Realtime Multitasking für Mikrocontroller

mit Arduino

Dieter Holzhäuser www.wieundwarum.de 10.2025

Mit diesem Beitrag möchte ich den Freunden der Mikrocontroller-Programmierung nicht nur ein einfaches Echtzeit-Multitasking vorstellen, sondern auch das unbedingt notwendige echtzeit-kompatible Zubehör.

Vor etwa 5 Jahren bin ich auf die Arduino IDE umgestiegen und benutze insbesondere das Board Arduino Nano, aber auch dem Arduino UNO nachempfundene selbst entwickelte Boards und neuerdings den ATtiny85. Also kommt mein Echtzeit-Multitasking in Form einer Arduino-Bibliothek, die ohne jeden Assemblercode in C++ geschrieben ist. Den Multitasking-Kernel verwende ich seit mehr als 12 Jahren.

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

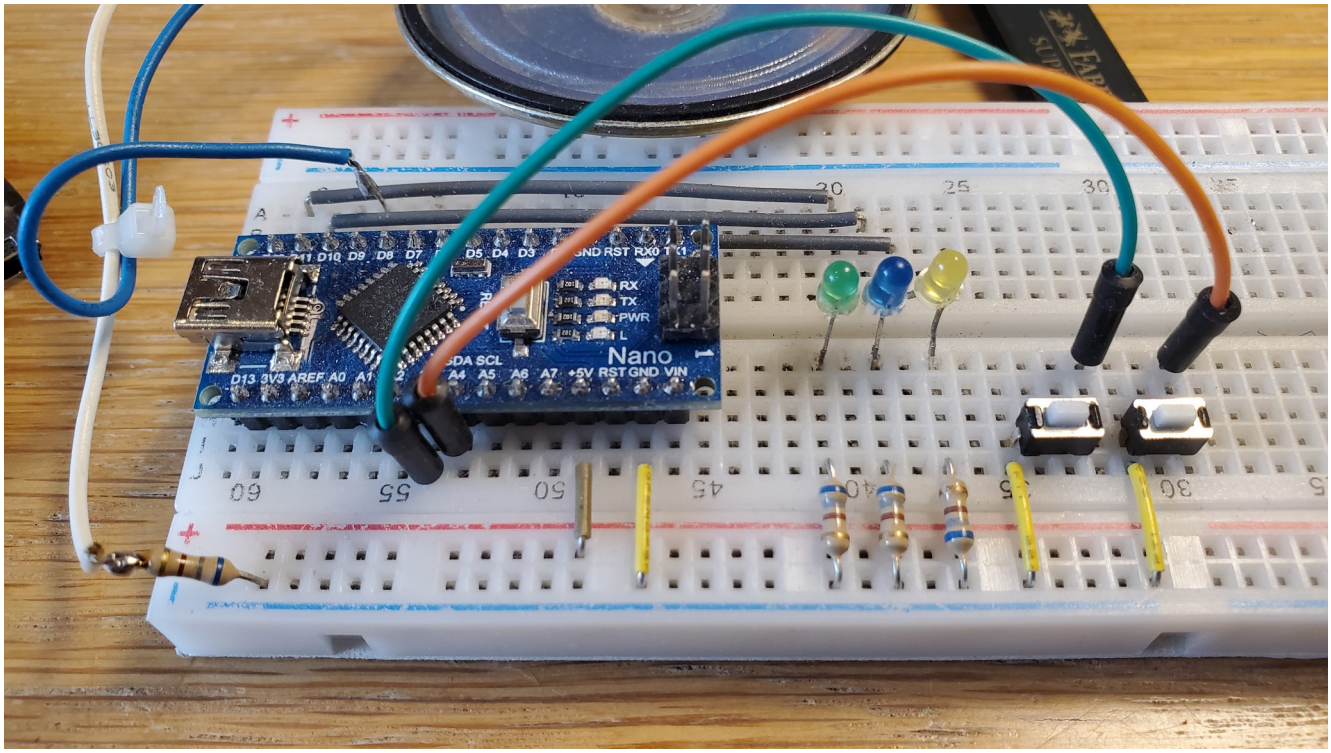
1. Hard- und Software

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

Die Tabs der Beispiele elements.ino und userinput.ino demonstrieren die Programmierschemata, nach denen mtos-Programme geschrieben werden.

Der Speicherbedarf des einfachst möglichen mtos-Sketches ist etwa 3 KB.

Die Hardware besteht aus dem Board Arduino Nano, an das drei LED und zwei 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 Beispiel-Sketch *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 Tab ELEM2 von Beispiel-Sketch elements.ino hochgeladen wird. Die drei LED blinken mit unterschiedlichen Frequenzen. Mit dem linken Taster (an Pin LBUTTON) kann das Blinken 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, wird es sehr schwierig.

Das liegt daran, dass die Hauptschleife kaum geeignet ist, unterschiedliche Dinge zu tun, insbesondere dann nicht, wenn sie zeitgerecht sein müssen.

Ein Blick auf den Code zeigt, wie einfach und strukturiert das Problem mit der Bibliothek mtos 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.
- Beim Umbau von Multitasking Programmen werden lediglich Tasks hinzugefügt oder entfernt.

Die skizzierte Heizungssteuerung wäre für mtos kein Problem, aber die Bibliothek mtos enthält keine so umfangreichen Beispiele. Weil der interessierte Programmierer nicht mal eben eine komplexe Peripherie aufbauen kann, 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 auch gezeigt werden, wenn die Tasks eines Programms Probleme lösen, die nicht zusammenhängen.

