Realtime Multitasking für Mikrocontroller

mit der Bibliothek mtRTOS2407 für Arduino

Dieter Holzhäuser www.wieundwarum.de 09.2024

Mit diesem Beitrag möchte ich den Freunden der Mikrocontroller-Programmierung ein einfaches Multitasking vorstellen, dessen Kernel ohne jeden Assembler-Code in C geschrieben ist, und das ich seit mehr als 10 Jahren verwende.

Vor etwa drei Jahren bin ich auf die Arduino IDE umgestiegen und benutze insbesondere das Board Arduino Nano, aber auch ähnliche selbst entwickelte Boards und neuerdings den ATtiny85.

Wie man die Arduino IDE auf dem PC installiert, Programme schreibt und wie man mit Nano umgeht, wird für diesen Beitrag vorausgesetzt. Auch etwas Englisch sollten meine Leser verstehen, denn ich schreibe Programme seit kurzem vollständig in Englisch, wenn auch noch unbeholfen und holprig. Zum einen werden solche Programme überall auf der Welt verstanden und zum andern lassen sich die Dinge kurz und prägnant ausdrücken. Nicht selten habe ich dadurch auch Fehler entdeckt.

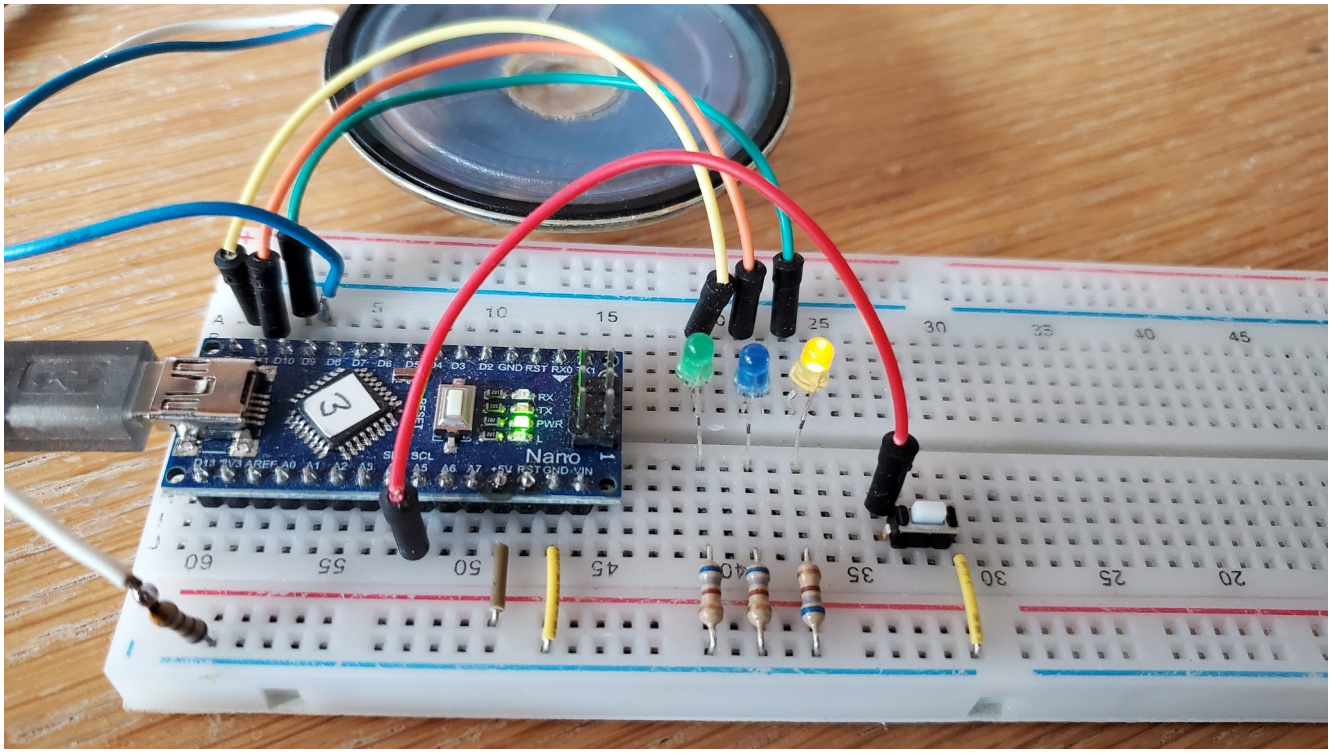
Inhaltsverzeichnis

1. Hard- und Software

Alles was gebraucht wird, um einzusteigen, ist die Bibliothek mtRTOS2407 und eine ganz einfache Hardware. Die Datei mtRTOS2407.zip ist von der oben angegebenen Website herunterzuladen. Als zip-Bibliothek wird mtRTOS2407 (kurz mtRTOS) in die Arduino-IDE aufgenommen und in Sketche eingebunden. Die zip-Datei enthält auch Beispielsketche, die unter Daten/Beispiele zu finden sind und die eine unverzichtbare Ergänzung dieses Textes darstellen.

Der Speicherbedarf eines einfachen mtRTOS-Sketches ist etwa 3 KB, wie die Beispiele *helloworld* und *base* zeigen.

Die Hardware besteht aus dem Board Arduino Nano, an das drei LED und ein Taster angeschlossen sind, was auf dem folgenden Bild zu sehen ist:



Der Kleinlautsprecher ist mit einem Vorwiderstand von 680 Ohm an Pin 9 angeschlossen. Falls man auf die Musik, die der Beispielsketch *fun* abspielen kann, keinen Wert legt, tut es auch eine flackernde LED.

Das Board Arduino Nano wird in diesem Text auch als Mikrocontroller oder kurz als MC bezeichnet.

