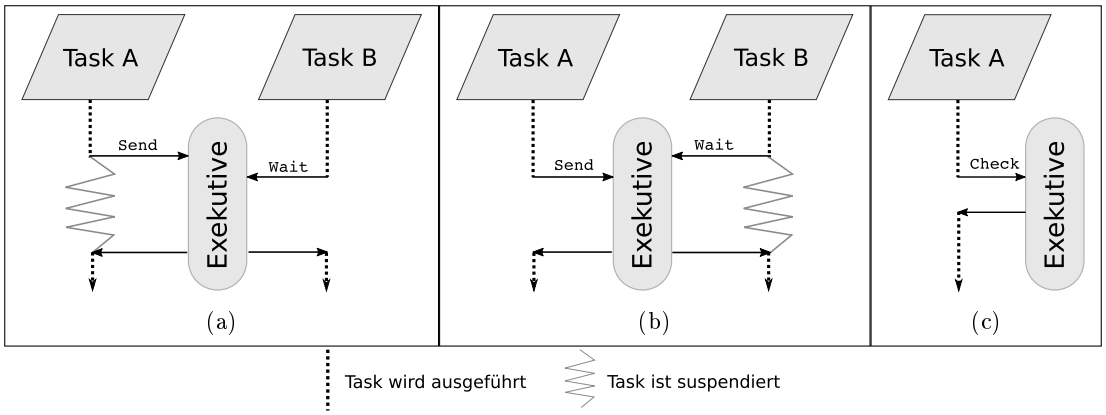


Abbildung 4: Task-Synchronisation mittels Signalen [vgl. 4, S. 90]

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 [vgl. 4, 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. 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 [vgl. 4, S. 91].

Die Umsetzung von Signalen erfolgt normalerweise über Funktionen [vgl. 4, S. 91], 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 wird aber von eher wenigen Echtzeitbetriebssystemen als native Implementierung angeboten [vgl. 4, S. 91].

3 Datentransfer ohne Synchronisation oder Koordination

Ist ein Datenaustausch zwischen Tasks erforderlich (zum Beispiel 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. Durch die fehlenden scharfen Kriterien einer Synchronisation oder Koordination, sind sie unkomplizierter in der Benutzung und leichter zu implementieren.

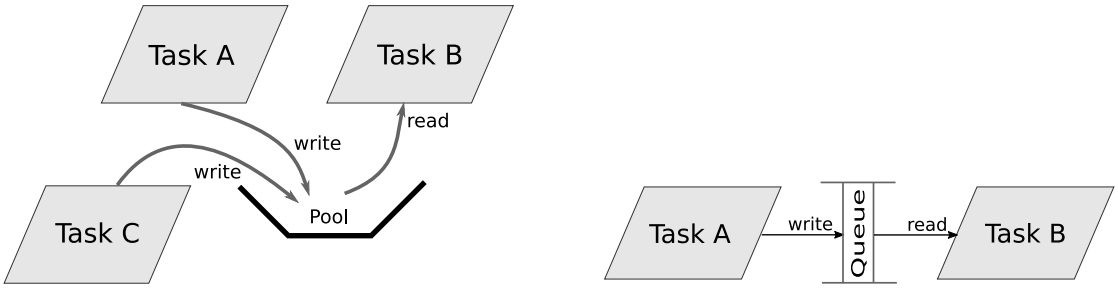


Abbildung 5: Datentransfer zwischen Tasks [vgl. 4, S. 95]

Abbildung 5 zeigt die Unterscheidung der hier aufgeführten zwei Prinzipien „zufälliger 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 (von Task zu Task). 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 [vgl. 4, S. 94]. Das bedeutet: Ein Pool besitzt eine variable Größe, ist also theoretisch nicht auf eine bestimmte, feste Anzahl an möglichen beinhalteten Elementen beschränkt, kann aber nur vordefinierte Objekte speichern (die stets die gleiche Speichergröße besitzen). 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. Wie noch gezeigt wird, bieten Pools allerdings - abhängig von der Implementierung - den Vorteil, dass sie nicht in einem zusammenhängenden Speicherblock untergebracht sein müssen, sondern die Daten im Speicher verteilt liegen können. Hier sei noch angemerkt, dass das Objekt, welches vom *char*-Zeiger referenziert wird, nicht im Speicherbereich des Pools liegen muss und in einem anderen Teil des Speichers liegen kann.

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 und können auch nicht ohne direkte Kommunikation mit öffentlichen Methoden des Pools lesen oder schreiben.