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 mtos **ereignisorientiert**.

3. mtos im Überblick

Der Multitasking Kernel bildet den Hauptteil der Bibliothek mtos. Neben der Verwaltung der Tasks, enthält er auch die Hauptschleife des Systems, die deshalb nicht mehr Sache des Programmierers ist.

Das Multitasking von mtos ist wegen seiner Ereignisorientierung sehr gut geeignet, technologische Prozesse zu begleiten (zu steuern). Solche Prozesse erzeugen Ereignisse in Form von Zustandsänderungen an Input-Pins. Auch ein mtos-Programm selbst erzeugt Ereignisse, und zwar Zeitabläufe und das Erscheinen von Semaphoren, siehe 5.5.

Der Programmlauf besteht prinzipiell darin, auf ein Ereignis zu warten, es mit Programmcode zu verarbeiten, wieder auf ein Ereignis zu warten usw. Die Steuerung dieses Vorgangs ist Sache des Kernels. Die Wartezeit auf das nächste Ereignis kann ausgeprägt sein und durchaus auch gegen Null gehen.

Es liegt auf der Hand, dass ein mtos-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 aber nicht angebracht, einem Rechner diese Eigenschaft zuzuschreiben, nur weil er besonders leistungsfähig ist. In der Praxis gibt es nur echtzeitfähige Systeme, weil sie tun, was sie sollen, andernfalls würden sie nicht eingesetzt. Systeme können jedoch durch ungeeignete Veränderungen ihre Echtzeitfähigkeit verlieren. Multitasking unterstützt Echtzeitfähigkeit, ist aber keine Voraussetzung dafür.

Die Bibliothek mtos bei geeigneter Programmierung echtzeitfähig, wenn Reaktionszeiten auf einen Hardware-Input von maximal 1 ms (im Ausnahmefall etwas mehr) zulässig sind. In speziellen Fällen können kürzere maximale Reaktionszeiten mit Interrupts erzielt werden, vorausgesetzt der Input Pin ist dafür geeignet.

Kaum ein Programm kommt ohne User-Kommunikation aus, und ohne die Möglichkeit, Daten unverlierbar im EEPROM des MC zu speichern. Übliche Implementierungen kommen unter mtos nicht in frage, denn sie würden die Echtzeitfähigkeit einer Anwendung durch Warteschleifen gefährden. Die Bibliothek stellt daher als weitere Komponenten den asynchronen seriellen User-Input und das echtzeit-kompatible Schreiben in den EEPROM zur Verfügung.

Diese beiden Komponenten und die im Multitasking-Kernel integrierte Hauptschleife sowie die mtos-Programmierschemata machen die Bibliothek mtos zu einem kleinen Betriebssystem.

4. Die Elemente der mtos-Programmierung

Jeder der 12 Tabs des Beispiel-Sketches elements.ino demonstriert ein oder mehrere dieser Elemente. Um einen der Tabs zum Laufen zu bringen, wird seine Definition in elements.ino dekommentiert. Es kann nur ein Tab laufen.

elements.ino enthält die notwendigen Hardware-Definitionen. Genauso wie die Pins des MC stellen Tasks und Semaphoren intern Zahlen dar (jeweils 0-18), weshalb es unverzichtbar ist, Namen (Symbole) für die verwendeten zu definieren.

Tasks müssen irgendwo gestartet werden, damit sie arbeiten. Meistens geschieht das im Arduino setup mit os.start, nachdem der Kernel mit os.initmt initialisiert wurde. Das setup endet mit dem Aufruf von os.runner. Das ist die endlos laufende Hauptschleife von mtos, deren Aufgabe es ist, Taskfunktionen laufbereiter Tasks aufzurufen.

Die Daten und Funktionen des mtos-Kernels haben ihre „Heimat“ in der Klasse Mtos. Den Zugriff darauf ermöglicht das Objekt os dieser Klasse.

4.1. ELEM1 Task, Taskfunktionen, mt-Funktionen

```
/////Task GPCYCLE        //general purpose cycle
void gpcycle1 () {       //to add own and other inits
  os.mtcoop (gpcycle2);
}
void gpcycle2 () {      //Task function as endless loop -> 1000 ms cycle
  WPIN(LED_BUILTIN, !RPIN(LED_BUILTIN));    //toggle LED as sign of life,
  os.mtdelay (1000, gpcycle2);
}
```

Task GPCYCLE ist eine „Allzweck-Task“, die in jedem Programm vorhanden sein sollte, um sie ggf. zu erweitern.

Sie besteht aus zwei Taskfunktionen. Taskfunktionen haben keine Argumente und keine Wertrückgabe. gpcycle1 ist die Startfunktion (siehe setup), die nach dem Start der Task einmal läuft. Sie kann z.B. Initialisierungen vornehmen, was aber in ELEM1 nicht nötig ist. Insofern könnte auch gpcycle2 die Startfunktion sein.

gpcycle1 ruft, wie jede Taskfunktion eine mt-Funktion auf, und zwar mtcoop. Diese mt-Funktion weist den Kernel an, sobald wie möglich die Taskfunktion gpcycle2 aufzurufen. gpcycle2 stellt eine mtos-Endlosschleife dar, die die eingebaute rote LED von Arduino-Nano mit einer Halbperiode von 1000 ms blinken lässt. Der Zyklus entsteht durch Aufruf der mt-Funktion mtdelay. Sie weist den Kernel an, den Timer der Task mit 1000 ms zu aktivieren und bei Eintritt des Ereignisses Zeitablauf wiederum gpcycle2 aufzurufen. Das Blinken der LED soll als Lebenszeichen des Systems verstanden werden.