2. Was ist Multitasking?

Ein erster Eindruck entsteht, wenn der Beispielsketch *base* hochgeladen wird. Drei LED blinken mit unterschiedlichen Frequenzen. Mit dem Taster kann das Blinken einer der LED angehalten und freigegeben werden.

Diesen Sketch herkömmlich, also mit der Hauptschleife *loop* zu programmieren, macht schon Mühe. Wenn der Sketch auch noch etwas anderes tun soll, ist die Sache ziemlich aussichtslos.

Das liegt daran, dass die Hauptschleife kaum geeignet ist, unterschiedliche Dinge zu tun, insbesondere dann nicht, wenn Warteschleifen erforderlich sind.

Ein Blick auf den Code von *base* zeigt, auch ohne Erklärung, wie einfach und strukturiert das Problem mit der Bibliothek mtRTOS zu lösen ist.

Ein Multitasking Programm besteht aus Tasks (ich verwende den Artikel „die“). Nach außen hin laufen Tasks wie selbstständige Programme. Das hat Vorteile:

- Komplexe Probleme sind leichter zu programmieren, wenn sie in Teilaufgaben (Tasks) zerlegt werden. Die wirklich notwendigen Abhängigkeiten oder Beziehungen gestaltet der Programmierer.
  Bei einer Heizungssteuerung zum Beispiel, könnte es je eine Task für den Kessel, die Heizkreise, die Warmwasserbereitung und auch eine für die Kommunikation mit dem Anwender geben. Die Task für den Kessel würde sich nur darum kümmern, die Forderung anderer Tasks nach einer bestimmten Temperatur des Heizungswassers zu erfüllen.
- Viele einfach erscheinende Probleme sind nur mit Multitasking *auch einfach* lösbar, wie das Beispiel *base* zeigt.
- Beim Umbau von Multitasking Programmen werden lediglich Tasks hinzugefügt oder entfernt.

Die Beispiele der Bibliothek sind natürlich nicht für eine so umfangreiche Peripherie wie eine Heizung geschrieben. Eine Heizungssteuerung wäre für mtRTOS kein Problem, wohl aber für den interessierten Programmierer, der eine komplexe Peripherie nicht mal eben aufbauen kann. Deshalb kommen die Beispielprogramme mit minimaler Hardware aus, was auch bedeutet, dass es kein großes Gesamtproblem gibt, das in Tasks zu zerlegen wäre. Der Parallellauf kann genauso gut -vielleicht noch besser- gezeigt werden, wenn die Tasks eines Programms kaum oder gar nicht zusammenarbeiten, wie das bei den Beispielsketchen der Fall ist.

Tasks können auf Mikrocontrollern nicht wirklich parallel laufen. Es genügt aber, wenn so getan wird, als ob. Das bekannte preemptive Multitasking, bei dem ein Scheduler bestimmte Zeitfenster vergibt, in denen eine Task laufen kann oder darf, kommt wegen des hohen Verwaltungsaufwands und besonderer Techniken für ein einfaches Multitasking nicht in Frage. Deshalb arbeitet mtRTOS *ereignisorientiert*.

3. mtRTOS im Überblick

Der Name setzt sich aus Multitasking (mt), Real Time (RT) und Betriebssystem (OS) zusammen.

Die Komponente *mt* der Bibliothek ist der Multitasking Kernel. Neben der Verwaltung der Tasks, enthält er auch die Hauptschleife des Systems, die deshalb nicht mehr Sache des Programmierers ist. Die im Kernel integrierte Hauptschleife macht die Bibliothek mtRTOS2407 zu einem einfachen Betriebssystem (OS).

Das Multitasking von mtRTOS ist ereignisorientiert. Es ist deshalb sehr gut geeignet, technologische Prozesse zu begleiten (zu steuern). Einerseits erzeugen solche Prozesse Ereignisse in Form von Zustandsänderungen an Input Pins, und andererseits müssen sie dem Ereignis *Zeitablauf* folgen, das vom Mikrocontroller generiert wird. Dazu kommen Ereignisse, die anderer Natur sind, und vom Programm erzeugt werden, siehe 5.3.

Der Programmlauf besteht prinzipiell darin, auf ein Ereignis zu warten, es mit Programmcode zu verarbeiten, wiederum auf ein Ereignis zu warten usw. Die Wartezeit kann ausgeprägt sein und durchaus auch gegen Null gehen.

Es liegt auf der Hand, dass das Programm in der Lage sein muss, unter allen Umständen den Ereignissen eines technologischen Prozesses zu folgen und Zeiten exakt einzuhalten, was als „echtzeitfähig“ bezeichnet wird. Es ist nicht richtig, einem Rechner diese Eigenschaft zuzuschreiben, nur weil er besonders leistungsfähig ist. Der technologische Prozess muss in die Betrachtung einbezogen werden. Mögliche Echtzeitfähigkeit kann jedoch durch ungeeignete Programmierung verloren gehen. Multitasking unterstützt Echtzeitfähigkeit, ist aber keine Voraussetzung dafür.

Das führt zu der Frage, für welche technologischen Prozesse mtRTOS *nicht* geeignet ist. Das sind solche, die generell Antwortzeiten von weniger als 1 ms verlangen. Darunter ist die Zeit zu verstehen vom Erscheinen eines Signals an einem Input Pin bis zur Reaktion des Programms. Kürzere Antwortzeiten sind mit Interrupts zu erzielen, vorausgesetzt der Input Pin ist dafür geeignet.

Funktionen, die länger als etwa 900 µs laufen, gefährden nicht nur die Antwortzeit von mt, sondern verlängern auch die programmierten Zeitabläufe. Deshalb ist im Einzelfall zu prüfen, ob diese Einschränkungen toleriert werden können. Außerhalb von Bibliotheken bietet mtRTOS Möglichkeiten, solche Probleme elegant zu lösen, siehe 5.4.

