9 Embedded Betriebssystem

»Nichts ist stärker als eine Idee, deren Zeit gekommen ist!«

VICTOR HUGO

Aufbauend auf den Mikroprozessorgrundlagen in Teil I und dem Wissen über Einsatz, Arbeitsweise und Programmierung der Peripheriemodule in Teil II wird in diesem Teil die Integration als Anwendung in einem Embedded System betrachtet.

Eine moderne Applikation verwendet meist ein Echtzeitbetriebssystem zur Aufteilung in interagierende Tasks (siehe Abschnitt 9.3). Oft sind diese Anwendungen in ein großes System eingebunden und mit der »Cloud« im Sinne des »Internet of Things« (IoT) vernetzt (siehe Kapitel 10).

9.1 Embedded Applikationsmodell

In herkömmlichen Embedded Systemen ohne Betriebssystem wird nach dem Reset und einer Basisinitialisierung direkt die `main()`-Funktion angesprungen. In einer »Endlos«-Schleife werden die Abarbeitungsfunktionen der einzelnen Applikationsmodule nacheinander wiederholt aufgerufen (»gepollt«). Dadurch bekommt jedes Modul Rechenzeit und kann mit den jeweiligen externen Komponenten kommunizieren. Der Programmfluss ist linear, allerdings ist die Dauer eines Schleifendurchlaufs von den jeweiligen Modulen abhängig und variiert ständig. Die Applikation ist monolithisch und schwer strukturierbar.

In Abb. 9–1 ist die Pulsoximeterapplikation in dieser Architektur dargestellt. In jedem Schleifendurchgang werden die Sensordaten eingelesen, der Zustand des Tasters bestimmt, die Daten verarbeitet und der Puls berechnet und anschließend auf dem Display angezeigt. Ein kurzes Schlafen des Systems schließt den Durchgang ab.

Die einzelnen Module werden auch dann gepolt, wenn sie keine Tätigkeit zu verrichten haben, beispielsweise wenn keine Daten vorliegen. Ist das Schleifenintervall zu lange, werden die Eingabedaten, die für die Applikation zugänglich gespeichert werden, hingegen nicht rechtzeitig ausgelesen und gehen im schlimmsten Fall verloren. Dies macht sich besonders bei Codeänderungen negativ bemerkbar: jede Änderung in einem Modul ändert die Schleifendauer und damit die Pollrate jedes einzelnen Moduls.

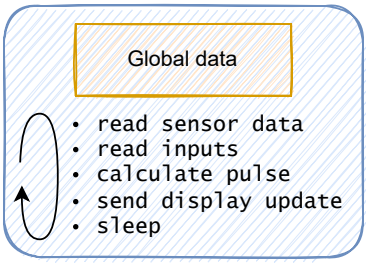

Abb. 9–1
Herkömmliche Hauptschleife ohne Betriebssystem

Bereits in Abschnitt 6.1 wurde der Latenz-Nachteil des Pollings erläutert. Eine Erhöhung der Poll-Rate bewirkt ein Steigen der CPU-Last. Interrupt-Systeme reduzieren Latenzzeit und CPU-Last. Abb. 9–2 zeigt die Abhandlung von Sensordaten und Eingaben per Interrupt. Da ISRs möglichst kurze Laufzeit haben sollen, ist es schwer möglich, das gesamte Applikationsverhalten inklusive Kommunikation wie beispielsweise das Senden von Updates an das Display über I^2C direkt in der ISR zu implementieren.

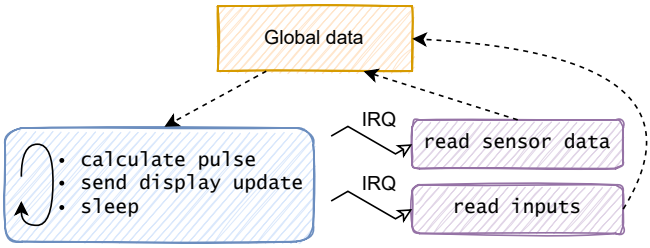

Abb. 9–2
Ereignisgesteuerte Applikation

Deshalb befinden sich die Kontrolle der Applikation, die Berechnung des Pulses und die Kommunikation weiterhin in der Hauptschleife. Um die Hauptschleife vom Interrupt-Kontext auf einfache Weise zu entkoppeln, werden die entsprechenden Daten global gespeichert. Der gemeinsame, mitunter »gleichzeitige« (nebenläufige) Zugriff aus ISRs und Hauptschleife kann schwere Probleme hervorrufen. Wird die Hauptschleife beispielsweise während der Abarbeitung der Daten durch einen Interrupt, der eben diese Daten ver-

ändert, unterbrochen, wird die unterbrochene Abarbeitung mit den neuen Daten, die zu den alten nicht mehr konsistent sind, fortgesetzt.