Durch den Aufruf von mt-Funktionen sorgt eine Taskfunktion dafür, dass die Task fortgesetzt wird. Das kann durch den Angabe einer anderen Taskfunktion geschehen, aber auch der eigenen, wie gpcycle2, wodurch der beschriebene Zylus entsteht. (Es gibt noch 2 weitere mt-Funktionen, nämlich mtpin und mtsema, siehe 5).

Es ist zweckmäßig, Taskfunktionen mit dem kleingeschriebenen Tasknamen und angehängter Nummer zu benennen, es sei denn sie tun etwas Spektakuläres, das durch einen entsprechenden Namen zum Ausdruck kommen soll.

> Prinzip der mtos-Tasks:
> Sie bestehen fast immer aus mehreren Taskfunktionen. Eine Taskfunktion bearbeitet ein vorangegangenes Ereignis, und zwar solange bis sie auf ein weiteres Ereignis warten muss. Sie beauftragt den Kernel durch Aufruf einer mt-Funktion, dieses Ereignis zu erkennen. Wenn es eintritt, setzt runner die Task

durch Aufruf der als Argumeht angegebenen Taskfunktion fort usw. (In diesem Sinn kann der Aufruf von mtcoop als „Sofortereignis“ angesehen werden.) Zu jeder Taskfunktion existiert ein Ereignis und umgekehrt.

Wenn eine Taskfunktion zum runner zurückkehrt, kann er die Taskfunktion irgendeiner anderen Task aufrufen, was ein sehr einfacher Taskwechsel ist. Das heißt: Taskfunktionen werden niemals direkt, sondern nur mit Hilfe von mt-Funktionen „irgendwann“ vom runner aufgerufen. Ruft eine Taskfunktion keine mt-Funktion auf, dann endet die Task.

4.2. ELEM2 Taskdaten, Inter Task Operation 1

```
Tnum fl (1,0);     //Tnum object to provide common task data nb[0] and nb[1]
                   //Note: short alternatives  OA and OB

#define FLAG  fl.nb[0]    //special short alternative

//////////////////Task TYELLOW     symmetric flashing
void tyellow1 () {
  if (FLAG)
    WPIN(YELLOW , !RPIN( YELLOW ) );
  else WPIN(YELLOW , HIGH);
  os.mtdelay (100, tyellow1);
}
//////////////////Task TGREEN      asymmetric flashing
void tgreen1 () {
  WPIN(GREEN, HIGH);
  os.mtdelay (100, tgreen2);
}
void tgreen2 () {
  if (FLAG) WPIN(GREEN, LOW) ;
  os.mtdelay (1000, tgreen1);
}
//////////////////Task BLBLUE    "Higher frequency" flashing
void tblue1 () {
  if (FLAG)
    WPIN(BLUE , !RPIN(BLUE) );
  else WPIN(BLUE, HIGH);
  os.mtdelay (25, tblue1);
}

///////////Task LPUSH     detect push of button and toggle FLAG
void lpush1 () {
  os.mtpin (LBUTTON, LOW, lpush2);   //wait for pushbutton is pressed
}
void lpush2 () {
  FLAG = !FLAG;
  os.mtcoop (lpush1);
}
```

Es gibt 3 Tasks, die so einfach sind, dass sie mit einer Taskfunktion auskommen. Jede lässt eine LED in unterschiedlicher Weise blinken, allerdings nur, wenn FLAG gleich 1 ist. Ist FLAG gleich 0, zeigen die LED Dauerlicht.

Task LPUSH wartet mit Taskfunktion lpush1 und mt-Funktion mtpin auf die fallende Flanke an Nano Pin LBUTTON, das heißt darauf, dass der daran angeschlossene Taster gedrückt wird. Daraufhin wird in Taskfunktion lpush2 FLAG umgeschaltet und die Task mit lpush1

fortgesetzt. Dadurch wird auf den nächsten Tastendruck gewartet usw. Insofern ist auch Task LPUSH eine endlos laufende Task.

Task LPUSH übt über FLAG Einfluss auf die drei Blink-Tasks aus, was als eine Art von Inter Task Operation ist.

Der Code der Allzweck-Task GPCYCLE, die den Start der übrigen Tasks übernimmt, ist hier nicht dargestellt.

FLAG ist ein Symbol für fl.nb[0]. fl ist ein Objekt der Klasse Tnum und stellt u.a. ein Array nb aus zwei long-Zahlen zur Verfügung, die von Tasks benutzt werden können. FLAG ist also eine long-Zahl, die zum flag umgewidmet wird, ohne Typwandlung auf den Typ bool.