In diesem Zusammenhang sollte man wissen, dass bei Uploads das im ROM vorhandene Programm zunächst gestartet wird, um dann durch den eigentlichen Upload abgebrochen zu werden. Das ist besonders auffällig bei so kurzen und einfachen Programmen wie *helloworld*.

Die Bibliothek mtRTOS hat zwei weitere Komponenten, und zwar den seriellen User Input (sui, siehe 6.2) und den EEPROM-Zugriff (ee, siehe 7).

4. mtRTOS Tasks

Sie bestehen fast immer aus *mehreren* Taskfunktionen, auch wenn die Aufgabe bei herkömmlicher Programmierung leicht mit *einer* Funktion zu lösen wäre. Das liegt daran, dass Taskwechsel sehr einfach zu realisieren sind, wenn auf jedes Ereignis mit einer dazu gehörenden (Task)funktion reagiert wird. (Die Aufgaben von Task HELLO und Task BLRED in den Beispielen sind jedoch so einfach, dass *eine* Taskfunktion genügt.)

4.1. Taskfunktionen

Taskfunktionen haben keine Argumente und keine Wertrückgabe.

Die folgende Task BLGREEN stammt aus dem Beispielsketch *base* und hat die Taskfunktionen *blgreen1* und *blgreen2*.

```
//////////////////Task BLGREEN    demo of asymmetrical flashing
void blgreen1 () {
  WPIN(GREEN, HIGH);
  mtdelay (100, blgreen2);
}
void blgreen2 () {
  WPIN(GREEN, LOW) ;
  mtdelay (700, blgreen1);
}
```

In der Benennung von Taskfunktionen sollte zum Ausdruck kommen, dass sie funktional zusammen gehören. Deshalb werden Taskfunktionen zweckmäßig mit dem klein geschriebenen Tasknamen und angehängter Nummer benannt. Davon kann abgewichen werden, wenn eine Gruppe von Taskfunktionen einen speziellen Zweck erfüllt. Bei umfangreichen Tasks sollte die Taskfunktion mit der angehängten Nummer 1 die Startfunktion sein und die notwendigen Initialisierungen vornehmen.

Für den Fortgang von Tasks ist in jeder Taskfunktion der Aufruf einer mt-Funktion unverzichtbar. mt-Funktionen bereiten den Aufruf der nächsten Taskfunktion in Verbindung mit einem Ereignis vor. Im Beispiel oben heißt die mt-Funktion *mtdelay* , und das Ereignis ist ein Zeitablauf. Näheres dazu siehe 5.1 .

Das Aufrufen von Taskfunktionen ist Sache der Hauptschleife von mt in Gestalt der endlos laufenden Funktion *runner*. Daher kehren Taskfunktionen dorthin zurück. Entscheidend für den Aufruf einer Taskfunktion ist das *nächste* Ereignis. Weil Ereignisse auch nahezu gleichzeitig auftreten können, werden die Tasks, die dadurch laufbereit werden, von mt in einer Warteschlange verwaltet.

4.2. mt-Schleifen

Task BLGREEN realisiert asymmetrisches Blinken. Sie läuft endlos, weil Taskfunktion *blgreen2* durch einfache Namensnennung mit *blgreen1* fortgesetzt werden kann. Insofern ist Task BLGREEN eine endlose mt-Schleife, was für die meisten Tasks zutrifft.

Nichts spricht gegen endliche Schleifen in einer Task. Wenn die Abbruchbedingung erfüllt ist, unterbleibt der Schleifenaufruf, und die Task wird regulär fortgesetzt. mt-Schleifen erfordern meistens eine Taskfunktion zur Initialisierung, insbesondere zum Setzen eines Schleifenzählers. Die mt-Schleife in Task PLAY von Sketch *fun* spielt Noten ab, und Task PRIME errechnet Primzahlen mit Hilfe geschachtelter Schleifen.

Weil die Namensnennung in mt-Funktionen genügt, um mt-Schleifen entstehen zu lassen, wird es dem Programmierer kaum bewusst, eine Schleife programmiert zu haben.

4.3. Start und Stopp von Tasks

Tasks werden durch die Funktion *start* gestartet. In den meisten Fällen geschieht das im Arduino-Setup. Die Startfunktion hat als Argument die Taskfunktion, die der *runner* als erste aufrufen soll.

Wenn in einer Taskfunktion keine mt-Funktion aufgerufen wird, endet die Task, was durchaus gewollt sein kann, wie bei den Tasks HELLO von Beispielsketch *helloworld* und Task ONBLUE von Beispielsketch *semaphore* . Eine Task endet auch, wenn die Funktion *stop* aufgerufen wird, was an beliebiger Stelle möglich ist. Eine beendete Task kann jederzeit neu gestartet werden.

4.4. Tasklose Funktionen

Die üblichen Funktionen sind tasklos, weil sie keine mt-Funktionen aufrufen. Tasklose Funktionen können in jeder Taskfunktion aufgerufen werden. Während sie laufen, gehören sie zur betreffenden Task.

4.5. Subtasks

Wenn an mehreren Stellen im Programm dasselbe abgrenzte Problem auftritt und es mit Taskfunktionen zu lösen ist, wird diese Gruppe von Taskfunktionen zweckmäßig als

Subtask behandelt. Die Subtask kann wie ein herkömmliches Unterprogramm (Subroutine) mit ihrer Startfunktion aufgerufen werden.