9.2 Multitasking

Ein »Multitasking«-Betriebssystem erlaubt eine strukturierte Herangehensweise, die auch Mechanismen zur Vermeidung von Problemen der Nebenläufigkeit bereitstellt. Abb. 9–3 zeigt einen geregelten Programmablauf, bei dem die Applikation in separate Tasks (bzw. Threads) unterteilt wird, die jeweils eigene Aufgaben abarbeiten. Jeder Task läuft eigenständig parallel zu den anderen Tasks, mit jeweils eigener Hauptschleife und eigenem Timing.

Für den Datenaustausch zwischen den Tasks stellt das Betriebssystem eigene Kommunikationsmittel zur Verfügung. Im Beispiel liefern zwei Tasks Daten vom Pulssensor und vom Taster an den Haupttask, der diese Daten sammelt, verarbeitet und das Ergebnis an den Display-Task zur Anzeige sendet. Das Beispiel macht deutlich, dass der Aufwand für die Kommunikation bei Einsatz eines Betriebssystems steigt. Wegen der klaren Strukturierung mit den Effekten der Fehlervermeidung und verbesserten Wartbarkeit werden auch moderne Kleinstsysteme, die über die notwendigen Ressourcen verfügen, zunehmend mit einem Betriebssystem aufgesetzt.

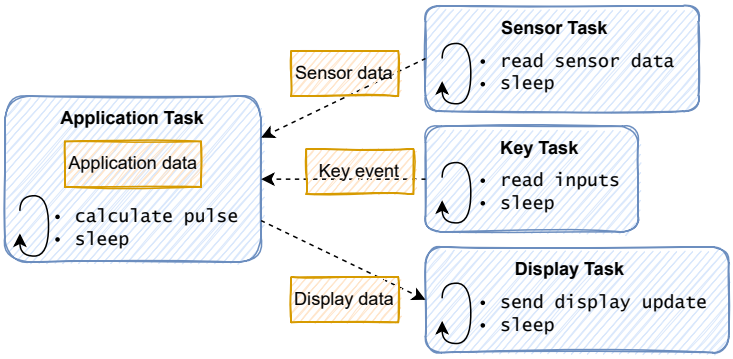

Abb. 9–3
Programmaufbau mit kommunizierenden Tasks

Da auf einer CPU mit nur einem Kern die parallelen Tasks nicht tatsächlich gleichzeitig ablaufen können, wechselt der »Scheduler« im Kern (»Kernel«) des Betriebssystems die einzelnen Tasks beständig ab. Beim kooperativen Multitasking arbeitet ein Task eine Teilaufgabe ab und teilt dann dem Scheduler per »Yield« mit, dass dieser zum nächsten Task wechseln kann. Beim präemptiven Multitasking unterbricht der Scheduler, typischerweise auf einer Zeitbasis, den gerade arbeitenden Task, um dann zum nächsten Task zu wechseln.

Der Scheduler bestimmt den nächsten Task mithilfe einer definierten Scheduling-Strategie. Eine einfache und häufige Strategie ist der zyklische Wechsel zum nächsten bereiten Task (»Round Robin«), oft unter Berücksichtigung einer Task-Priorität.