4.3. ELEM3 Inter Task Operation 2 und 3

```
/////////////////Task TYELLOW      symmetric flashing
void tyellow1 () {
  WPIN(YELLOW , !RPIN( YELLOW ) ) ;
  os.mtdelay (100, tyellow1);
}

/////////////Task RPUSH    detect push of button and start or stop task TYELLOW
void yellon () {        //called in case of stop
  WPIN(YELLOW, HIGH);
}
/////
void rpush1 () {
  os.mtpin (RBUTTON, LOW, rpush2);   //waiting for pushbutton is pressed
}
void rpush2 () {
  if (os.is_stop(TYELLOW) )
    os.start (TYELLOW, tyellow1);
  else os.stop (TYELLOW, yellon);
  os.mtcoop (rpush1);
}

///////////////Task TBLUE     symmetric flashing
void tblue1 () {
  os.mtsema (GPSEMA, tblue2 );     //use semaphore
}
void tblue2 () {
  os.signal (GPSEMA);           //restore semaphore
  WPIN(BLUE , !RPIN(BLUE) ) ;
  os.mtdelay (700, tblue1);
}

///////Task LPUSH   detect push of button and aktivate/deactivate Task TBLUE by semaphore
void lpush1 () {
  os.mtpin (LBUTTON, LOW, lpush2);   //wait for button is pressed
}
void lpush2 () {
  if (os.issema(GPSEMA )) {
    os.clearsema(GPSEMA);         //steal semaphore (deactivate)
    WPIN(BLUE, HIGH);
  }
  else os.signal (GPSEMA );      //give back semaphore (activate)
  os.mtcoop (lpush1);
}
```

Task TYELLOW lässt die LED YELLOW symmetrisch blinken. Mit einem Taster an Pin RBUTTON soll das Blinken angehalten oder fortgesetzt werden. Es ist Aufgabe von Task RPUSH mit rpush1 auf diesen Tastendruck zu warten und den Vorgang mit rpush2 auszuführen. Dazu wird die start/stop-Methode angewandt. Wenn Task TYELLOW gestoppt ist (Abfrage-Funktion is_stop), wird sie gestartet, andernfalls gestoppt. Im zweiten Argument der Funktion stop kann eine Funktion angegeben werden (hier yellon), die beim Stopp aufgerufen wird. Funktion yellon sorgt dafür, dass bei gestoppter Task TYELLOW die LED leuchtet. Die start/stop-Methode ist eine weitere Möglichkeit, Einfluss auf Tasks auszuüben.

Die dritte Möglichkeit besteht darin, eine Semaphore zu verwenden. In elements ist bereits die Allzweck-Semaphore GPSEMA definiert.

Das Blinken der LED durch Task TBLUE mit Hilfe der Semaphore anzuhalten und fortzusetzen, ist eine eher ungewöhnliche Anwendung einer Semaphore.

Der Lauf von Task TBLUE wird vom Vorhandensein der Semaphore GPSEMA bestimmt. In Taskfunktion tblue1 wird mit der mt-Funktion mtsema auf das Erscheinen von GPSEMA gewartet. Aber es ist nur dann ein Warten, wenn die Semaphore nicht vorhanden ist. In diesem Fall hört das Blinken auf. Erscheint die Semaphore oder ist sie vorhanden, geht es mit Taskfunktion tblue2 weiter. Weil durch ihre Verwendung eine Semaphore verschwindet, wird sie sofort mit der Funktion signal wiederhergestellt. Dann kann tblue2 die eigentliche Aufgabe der Task ausführen und den Blinkzyklus erzeugen. Die Semaphore steht zur erneuten Verwendung zur Verfügung usw.

Task LPUSH verarbeitet den Tastendruck eines Tasters an Pin LBUTTON. Um dadurch das Blinken von Task TYELLOW anzuhalten, wird ihr mit der Funktion clearsema die Semaphore GPSEMA „weggenommen“. Entsprechend wird ihr zum Fortsetzen des Blinkens mit der Funktion signal die Semaphore „zurückgegeben“. Ob das eine oder das andere zu tun ist, entscheidet die Funktion issema, die angibt, ob die Semaphore existiert oder nicht.

Wie bei Tab ELEM2 übernimmt die nicht dargestellte Task GPCYCLE den Start der Tasks. Zusätzlich gibt es den Aufruf os.signal (GPSEMA). Dadurch entsteht die Semaphore, und die LED BLUE blinkt nach einem Reset.

Der wesentliche Unterschied der drei Inter Task Operations ist, dass die start/stop-Methode sofort wirkt. Die beiden anderen Methoden (flag und Semaphore) wirken erst, wenn die Halbperiode des Blinkens endet. Das fällt besonders auf bei langen Blinkzyklen.

4.4. ELEM4 Endliche Schleifen

```
Tnum bl (0,5);         //Tnum object to provide task data

//////////Task TBLUE
void tblue1 () {
  bl.nb[0] = bl.nb[1] ;
  os.mtpin (LBUTTON, LOW, tblue2);  //wait for LBUTTON is pressed
}
void tblue2 () {                  //loop
  WPIN(BLUE, HIGH ) ;             //series of flashes
  os.mtdelay (500, tblue3);
}
void tblue3 () {
  WPIN(BLUE, LOW) ;
  if (--bl.nb[0] == 0)            //counter of loop
    os.mtdelay (1500, tblue1);      //end of loop
  else os.mtdelay (500, tblue2);    //repeat
}
```

Task TBLUE erzeugt 10 Lichtblitze, wenn der Taster an Pin LBUTTON gedrückt wird. Nach dem letzten Lichtblitz ist die Task bereit für den nächsten Tastendruck usw.