Die Startfunktion hat mindestens ein Argument, das die Taskfunktion enthält, mit der die Task fortzusetzen ist, wenn die Subtask endet, siehe auch 5. Die Argumente müssen zur Verwendung gespeichert werden. Nicht ganz alltäglich ist die Definition der Zeigervariablen xxxxx, in der die Fortsetzungsfunktion gespeichert wird:

```
void (*xxxxx)(void);      //save task function to continue task
```

Subtasks dürfen nur aufgerufen werden, wenn ein Wiedereintritt ausgeschlossen ist. (Sie sind nicht reentrant.) Andernfalls ist eine Ausschluss-Semaphore zu verwenden, siehe 5.3.

Subtasks sind selten und kommen in den Beispielsketchen nicht vor. In der Bibliothek mtRTOS gibt es jedoch die Subtask *eewrite*, deren Taskfunktion *ee1*, eine Gruppe von Bytes in den EEPROM schreibt, siehe 7 .

4.6. Zustände von Tasks

Eine Task kann sich in einem von vier Zuständen befinden:

- Wenn die Task gestoppt wurde bzw. noch nicht gestartet ist, hat sie den Zustand TERMINATED
- Solange eine Taskfunktion läuft, ist die zugehörige Task im Zustand RUNNING
- Eine Task, die laufen könnte, ist im Zustand READY (laufbereit).
- Wenn die Task auf ein Ereignis wartet, ist sie im Zustand WAITING. Es gibt 5 Varianten dieses Zustands, je nachdem auf welches Ereignis (welche Ereignisse) gewartet wird.

Zusammenstellung

```
Zustand         Kennzahl

READY            0
TERMINATED       5
RUNNING          6
WAITING_S       15     //Semaphore (Sema)
WAITING_T       23     //Timeout
WAITING_B       39     //Input pin edge
WAITING_TS      31     //Both Timeout or Sema
WAITING_TB      55     //Both Timeout or Input pin edge
```

5. Die mt-Funktionen

Weil es genügt, vier Ereignisse zu unterscheiden, gibt es vier mt-Funktionen:

```
   mt-Funktion  Ereignis        Erzeuger des Ereignisses

   mtdelay      timeout event   Timer2 ISR (Interrupt Service Routine), Zeitablauf
   mtpin        pin event       Timer2 ISR, Zustandsänderung eines Hardware Pins
   mtsema       semaphore event Programm, und zwar durch Funktion signal
   mtcoop       immediate event Programm, und zwar durch mt-Funktion mtcoop selbst
```

Gemeinsam ist allen mt-Funktionen das Argument mit dem Zeiger der Taskfunktion, mit der die Task fortgesetzt werden soll, wenn das jeweilige Ereignis eintrifft.

> Ein Funktionsname als Argument ist eher ungewöhnlich. Tatsächlich handelt es sich um eine Zahl, und zwar um die Adresse (Zeiger) des ersten Bytes der Funktion im Speicher.

mt-Funktionen können auch als Auftrag an das Betriebssystem verstanden werden, das zugeordnete Ereignis zu erkennen, um dann die spezifizierte Taskfunktion aufzurufen.

5.1. mtdelay

Die Timer2 ISR läuft zu jeder Millisekunde. Sie zählt die Zeitzähler der Tasks herunter, die auf einen Zeitablauf warten (timeout event). Das sind die Tasks, die die mt-Funktion *mtdelay* aufgerufen haben. Erreicht der jeweilige Zeitzähler Null, wird die Task laufbereit (READY).

```
mtdelay (700, blgreen1);
```

5.2. mtpin

Timer2 ISR prüft außerdem zu jeder Millisekunde die Zustandsänderungen der-Pins, die in den Aufrufen der mt-Funktion *mtpin* als Argument angegeben wurden. Tritt die spezifizierte Zustandsänderung ein (pin event), wird die jeweilige Task laufbereit (READY).

Mit dem folgenden Aufruf von *mtpin* wartet Task PUSHBUTTON aus Beispielsketch *base* darauf, dass der Taster, der am Input Pin BUTTON angeschlossen ist, gedrückt wird. Er verursacht dabei eine Zustandsänderung von HIGH nach LOW, was im zweiten Argument mit LOW zum Ausdruck kommt.

```
mtpin (BUTTON, LOW, pushbutton3);
```

5.3. mtsema

Nicht nur timeout events und pin events sind relevante Ereignisse, sondern auch der Abschluss einer Berechnung oder eines Vorgangs, das Überschreiten eines Maximalwertes, das Erscheinen eines Interrupts und andere.

Solche Ereignisse werden durch Aufruf der Funktion *signal* zu Semaphore-Ereignissen (semaphore event). Selbst timeout events und pin events können zu semaphore events umgewidmet werden. Für jedes semaphore event wird eine Semaphore definiert, wie im Beispielsketch *semaphore* , der die Anwendung von Semaphoren demonstriert.

```
//////////////Semaphore Definitions ( 0 to 17 )
#define EXCLSEMA 0
#define SIGNALSEMA 1
#define PAUSESEMA 2
```

Der Tastendruck auf BUTTON wird in Task PUSHBUTTON mit *signal* in zwei Semaphore-Ereignisse umgewidmet. Dadurch nimmt die Semaphore SIGNALSEMA Einfluss auf Task ONBLUE und die Semaphore PAUSESEMA auf Task BLYELLOW (simple intertask communication).

SIGNALSEMA löst mit Task ONBLUE einen Lichtimpuls aus.