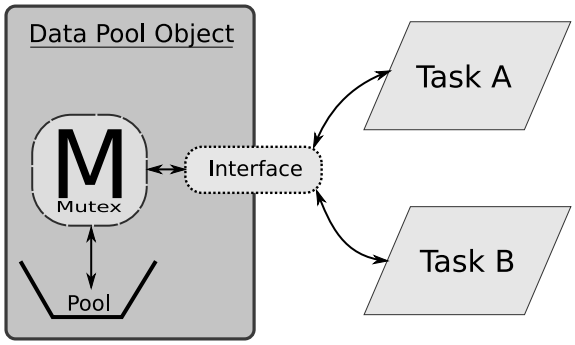


Abbildung 6: Ein mit Mutex geschützter Pool [vgl. 4, S. 95]

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. (R/W) Methoden bereitstellt, erfolgt die Kommunikation bzw. die Datenübermittlung in beide Richtungen. Um Schreib- oder Lesefehler zu vermeiden, verwenden Pools intern gern einen sogenannten Mutex, also einen Sicherungsmechanismus, der keinen mehrfach 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. Dies ist vergleichbar mit einer Schrankenabfertigung im Straßenverkehr [vgl. 4, 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 verkettete 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, im Speicher verteilt liegen können. Das macht Pools zu einer für das Speichermanagement flexibleren Datenstruktur [vgl. 4, S. 96].

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 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 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 bildlich vergleichbar mit einem Einbahnstraßen-Verbindungstunnel: Ein Task legt Daten in der Warteschlange ab, ein anderer liest diese daraus (5b). Damit ergeben sich die beiden Funktionen `Dequeue` und `Enqueue`, die entweder ein Objekt aus der Warteschlange zurückgeben oder ihr ein neues hinzufügen. Abbildung ?? verdeutlicht diese Funktionsweise einer Warteschlange.

Intern werden - neben einem n-großen Feld (Array) - gekapselt zwei 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 (im Falle einer FIFO-Struktur gleich dem ältesten Element). Außerdem wird ein Zähler gespeichert, der Auskunft über die Anzahl Elemente in der Warteschlange gibt.

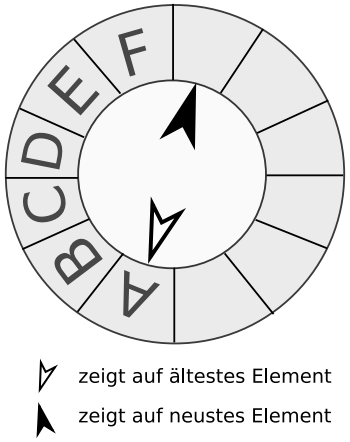


Abbildung 7: Prinzip einer FIFO-Warteschlange als Ring Buffer dargestellt [vgl. 4, S. 97]

Die Implementierung einer Warteschlange kann auf verschiedene Arten erfolgen, hier wird beispielhaft die Implementierung als Ring Buffer (auch Circular Buffer) aufgezeigt: Zur Initialisierung zeigen beide Zeiger bzw. Indizes auf das erste Feld des internen Arrays. Der Zähler wird auf Null gesetzt. Mit diesen Informationen gilt es zwei Fälle im Betrieb gesondert zu behandeln, nämlich dann, wenn beide Zeiger/Indizes auf das gleiche Feld zeigen:

- Ist der Zähler bei 0 (die Warteschlange also leer), darf kein weiteres Element mehr entnommen werden können.
- Ist der Zähler größer 0 (die Warteschlange somit vollständig befüllt) wird bei `Enqueue` das älteste Element im Falle eines Hinzufügens überschrieben.

In jedem anderen Fall von `Enqueue` werden alle in der Warteschlange befindlichen Elemente um ein Feld nach hinten verschoben (damit wird das älteste Element überschrieben) und das neuste Element wird vorne eingefügt.

`Dequeue` übergibt stets das älteste Element und entnimmt es gleichzeitig der Warteschlange, womit ein leeres Feld entsteht und der betreffende Zeiger bzw. Index so angepasst wird, dass er auf das zweitälteste Element zeigt.

Damit entsteht eine relativ einfach zu implementierende Datenstruktur für die Task-zu-Task-Kommunikation mit der verschiedene Auslieferungskriterien (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 Synchronisation sicherstellen und auch Daten verwalten muss. Die Daten müssen also in einem vordefinierten Zeitfenster beim Empfänger eingehen. Dies kann so gehandhabt werden, 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 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 kann mit folgenden beiden Befehlen implementiert werden: `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 nachbilden.