Die Taskfunktionen tblue2 und tblue3 bilden eine endliche Schleife, die 10mal durchlaufen wird. Dafür ist die Zählvariable bl.nb[0] unverzichtbar. Erreicht der Zähler 0, wird die Schleife beendet, und in Taskfunktion tblue1 wird der Zähler für den nächsten Durchgang mit 10 initialisiert. Dieser Wert ist in bl.nb[1] gespeichert. bl ist ein Objekt der Klasse Tnum, die Speicherplätze für Taskdaten bereitstellt.

Task TBLUE insgesamt ist eine endlose Schleife, die für jeden Durchgang einen Tastendruck benötigt.

Die mtos-Schleife ist echtzeitgerecht und so einfach, dass dem Programmierer kaum bewusst ist, eine Schleife programmiert zu haben.

4.5. ELEM5 lokal verwendete Subtask

```
Tnum bl (0,0);     //Tnum object saves flash duration and function pointer to continue task
LPUSH

//Subtask subblue is called (twice) by task LPUSH only, that's local use
void subblue1 () {      //single flash of LED BLUE
  WPIN(BLUE, HIGH ) ;
  os.mtdelay (bl.OA, subblue2);
}
void subblue2 () {
  WPIN(BLUE, LOW ) ;
  os.mtdelay (500, (void(*)())bl.OB );    //cast to function pointer type void(*)()
}

////////////////Task LPUSH
void lpush1 () {
  os.mtpin (LBUTTON, LOW, lpush2);   //wait for pushbutton is pressed
}
void lpush2 () {
  bl.OA = 800;
  bl.OB = (long) lpush3;      //save function pointer
  subblue1 ();                //shorttime flash
}
void lpush3 () {
  bl.OA = 1500;
  bl.OB = (long) lpush1;      //save function pointer
  subblue1 ();                //longtime flash
}
```

Mit Task LPUSH erscheint nach einem Tastendruck auf den Taster an Pin LBUTTON ein Lichtblitz von 800 ms Dauer und nach einer Pause einen weiterer von 1500 ms. Dafür wird jeweils die Subtask subblue1 aufgerufen, die einen Lichtblitz erzeugen kann.

Wenn an mehreren Stellen in einer Task oder in verschiedenen anderen Tasks 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 Argumente haben kann. Eine gute Alternative dazu ist, die Daten in ein Objekt der Klasse Tnum zu legen. Eine unverzichtbare Date ist der Funktionszeiger auf die Taskfunktion, mit der die Task fortgesetzt werden soll, wenn die Subtask endet. Zum Speichern des Zeigers in einem Objekt von Tnum muss er in den Typ long gewandelt werden und dieser umgekehrt in den Typ void(*)() , wenn die Subtask endet.

Subtasks sind nicht reentrant und dürfen nur aufgerufen werden, wenn ein Wiedereintritt ausgeschlossen ist. Das ist immer gegeben, wenn die Aufrufe in der derselben Task erfolgen (lokal), wie im Beispiel oben. Andernfalls ist eine Ausschluss-Semaphore zu verwenden, siehe 4.6.

4.6. ELEM6 global verwendete Subtask und gegenseitiger Ausschluss

```
Tnum ht (0,0);  //Tnum object saves flash duration and function pointer to continue task
LPUSH or RPUSH

void subblue1 () {          //single flash of LED BLUE
  WPIN(BLUE , HIGH ) ;
  os.mtdelay (ht.OA, subblue2);
}
void subblue2 () {
  WPIN(BLUE , LOW ) ;
  os.mtdelay (500, subblue3);
}
void subblue3 () {
  os.signal (GPSEMA);                  //end of exclusion
  os.mtcoop ( (void(*)())ht.OB );     //cast to function pointer type void(*)()
}
/////////////Task LPUSH
void lpush1 () {
  os.mtpin (LBUTTON, LOW, lpush2);   //waiting for pushbutton is pressed
}
void lpush2 () {
  os.mtsema (GPSEMA, lpush3);        //if applicable, wait for end of exclusion
}
void lpush3 () {
  ht.OA = 800;
  ht.OB = (long) lpush1;          //save function pointer
  subblue1 ();                    //call subtask, continue by calling lpush1
}

////////////////Task RPUSH
void rpush1 () {
  os.mtpin (RBUTTON, LOW, rpush2);   //waiting for pushbutton is pressed
}
void rpush2 () {
  os.mtsema (GPSEMA, rpush3);        //if applicable, wait for end of exclusion
}
void rpush3 () {
  ht.OA = 3000;
  ht.OB = (long) rpush1;     //save function pointer
  subblue1 ();               //call subtask, continue by calling rpush1
}
```

Jeder der beiden Taster an Pin RBUTTON und LBUTTON möchte einen individuellen Lichtimpuls mit LED BLUE erzeugen. Dafür muss die Subtask subblue1 sowohl in Task LPUSH als auch in Task RPUSH aufgerufen werden. In diesem Fall wäre ein Wiedereintritt in die Subtask subblue1 möglich (siehe auch 4.5), was verhindert werden muss. Das ist analog zu einer knappen Ressource, auf die nicht gleichzeitig mehreren Verwendern zur Verfügung steht.

Die Semaphore GPSEMA ermöglicht den gegenseitigen Ausschluss (mutual exclusion). Nur die Task, die die Semaphore hat oder bekommt, kann die Subtask aufrufen. Deshalb muss die Subtask die Semaphore mit signal erzeugen bevor sie endet. Aber auch nach

Reset muss die Semaphore erzeugt werden (siehe auch die hier nicht dargestellte Taskfunktion gpcycle1).