Dagegen realisiert Task BLYELLOW einen Blinkzyklus. PAUSESEMA stoppt diesen Zyklus oder gibt ihn frei. Bei jedem Durchgang wird PAUSESEMA mit *mtsema* geprüft. Weil die Semaphore dadurch zurückgesetzt wird, wenn sie gesetzt ist, wird sie direkt danach mit *signal* restauriert, wodurch der Zyklus in Gang bleibt.

```
void blyellow1 () {
  mtsema (PAUSESEMA, blyellow2);
}
void blyellow2 () {
  signal (PAUSESEMA);   // set self continuing !immediate!
  WPIN(YELLOW, !RPIN(YELLOW) ) ;
  mtdelay (1000, blyellow1);
}
```

Mit die wichtigste Anwendung von Semaphoren ist der gegenseitige Ausschluss (mutual exclusion). Task SHORTGREEN will LED GREEN für einen kurzen Lichtimpuls nutzen und Task LONGGREEN für einen langen. Beide Tasks arbeiten zyklisch, aber mit zufälligen Zykluszeiten. Mit der Ausschluss-Semaphore EXCLSEMA werden die Zugriffe in eine Reihenfolge gebracht. EXCLSEMA wird wie ein Staffelstab verwendet. Die Task, die ihn besitzt, erzeugt ihren Impuls und gibt den Staffelstab weiter. Wenn die Task den nächsten Impuls erzeugen will, muss sie warten bis sie den Staffelstab wieder bekommt. Als Extra geben die beiden Tasks jeweils ihre Tasknummer aus, wenn sie „dran“ sind.

Der gegenseitige Ausschluss kommt immer zum Tragen, wenn zwei oder mehr Tasks sich eine knappe Ressource teilen müssen, im Beispiel ist das LED GREEN. Daher kann es sein, dass mehr als eine Task auf dieselbe Semaphore warten. mtRTOS sieht für diese Tasks eine Warteschlange vor.

Ganz wichtig ist, dass im Arduino-Setup oder einer initialisierenden Taskfunktion die Ausschluss-Semaphore gesetzt wird. Nur dann kommt die *erste* Zuteilung der knappen Ressource zu Stande.

5.4. mtcoop

Das mit der mt-Funktion *mtcoop* verknüpfte Ereignis ist ihr Aufruf selbst, was als Sofort-Ereignis (immediate event) anzusehen ist. Das heißt, *mtcoop* bringt den *runner* dazu, unmittelbar die als Argument übergebene Taskfunktion aufzurufen. *mtcoop* ermöglicht den Übergang von einer Taskfunktion auf eine andere ohne ein reales Ereigniss. Das kann bei der Programmierung von Tasks erforderlich sein, dient aber insbesondere zur Kooperation.

Auf Kooperation zu achten, ist Sache des Programmierers. Wenn sich bei der Programmierung abzeichnet, dass eine Taskfunktion länger als 500 µs läuft, sollte das Problem unter Verwendung von *mtcoop* auf zwei oder mehr Taskfunktionen verteilt werden.

Im Extremfall gibt es in einer Task keine Ereignisse, das heißt, sie besteht nur aus einer langlaufenden mt-Schleife unter Verwendung von *mtcoop*. Eine solche Task begleitet keinen technologischen Prozess, sondern führt eine langwierige Berechnung aus. Das ist bei Task PRIME aus dem Beispielsketch *fun* der Fall, die Primzahlen berechnet. Kooperation heißt, die Kontrolle freiwillig und rechtzeitig an den *runner* zurückzugeben, damit andere Tasks zum Zug kommen.

Bei den meisten Programmen für technologische Prozesse ist die Frequenz von Ereignissen im Mittel gering, und der Code von Taskfunktionen läuft nur für sehr kurze

Zeit. Der *runner* hat daher relativ lange Phasen, in denen er Leerlaufzyklen macht. Das Programm insgesamt „hangelt“ sich über Leerlaufphasen von Ereignis zu Ereignis.

Daraus folgt, dass Tasks für umfangreiche Berechnungen, die der Programmierer kooperativ gestaltet hat, dem *runner* keine Gelegenheit für Leerlauf geben. Die Bibliotheksfunktion *getidle* gibt in diesem Fall 0 zurück. Das Echtzeitverhalten der anderen Tasks wird dadurch in keiner Weise beeinträchtigt. Man könnte auch sagen, die Fähigkeit des Mikrocontrollers, sinnvolle Dinge zu tun, wird maximal genutzt.

5.5. Timeout

Es ist eine besonders nützliche Eigenschaft von mtRTOS, dass in einer Taskfunktion zwei mt-Funktionen aufgerufen werden können, wenn die eine *mtdelay* ist. Das heißt, die andere mt-Funktion ist *mtsema* oder *mtpin* (*mtcoop* wäre auch möglich, ergibt aber keinen Sinn). Das Ereignis, das zuerst erscheint, hat „gewonnen“, das heißt, die jeweils zugeordnete Taskfunktion wird ausgeführt und die andere nicht. Auf diese Weise sind ohne Programmieraufwand Timeout-Lösungen möglich, ohne die kein Multitasking auskommt. Task REAC im Beispielsketch *fun* enthält einige Timeouts.

In der folgenden Taskfunktion aus Task REAC wird 2 s auf einen Tastendruck gewartet. Erfolgt er rechtzeitig, geht es mit *reac5* weiter, andernfalls mit *reac6*. Das heißt, das Warten auf einen Tastendruck wird durch den Timeout abgebrochen.

```
void reac4 () {                 //start of measurement
  WPIN(REACLED,HIGH );          //now user must react as quickly as possible
  tbegin = getclock();
  mtdelay ( 2000, reac6);       //button not pressed within 2 s, process aborted
  mtpin ( BUTTON, LOW, reac5); //waiting for push
}
```

Es geht auch umgekehrt. In der folgenden Taskfunktion hält der Timeout einen Zyklus in Gang, der durch einen Tastendruck abgebrochen wird.

```
void reac2 () {                  //flashing LED by an embedded 100 ms cycle
  WPIN(REACLED, !RPIN (REACLED) );
  mtdelay (100, reac2);
  mtpin (BUTTON, LOW, reac3);   //waitung for push
}
```

6. Serielle User Kommunikation

Eine ereignisorientierte Umgebung erfordert, dass auch ein serieller User Input ein Ereignis ist. Ob das mit Methoden der Klasse Serial von Arduino möglich ist, bleibt offen. Unabhängig davon stellt die Bibliothek mtRTOS dafür ihre eigene Lösung bereit, und zwar in Form von Task SUI (serial user input). Insbesondere ermöglicht Task SUI die Eingabe von mehr als einer Zahl je Input und andere Spezialitäten. Die Eingabe von Text ist nicht vorgesehen, weil mtRTOS-Programme damit kaum etwas anfangen können.