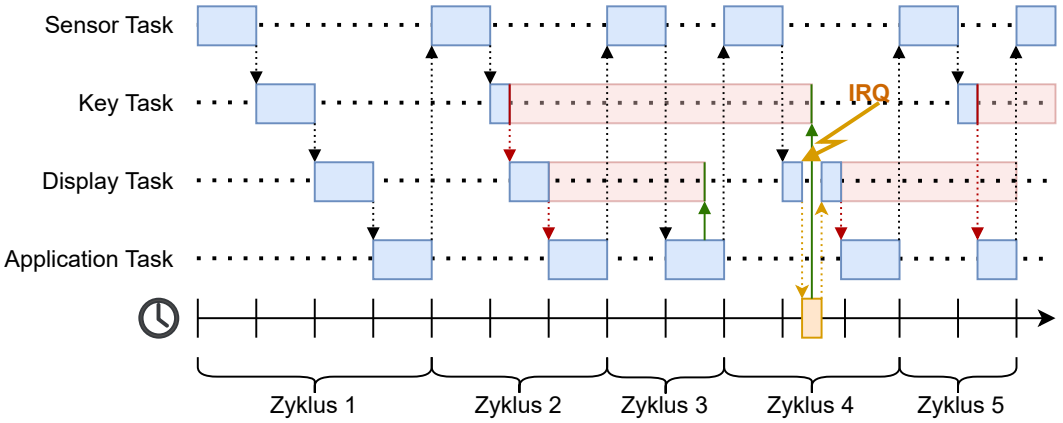

Abb. 9–4 *Arbeitsweise des Schedulers mit Round-Robin-Strategie*

In Abb. 9–4 ist das Scheduling ohne Task-Prioritäten dargestellt (für prioriätenbasiertes Scheduling siehe Abschnitt 9.4.7). Zyklus 1 zeigt ein Durchlaufen aller Tasks mit der Round-Robin-Strategie. Jeder Task dieses präemptiven Systems wird für die Dauer einer Zeitscheibe ausgeführt, um dann unterbrochen zu werden. Der aktuelle Kontext des Tasks, im einfachsten Fall bestehend aus allen Registern, wird im »Task Control Block« (TCB) gesichert (siehe Abschnitt 9.3.1). Für den Wechsel zum nächsten Task wird der Kontext aus dem entsprechenden TCB geladen und der Task exakt dort fortgesetzt, wo er zuletzt unterbrochen wurde. Bei einer Dauer von 10 ms für jeden Zeitschlitz und vier Tasks erhält jeder Task mindestens 25 Mal pro Sekunde die CPU.

Der Kernel des Betriebssystems führt den Taskwechsel mit Hilfe eines periodischen Timers im Interrupt-Kontext durch (in der Abbildung mit gestrichelten Pfeilen dargestellt). Je weniger Daten im TCB zu sichern sind, desto schneller kann der Wechsel erfolgen. Prozesse unterscheiden sich von Tasks bzw. Threads dadurch, dass sie virtuellen Speicher und einen Speicherschutz gegenüber anderen Prozessen haben. Die Einstellungen der dafür notwendigen Memory Management Unit (MMU) müssen beim Prozesswechsel ebenso gewechselt werden, was einen Mehraufwand gegenüber einem Taskwechsel bedeutet. Aus diesem Grund spricht man bei einem Task von einem »leichtgewichtigen« Prozess. MMUs sind in PC-Systemen zu finden, und deren Betriebssysteme wie Windows, macOS und Linux

bieten Prozesse und Threads zur Aufteilung von Prozessen an. Embedded Kleinsysteme bieten oft nur ein Speicherschutzmodul wie die RISC-V Physical Memory Protection (PMP). Mit diesem Modul kann der Zugriff auf Speicherbereiche definiert werden, der Aufbau eines virtuellen Speichermodells ist aber nicht möglich (siehe auch Abschnitt 9.5).

Wenn Tasks auf externe Ereignisse warten, können sie den Zustand »Blocked« annehmen. Blockierte Tasks (rot eingezeichnet) werden vom Scheduler nicht mehr ausgeführt, bis die Blockade behoben ist. In Zyklus 2 blockieren der Key-Task, um auf einen Interrupt zu warten, und der Display-Task, da er keine neuen Daten anzuzeigen hat. Beim Eintritt in die Blockade erfolgt ein Wechsel (»Yield«) zum nächsten Task noch während der aktiven Zeitscheibe (roter Pfeil).

In Zyklus 3 generiert der Application-Task Daten, die auf dem Display angezeigt werden sollen. Die Blockierung des Display-Tasks wird aufgehoben (grüner Pfeil), und der Task wird »Ready«, fängt aber nicht unmittelbar an zu laufen. Im Beispiel arbeitet der Application-Task erst seinen Zeitschlitz ab, worauf der Scheduler in Zyklus 4 den wartenden Sensor-Task ausführt. Beim darauf folgenden Taskwechsel wird an den Display-Task übergeben. Dieser sendet die Anzeigedaten an das Display und wartet blockierend auf die nächsten Daten.

In Zyklus 4 wird während der Abarbeitung des Display-Tasks ein Interrupt durch Drücken des Tasters ausgelöst. In die ISR wird sofort verzweigt, um den Interrupt schnellstmöglich abzuhandeln. Dort wird die Blockierung des Key-Tasks aufgehoben und zum unterbrochenen Display-Task zurückgekehrt. In Zyklus 5 erfolgt dann die Abarbeitung des Tastendrucks durch den laufenden Key-Task.

Für das blockierende Warten und die Aufhebung der Blockierung stellt das Betriebssystem als grundlegende Datenstruktur den Semaphor (siehe Abschnitt 9.4.1) zur Verfügung. Auf diesem Synchronisierungsmechanismus wird die Kommunikation zwischen Tasks mit Queues, Mutexes, Mailboxes usw. vom Betriebssystem bereitgestellt.

9.3 Echtzeitbetriebssystem

Ein Betriebssystem übernimmt die Verwaltung der Ressourcen eines Rechners und stellt diese den Anwendungsprogrammen zur Verfügung. Es stellt die Serviceschicht zwischen Hardware und Applikation dar (siehe Abschnitt 6.2.1) und stellt neben der Programmierschnittstelle auch Serviceprogramme zur Verfügung. Es wird diskutiert, ob Entwicklungstools wie Compiler und Editor auch Teil des Be-