Damit es gerecht zugeht, sieht mtos eine Warteschlange für die Aufrufe der mt-Funktion mtsema vor, siehe auch 4.8

Die üblichen Funktionen können von jeder Taskfunktion ohne Einschränkung aufgerufen werden. Sie werden Teil der betreffenden Task, solange sie laufen.

4.7. ELEM7 endliche Task statt Subtask

```
Tnum bl (0,0);     //Tnum object saves flash duration

///////Finite Task TBLUE creates a single flash of LED BLUE
void tblue1 () {
  WPIN(BLUE , HIGH ) ;
  os.mtdelay (bl.OA, tblue2);
}
void tblue2 () {
  WPIN(BLUE , LOW ) ;
  os.mtdelay (300, tblue3);
}
void tblue3 () {
  os.signal (GPSEMA);     //end of exclusion
}                         //no os.mt function call -> task stops -> finite task

////////////////Task LPUSH
void lpush1 () {
  os.mtpin (LBUTTON, LOW, lpush2);   //wait for pushbutton is pressed
}
void lpush2 () {
  os.mtsema (GPSEMA, lpush3);        //if applicable, wait for end of exclusion
}
void lpush3 () {
  bl.OA = 800;
  os.start (TBLUE, tblue1);       //flash 800 ms
  os.mtcoop (lpush1);
}

////////////////Task RPUSH
void rpush1 () {
  os.mtpin (RBUTTON, LOW, rpush2);   //wait for pushbutton is pressed
}
void rpush2 () {
  os.mtsema (GPSEMA, rpush3);        //if applicable, wait for end of exclusion
}
void rpush3 () {
  bl.OA = 3000;
  os.start (TBLUE, tblue1);       //flash 3000 ms
  os.mtcoop (rpush1);
}
```

ELEM7 leistet das gleiche wie ELEM6. Allerdings wird, statt einer Subtask, die endliche Task TBLUE gestartet. Der gegenseitige Ausschluss mit Semaphore GPSEMA ist auch hier erforderlich. Der Unterschied zu ELEM6 ist, dass Task TBLUE den Lichtimpuls erzeugt, ohne dass die Tasks LPUSH und RPUSH auf das Ende des Vorgangs warten müssen, nachdem sie TBLUE gestartet haben. Das kann durchaus bei komplexeren Aufgaben von Bedeutung sein.

4.8. ELEM8 faire Zuteilung

```
/////////////////Task TYELLOW    single flash triggered by semaphore
void tyellow1 () {
  WPIN(YELLOW, LOW) ;
  os.mtsema (GPSEMA, tyellow2 );
}
void tyellow2 () {
  WPIN(YELLOW, HIGH);
  os.mtdelay (100, tyellow1);
}

/////////////////Task TGREEN     dto.
void tgreen1 () {
  WPIN(GREEN, LOW) ;
  os.mtsema (GPSEMA, tgreen2 );
}
void tgreen2 () {
  WPIN(GREEN, HIGH);
  os.mtdelay (100, tgreen1);
}

///////////////Task TBLUE        dto.
void tblue1 () {
  WPIN(BLUE, LOW) ;
  os.mtsema (GPSEMA, tblue2 );
}
void tblue2 () {
  WPIN(BLUE, HIGH);
  os.mtdelay (100, tblue1);
}

////////Task LPUSH     detect push of button and transform pin event to semaphore event
void lpush1 () {
  os.mtpin (LBUTTON, LOW, lpush2);  //wait for pushbutton is pressed
}
void lpush2 () {
  os.signal (GPSEMA);
  os.mtcoop (lpush1);
}
```

Jede der drei Tasks TYELLOW, TGREEN und TBLUE möchte mit der jeweiligen LED durch einen Lichtimpuls signalisieren, dass der Taster an Pin LBUTTON gedrückt wurde, was aber nur einer Task zur selben Zeit erlaubt ist.

Task LPUSH erwartet das Pin-Ereignis. In Taskfunktion lpush2 wird es mit signal in ein Semaphore-Ereignis (GPSEMA) gewandelt, auf das jede der drei Tasks wartet. Da jeweils nur eine Task die Semaphore erhalten kann, entsteht ein gegenseitiger Ausschluss. Dadurch ist die Forderung, dass zur selben Zeit nur eine Task den Tastendruck signalisieren darf, erfüllt. Eine interne Warteschlange für Tasks, die auf eine Semaphore warten, sorgt für eine gerechte Zuteilung, das heißt, jede der drei Tasks “kommt mal dran“.

4.9. ELEM9 keine Zuteilung

```
/////////////////Task TYELLOW    single flash triggered by semaphore
void tyellow1 () {
  WPIN(YELLOW, LOW) ;
  os.mtpin (LBUTTON, LOW, tyellow2 );
}
void tyellow2 () {
  WPIN(YELLOW, HIGH);
  os.mtdelay (100, tyellow1);
}

/////////////////Task TGREEN     dto.
void tgreen1 () {
  WPIN(GREEN, LOW) ;
  os.mtpin (LBUTTON, LOW, tgreen2 );
}
void tgreen2 () {
  WPIN(GREEN, HIGH);
  os.mtdelay (100, tgreen1);
}

///////////////Task TBLUE        dto.
void tblue1 () {
  WPIN(BLUE, LOW) ;
  os.mtpin (LBUTTON, LOW, tblue2 );
}
void tblue2 () {
  WPIN(BLUE, HIGH);
  os.mtdelay (100, tblue1);
}
```

Auch in ELEM9 signalisieren drei Tasks (wie in ELEM8) einen Tastendruck, nur ohne Einschränkung. Die Umwandlung des Pin-Ereignisses in ein Semaphore-Ereignis ist überflüssig und eine Task, in der das geschieht, ebenso. Es genügt, wenn jede der drei Tasks auf das Pin-Ereignis wartet.