Task SUI detektiert serielle User Inputs und macht eine Vorauswertung, um dann das Ergebnis der Anwendung als Semaphore-Ereignis zur Verfügung zu stellen. Das bedeutet, der User ergreift die Initiative. Er kann jederzeit etwas eingeben, worauf das Programm reagieren muss. (Grafische Benutzeroberflächen arbeiten genauso.)

Der serielle Output, zum Beispiel durch Serial.print (als PR definiert), ist überall in einem mtRTOS-Programm möglich. Er ist aber nur sinnvoll als Antwort auf User Inputs und andere Aktionen des Users sowie temporär zur Fehlersuche. Zu beachten ist, dass unter

mtRTOS die auszugebenden Zeichen in den Ausgabepuffer passen. Falls nicht, wartet das Programm auf das Freiwerden des Puffers.

6.1. User Kommunikation praktisch

In Verbindung mit technologischen Prozessen arbeitet der MC größtenteils stand alone, also ohne angeschlossenen PC. Auch aus diesem Grund darf das Programm keine User-Eingabe anfordern und warten, sondern muss mit den Informationen auskommen, die es hat. Zwecks Kommunikation ergreift der User die Initiative, das heißt, er schließt den PC an den MC an, startet Arduino und öffnet den seriellen Monitor, um nach Belieben zu kommunizieren. Danach wird die Verbindung getrennt.

> Dabei entsteht das folgende Problem: Beim Öffnen des seriellen Monitors führt der MC einen Reset aus, was im regulären Betrieb nicht sein darf. Abhilfe schafft ein Kondensator von 100 nF zwischen Reset-Pin und VCC. Weil dieser Kondensator auch Uploads verhindert, muss er während der Programmentwicklung entfernt werden.

Beispiele wie *fun* oder *base* arbeiten dagegen für den *User*, weshalb der serielle Monitor unverzichtbar ist. Der User gibt ein, was geschehen soll, und darauf warten *Tasks*, aber nicht das Programm.

6.2. Der serielle User Input von mtRTOS

Die Möglichkeit, dass der User *jederzeit* eine Eingabe machen kann, erfordert eine Information über den Zweck der Eingabe. (Bei grafischen Benutzeroberflächen steckt diese Information im benutzten Eingabefeld des betreffenden Fensters.) Task SUI erwartet dafür einen Kennbuchstaben, der auch ID oder id genannt wird. Sowohl der User als auch das Programm verbinden mit einer bestimmten ID den gleichen Zweck, das heißt, der User muss nicht nur die ID, sondern auch ihre Bedeutung kennen. Zweckmäßig stellt die Anwendung dafür ein Menü bereit, siehe 6.3.

Die Eingabe beginnt mit der ID. Es können bis zu vier Zahlen folgen, die in der Bibliothek mtRTOS als long-Zahlen gespeichert werden.

Um einen Vorgang auszulösen, der *keine* Zahleneingabe erfordert, wird nur die ID eingegeben. Die Eingabetaste (LF) allein wirkt wie eine spezielle ID.

Die User-Eingabe hat das folgende Format:

```
      Serial user input format
LF  or  id LF  or  id num (num) (num) (num) LF    (blank(s) as separator)
      Format of num (digits 1 2 3 as example)
#  or  123  or  -123  or  1.23  or  -1.23
```

Es gibt zwei Besonderheiten:

- Das Zeichen # steht für “keine Zahl“ und wird intern durch die kleinste long-Zahl (negativ) repräsentiert, die als NONB definiert ist. Es stellt die numerische Entsprechung von Ausnahmen dar. Beispiele: Wenn der User statt einer Zykluszeit „ausnahmsweise“ das Zeichen # eingibt, dann stoppt der Zyklus. Ein Messwert, der ausnahmsweise nicht zu ermitteln ist, wird als NONB gespeichert und mit dem Zeichen # ausgegeben.
- Der User muss den Wert einer Variablen vom Typ float mit einem Dezimalpunkt eingeben. Diese Eingabe wird als 2stellige Dezimalzahl angesehen, unabhängig von der tatsächlich eingegebenen Stellenzahl. Die

Bibliothek mtRTOS speichert die Eingabe jedoch als long-Zahl, das heißt mit dem 100fachen des Wertes der Dezimalzahl. Zum Beispiel wird -0.123 als -12 gespeichert und 12.3 als 1230 .

Eine korrekte User Eingabe wird durch die Bibliotheks-Semaphore INPSEM signalisiert. Die Daten stehen in Variablen, die als *extern* deklariert sind, wodurch die Anwendung Zugriff hat. Somit besteht zwischen Bibliothek und Anwendung das folgende Dateninterface:

```
//Semaphore  INPSEM   ->  serial user input has taken place
//extern char id      ->  ID:   0 (LF), 'a'to'z' , 'A'to'Z'
//extern char inum    ->  number of numeric data, inum == 0 means input of id only
//extern long lnum [MAXNUM]  ->  array of numeric data
```

6.3. Wie das Dateninterface verwendet wird

Zum Verständnis dieses Abschnitts ist der Beispielsketch *serialcomm* erforderlich.

Task BLYELLOW von *serialcomm* lässt eine LED symmetrisch blinken, wobei der User die Zeit der halben Periode *yetime* ändern kann. Darüber hinaus kann er den Zyklus anhalten und freigeben.