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 bereit gemeldet, wird der Sender suspendiert. Auch umgekehrt ist der Fall möglich, wie Abbildung 8b zeigt: Hier wird zuerst nach einer Nachricht gefragt und bis diese vorliegt, wird der entsprechende Task suspendiert. Besteht letztendlich eine Verbindung zweier Tasks über die Mailbox, wird die betreffende Nachricht ausgetauscht. Abbildung 8c zeigt

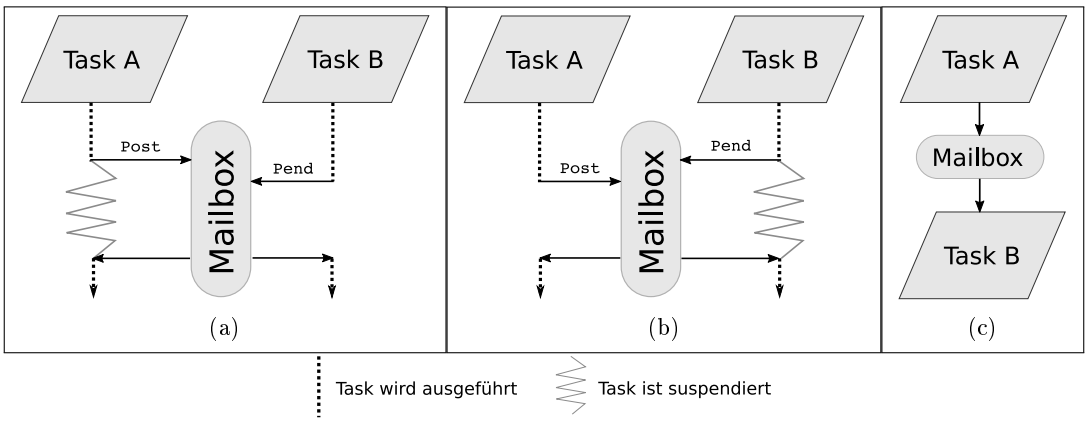


Abbildung 8: Datentransfer mit Task-Synchronisation [vgl. 4, S. 99]

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

Tabelle 3 zeigt die Teilergebnisse einer Messung, die auf einem Echtzeitbetriebssystem mit dem Publisher/Subscriber-Prinzip durchgeführt wurde: Alle fünf Sekunden beginnend ab 10s der Laufzeit des Systems wird - zum Vergleich - die aktuelle Systemzeit ausgegeben und jede neue Sekunde die aktualisierte Systemzeit in ein Topic (Mailbox) veröffentlicht. Der Empfänger erhält diese Zeitangabe (in Form von Nanosekunden seit Systemstart) und gibt die Differenz zur aktuellen Systemzeit aus, womit die Dauer sichtbar wird, wie lange die Daten vom Sender zum Empfänger unterwegs waren. Durchgeführt wurde diese Messung auf Basis einer virtuellen Maschine und dem Echtzeitbetriebssystem RODOS [2].

Diese Messung soll zugleich einen Eindruck vermitteln, in welcher zeitlichen Größenordnung die Datenübermittlung in etwa erfolgt. Es ist leicht erkennbar, dass sich die Zeitdifferenz, mit einer Ausnahme, im Nanosekundenbereich bewegt. Für viele Anwendungsbereiche dürfte dieser Jitter noch im vertretbaren Bereich liegen.

Mit den in der Abbildung 8 dargestellten Möglichkeiten, können so relativ leicht Daten synchron zwischen Tasks ausgetauscht werden. Entweder über eine Mailbox die auf Signalen basiert oder mittels des Publisher/Subscriber-Prinzips.

0:	10.000.003.014	ns ($t_{System} \approx 10s$)
1:	1.206	ns
2:	686	ns
3:	691	ns
4:	736	ns
5:	757	ns
6:	15.000.003.484	ns ($t_{System} \approx 15s$)
7:	762	ns
8:	724	ns
9:	769	ns
10:	690	ns
11:	633	ns
12:	20.000.002.990	ns ($t_{System} \approx 20s$)

Tabelle 3: Zeitliche Messung der Dauer eines synchronisierten Datentransfers

5 Zusammenfassung

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

So erreicht man Task-Koordination, die Steuerung eines Ablaufs, am besten mit Condition Flags, eine einseitige-Synchronisierung mit Event Flags und kann beiderseits 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 ist.

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

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

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

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