Anders als beim Semaphoren-Ereignis wird jede der drei Tasks laufbereit. Auch für laufbereite Tasks gibt es intern eine Warteschlange. Die Taskfunktionen tyellow2, tgreen2 und tblue2 werden daher vom runner nacheinander aufgerufen. Das geht so schnell, dass nach außen hin die Reihenfolge nicht zu bemerken ist, das heißt, der Tastendruck wird durch drei gleichzeitige Lichtimpulse signalisiert. Oder so: Eine Zuteilung gibt es nicht.

4.10. ELEM10 Timeout beim Pin-Ereignis

```
Tnum bl (BLUE, 800);

////////Task LPUSH
void lpush1 () {
  WPIN (bl.OA, HIGH) ;
  os.mtpin (LBUTTON, LOW, lpush2);  //wait for pushbutton is pressed
  os.mtdelay (300, lpush3);         //timeout (push of button not in time);
}
void lpush2 () {            //push of button in time, change LED (colour)
  if (bl.OA == BLUE) {
    bl.OA = GREEN;
    WPIN (BLUE, LOW);
```

```
  }
  else {
    bl.OA = BLUE;
    WPIN (GREEN, LOW);
  }
  os.mtdelay (bl.OB, lpush1);
}
void lpush3 () {            //push of button not in time, continue flashing with same LED
  os.mtdelay (bl.OB - 300, lpush4);
}
void lpush4 () {
  WPIN (bl.OA, LOW) ;
  os.mtdelay (bl.OB,lpush1);
}
```

Es ist eine besonders nützliche Eigenschaft von mtos, dass die mt-Funktionen mtpin und mtsema mit mtdelay kombiniert werden können. Die beiden Kombinationen wirken als Timeout. Das Ereignis, das zuerst erscheint, hat „gewonnen“. Die jeweils zugeordnete Taskfunktion wird ausgeführt und die andere nicht. ELEM10 demonstriert den Timeout für das Pin-Ereignis.

Die Grundfunktion von Task LPUSH ist, die LED, die in bl.OA gespeichert ist, symmetrisch mit der in bl.OB gespeicherten Zeit blinken zu lassen. Dafür sind die Taskfunktionen lpush1, lpush3 und lpush4 zuständig. In lpush1 sind mtdelay und mtpin kombiniert . Das ergibt den Timeout beim Pin-Ereignis. Wenn innerhalb von 300 ms, nachdem die LED eingeschaltet wurde, der Taster an Pin LBUTTON gedrückt wird, wird statt lpush3 die Taskfunktion lpush2 aufgerufen. lpush2 trägt die jeweils andere der beiden LED BLUE und GREEN in bl.OA ein. Das heißt, jeder rechtzeitige Tastendruck wechselt die Blinkfarbe.

4.11. ELEM11 Überwachung der CPU-Belastung

```
//////////////////Task TYELLOW     symmetric flashing
void tyellow1 () {
  delayMicroseconds (50);          //simulation of CPU peek load, try: 0, 50, 200, 700
  WPIN(YELLOW , !RPIN( YELLOW ) );
  os.mtdelay (1000, tyellow1);        //try: 100, 500, 1000
}

/////Task GPCYCLE        //general purpose cycle
void gpcycle1 () {       //to add own and other inits
  os.start (TYELLOW, tyellow1);
  os.mtcoop (gpcycle2);
}

void gpcycle2 () {      //Task function as endless loop
  WPIN(LED_BUILTIN, !RPIN(LED_BUILTIN));    //toggled LED as sign of life,
  PR ("idle: ");
  PRln ( os.idle );
  os.mtdelay (1100, gpcycle2);    //try: 800, 1000, 1100, 2000
}
```

Task TYELLOW lässt zwar die LED YELLOW blinken, aber, wie bei allen Tasks der Tabs von elements.ino, wird dadurch die CPU kaum belastet. Um eine deutliche CPU-Last zu simulieren, enthält Taskfunktion tyellow1 die herkömmliche Arduino-Warteschleife delayMicroseconds.

Die Allzweck-Task GPCYCLE hat in ELEM11 die Aufgabe, zyklisch die Zahl os.idle im seriellen Monitor auszugeben. (Die Definition PR steht für Serial.print.)

Was bedeutet die Zahl in os.idle?

Der runner zählt fortlaufend seine Leerlaufzyklen während der Zeit, die er auf eine laufbereite Task warten muss, um die nächste Taskfunktion zu starten. In jeder Millisekunde wird diese Zahl geprüft und zurückgesetzt. Je kleiner sie ist, desto höher ist die CPU-Last. Zu Beginn jeder Sekunde wird die kleinste Zahl der Leerlaufzyklen je Millisekunde (höchste CPU-Last), die es in der Sekunde zuvor gegeben hat, in os.idle gespeichert.