Auch Task BLBLUE lässt eine LED blinken. Der User nimmt mit zwei Zahlen Einfluss auf den Zyklus, nämlich mit der Periodenzeit *period* und dem Tastverhältnis *ratio* als Dezimalzahl.

Task COMM

Da die Bibliothek mtRTOS eine User-Eingabe durch die Semaphore INPSEM signalisiert, muss eine Task der Anwendung darauf warten. Zweckmäßig wird diese Task in einem eigenen Tab am Ende des Sketches angesiedelt. Im Beispielsketch *serialcomm* handelt es sich um Task COMM in Tab *z_comm.ino*.

```
void comm2 () {
  mtsema (INPSEM, comm3); // User input has taken place
}
void comm3 () {           // evaluation of id
  if (id == 'i' ) {
    PR ("i idle: ");
    PRln ( getidle() );
  }
  else if (id == 'y' ) {
         mtcoop (do_yetime );
         return;
       }
       else if (id == 'b' ) {
              mtcoop (do_bltime );
              return;
            }
            else PRln ("Valid IDs: i idle, y yellow, b blue");
  mtcoop (comm2);
}
```

Wenn Semaphore INPSEM erscheint, prüft Taskfunktion *comm3* die vom User eingegebene ID. Ist sie gültig, wird sie ausgewertet, andernfalls wird eine Aufzählung der gültigen IDs, verbunden mit ihrem Zweck ausgegeben. Diese Aufzählung ist zweierlei: ein Hinweis darauf, dass eine falsche ID eingegeben wurde und ein kleines Menü. Wenn der

User am Menü interessiert ist, benutzt er zweckmäßig die Eingabetaste statt eine ungültige ID. Weil Text zu sparen ist, kann das Menü nur eine Gedächtnisstütze sein. Die Einzelheiten sollte der User kennen.

Das Auswerten einer ID kann sehr einfach sein, aber auch ziemlich komplex. Der User erhält in jedem Fall eine Rückmeldung.

Mit ID i lässt sich der User die CPU-Leerlaufkennzahl *idle* ausgeben, die von der Bibliotheksfunktion *getidle* zurückgegeben wird. (Es hätte auch irgendein anderer Vorgang sein können.) Zusätzlich eingegebene Zahlen werden nicht beachtet. Einfacher geht es nicht.

Die Auswertung der beiden IDs y und b ist wesentlich umfangreicher und deshalb in die Taskfunktionen *do_yetime* sowie *do_bltime* verlegt worden. In beiden Funktionen wird als erstes *inum* überprüft. Das ist die Anzahl der eingegebenen Zahlen. Die Zahleneingabe wird als SET-Vorgang bezeichnet. Wenn *inum* gleich Null ist, also keine Zahl eingegeben wurde, möchte der User die aktuellen Daten abfragen, was die entsprechende Ausgabe zur Folge hat. Es handelt sich dann um einen GET-Vorgang,

Taskfunktion do_yetime

Die Zahl, die mit ID y eingegeben wird, erscheint im Dateninterface und muss, wie bei den meisten SET-Vorgängen in bestimmten Grenzen liegen. Wenn das der Fall ist, arbeitet Task BLYELLOW mit der neuen Zeit, die auch dauerhaft im EEPROM gespeichert wird, siehe 7.

Gibt der User statt einer Zahl das Zeichen # ein, dann stoppt und startet Task BLYELLOW abwechselnd. Mit dem Aufruf der Funktion *stop* ist der Aufruf der Funktion *yestop* verbunden, die dafür sorgt, dass die LED bei jedem Stopp leuchtet. Das ist als Vorgabe anzusehen. Genauso hätte auch eine ausgeschaltete LED vorgegeben sein können.

Taskfunktion do_bltime

Taskfunktion *do_bltime* als Reaktion auf ID b erwartet im Dateninterface die beiden Zahlen, die den Blinkzyklus von Task BLBLUE bestimmen. Die erste ist die Periodenzeit und die zweite das Tastverhältnis als Dezimalzahl. Die beiden Zahlen werden akzeptiert, wenn sie innerhalb bestimmter Grenzen liegen, und in die LOW-Zeit *ltime* und die HIGH-Zeit *htime* umgerechnet, womit Task BLBLUE tatsächlich arbeitet. Auch diese beiden Zahlen werden im EEPROM gespeichert.

Bei diesem SET-Vorgang ist die Möglichkeit, mehr als eine Zahl einzugeben, von Bedeutung. Der Blinkzyklus arbeitet beim Abschluss der Eingabe mit *beiden* Daten. Würde der serielle User Input von mtRTOS nur *eine* Zahl akzeptieren, müssten zwei Eingaben nacheinander gemacht werden. In der Zwischenzeit wäre der Blinkzyklus nicht korrekt. Auch könnte die zweite Eingabe vergessen werden.

Beispiel serialcomm als Vorlage

Jede Anwendung unter mtRTOS kann das Programmierschema von Task COMM verwenden und auch das Hauptprogramm als Muster benutzen. Darüber hinaus kommt die Verwendung des EEPROM in *serialcomm* vor. Insofern ist *serialcomm* sehr gut als Vorlage für mtRTOS-Programme geeignet.

7. EEPROM

Der EEPROM ist für mtRTOS-Programme unverzichtbar, weil viele Daten nach einem Reset (gleichbedeutend mit Stromausfall) erhalten bleiben müssen. Das Schreiben in den

EEPROM bedeutet allerdings *Warten.* Weil dafür nur eine mt-Schleife in Frage kommt, wird auch dieses Problem durch die Bibliothek mtRTOS gelöst.

eewrite

Die Subtask *eewrite* schreibt Daten mit Hilfe der Subtaskfunktion *ee1* in den EEPROM. Der Vorgang dauert etwa 4 ms je Byte, auf dessen Abschluss jeweils mit einer mt-Schleife gewartet wird. *eewrite* kommt in den meisten Fällen nach einem seriellen User Input zum Einsatz, so wie in Task COMM von Beispielsketch *serialcomm* . Der folgende Aufruf betrifft die Variable *yetime*.

```
eewrite ( &yetime, sizeof (long), EEyetime, comm2) ;
```

Das erste Argument ist der Zeiger auf die Variable, und das zweite gibt die Größe der Variablen bzw. ihres Typs an. Es folgt die EEPROM-Schreibadresse. Sie ist 0, weil der Programmierer EEyetime als 0 definiert hat. Der Adressraum des EEPROM geht von 0 bis 1023, was 1 KByte entspricht.

> Um die Anordnung der Daten im EEPROM kümmert sich zweckmäßig der Programmierer, indem er Adressdefinitionen vornimmt.
>
> Man könnte die Adressvergabe auch dem Compiler überlassen, und zwar so:
>
> ```
> long eeyetime EEMEM;     //EEPROM Variable definieren.
> eewrite ( &yetime, sizeof (long), &eeyetime, comm2);
> ```
>
> Diese Methode hat den Nachteil, dass während der Entwicklung das Programm mit falschen Daten starten kann, weil der Compiler die Adressstruktur verändert hat.

Das letzte Argument ist der Zeiger auf die Taskfunktion *comm2*, mit der die aufrufende Task COMM fortgesetzt wird, wenn die Subtask endet.

eeread

Zum Auslesen des EEPROM dient die Funktion *eeread*. Sie ist nicht zeitkritisch und deshalb keine Subtask. Beim Start von Task COMM in Gestalt von Taskfunktion *comm1*, das heißt, nach einem Reset, wird der Inhalt des EEPROM unter Adresse EEyetime mit dem folgendem Aufruf in die Variable *yetime* kopiert:

```
eeread ( EEyetime, sizeof (long), &yetime);
```

Die Argumente sind die gleichen wie beim Aufruf von *eewrite* oben, nur sind sie anders angeordnet, und der Funktionszeiger fehlt.

Ausschluss-Semaphore EESEM

Solange Subtask *eewrite* arbeitet, darf es keinen weiteren Aufruf von *eewrite* und auch keinen Aufruf von *eeread* geben. Wenn das nicht einzuhalten ist, muss vor dem Aufruf auf die Ausschluss-Semaphore EESEM gewartet werden, die in der Bibliothek definiert ist.

Im Beispielsketch *serialcomm* werden auch die Variablen *period* und *ratio* im EEPROM gespeichert, nachdem sie gemeinsam eingegeben wurden.

Im Gegensatz zu den anderen EEPROM-Zugriffen in Task COMM, würde der Aufruf von *eewrite (&ratio, ...* die oben angegebene Forderung nicht erfüllen, weil er unmittelbar auf *eewrite (&period, ...* folgt. Deshalb kommt hier die Semaphore EESEM zum Einsatz. Wie das geschieht, geht aus Taskfunktion *do_bltime* hervor.

Warten

Zu beachten ist, dass nicht nur *eewrite* das Ende des Schreibvorgangs abwartet, sondern auch die aufrufende Task. Für Task COMM ist das ohne Bedeutung, da sie „nur“ serielle User Inputs auswertet.

8. Beispielsketch *fun*

Dieser Sketch ist der aufwändigste der Beispiele, weshalb ihm ein eigener Abschnitt gewidmet ist. Der Sketch realisiert drei völlig unterschiedliche Vorgänge, die ein bisschen Spaß machen sollen. Auslöser ist der User.

Der Reaktionstester in Gestalt von Task REAC läuft in einer endlosen mt-Schleife, die durch Drücken des Tasters verlassen wird, um die Reaktionszeit des Users zu messen. Danach wird die Schleife fortgesetzt.

Task PLAY spielt eines von drei Musikstücken. Wenn der User die Nummer 0, 1 oder 2 eingibt, startet die Task. Sie stoppt, wenn das Ende des Stücks erreicht ist, oder wenn es durch die Eingabe des Zeichens # abgebrochen wird. Timer1, der die nötigen Frequenzen an einem Output Pin erzeugt, wird von Task PLAY entsprechend den Daten des Musikstücks gesteuert. Diese Daten liegen im ROM, und der Zugriff erfolgt mit Hilfe von Zeigern.

Task PRIME läuft wie Task REAC in einer endlosen mt-Warteschleife. Wenn der User das Zeichen # eingibt, wird zu einer weiteren Schleife gewechselt, die fortgesetzt Primzahlen berechnet. Mit der gleichen Eingabe kann die Berechnung angehalten werden.

Es geht bei Task PRIME nicht um die Ergebnisse, sondern darum, den MC zu Demonstrationszwecken vollständig auszulasten, wofür sich die Primzahlberechnung gut eignet. Dennoch erscheinen die Ergebnisse im seriellen Monitor, um zu zeigen, dass die Berechnung läuft und auch, um sie kontrollieren zu können. Eine für den User angemessene Ausgabefrequenz entsteht, wenn Primzahlen mit mindestens 10 Stellen berechnet werden. Deshalb beginnt die Berechnung mit der Zahl 1000000001, worauf der User keinen Einfluss hat.

Der User kann die Vorgänge so starten, dass sie gleichzeitig laufen, und er wird sehen bzw. hören, dass sie sich untereinander nicht beeinflussen. Das zu zeigen ist der eigentliche Zweck von Sketch *fun*.

9. Zum Schluss

Ich hoffe, dass durch diesen Beitrag einige meiner Leser auf den Geschmack gekommen sind und manche ihrer Projekte mit mtRTOS realisieren.