Man kann Tests anstellen, wobei drei Werte geändert werden können:

die CPU-Last mit delayMicroseconds in tyellow1,
der Zyklus, in dem die CPU-Last auftritt, ebenfalls in tyellow1
der Anzeigezyklus in gpcycle2

Es werden fast immer stark schwankende Werte ausgegeben. Das liegt daran, dass sich die Phasen der Zyklen von tyellow1 und gpcycle2 sowie dem internen Auswertezyklus je nach Testwert gegeneinander verschieben. Es kann dann Phasen geben, in denen keine oder maximale CPU-Last registriert wird. Dazu kommt, dass auch die Ausgabe mit PR (Serial.print) eine gewisse CPU-Last bedeutet.

Eine bessere Möglichkeit, die CPU-Last zu untersuchen, ist in Beispiel userinput.ino zu finden.

4.12. ELEM12 echtzeitgerechte Langzeitberechnungen

```
//////////////Task TBLUE     "Higher frequency" flashing
void tblue1() {
  WPIN(BLUE, !RPIN(BLUE));
  os.mtdelay ( 20 , tblue1);   //try 1 and check with DSO
}
#define CALC LPUSH       //Task LPUSH is now task CALC

Tnum ca (0, 0);

////////Task CALC
void calc1() {
  delayMicroseconds (900);   //simulates part one of a calculation
  ca.OA++;
  os.mtcoop(calc2);
}
void calc2() {
  delayMicroseconds (800);   //simulates part two of a calculation
  os.mtcoop(calc3);
}
void calc3() {
  delayMicroseconds (950);   //simulates part three of a calculation
  os.mtcoop ( calc1);
}

/////Task GPCYCLE        //general purpose cycle
void gpcycle1() {        //to add own and other inits
  os.start(CALC, calc1);
  os.start(TBLUE, tblue1);
  os.mtcoop(gpcycle2);
}
void gpcycle2() {                        //Task function as endless loop
```

```
  WPIN(LED_BUILTIN, !RPIN(LED_BUILTIN));  //toggled LED as sign of life
  PR (ca.OA);        //counter shows that task CALC is working
  PR (" ");
  PRln (os.idle);    //always zero  (max. CPU load)
  os.mtdelay(2000, gpcycle2);
}
```

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.

Für den Programmfortschritt braucht es Ereignisse. Trotzdem ist mtos für Tasks ohne Ereignisse, z.B. für langwierige Berechnungen geeignet, wenn sie kooperativ gestaltet werden.

Die in CALC umbenannte Task LPUSH ist eine solche Task. Die eingefügten Arduino-Warteschleifen mit delayMicroseconds verbrauchen Rechenzeit und sollen eine langwierige Berechnung simulieren. Die Kooperation von Task CALC besteht darin, dass die gesamte Rechnung auf drei Taskfunktionen verteilt ist, von denen jede weniger als 1 ms Rechenzeit braucht. Weil es keine wirklichen Ereignisse gibt, wird für den Übergang die mt-Funktion mtcoop benutzt. Wie bei allen mt-Funktionen geht mit mtcoop die Kontrolle an den runner über. Er arbeitet dann die Warteschlange der laufbereiten Tasks ab, in der Task CALC an letzter Stelle steht.

Task TBLUE wird durch die Langzeitberechnungen von Task CALC nicht beeinflusst und arbeitet zeitrichtig. Man kann das mit einem Oszilloskop und der höchstmöglichen Blinkfrequenz kontrollieren.

Task CALC wird „nachrangig“, also im Hintergrund abgearbeitet und nutzt die andernfalls verschenkte Rechenkapazität der CPU voll aus. Dadurch gibt es keine Leerlaufzyklen und die Zahl in os.idle ist 0. Mit Task GPCYCLE wird zyklisch, neben os.idle, ein Zähler ausgegeben, der anzeigt, dass Task CALC arbeitet.

Das System kommt an seine Grenzen, wenn eine längere Berechnung nicht kooperativ gestaltet werden kann, z.B. wenn langlaufende Funktionen fremder Bibliotheken eingebunden werden.

5. mt-Funktionen genauer betrachtet

Taskfunktionen arbeiten gemeinsam an einem abgegrenztem Problem, für das die Task bzw. der Taskname steht. Eine oder zwei mt-Funktionen müssen in jeder Taskfunktion aufgerufen werden. Sie sind das Bindeglied dieser zusammengehörenden Taskfunktionen her.

> Der Programmierer hat darauf zu achten, dass nur die Taskfunktionen ein und derselben Task auf diese Weise verbunden werden. Eine mt-Funktion könnte durchaus eine Taskfunktion einer anderen Task benennen, was unbedingt zu vermeiden ist. Entsprechende Tasknamen helfen dabei.

Nicht nur der Kernel, sondern auch mt-Funktionen ändern den Taskzustand.

5.1. Taskzustände

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
- Vom Aufruf einer Taskfunktion durch den runner bis zu ihrer Rückkehr 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     //Time
WAITING_B       39     //Input pin edge
WAITING_TS      31     //Both Time or Sema
WAITING_TB      55     //Both Time or Input pin edge
```

Funktion os.get_taskstate gibt die Zustandskennzahl der als Argument angegebenen Task zurück.

5.2. Übersicht der mt-Funktionen

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

```
mt-Funktion  Ereignis           Erzeuger des Ereignisses              neuer Taskzustand

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

Gemeinsam ist allen mt-Funktionen das Argument als 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 den Kernel verstanden werden, das zugeordnete Ereignis zu erkennen, und, wenn es eingetroffen ist, sobald wie möglich die spezifizierte Taskfunktion aufzurufen.

5.3. 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. Der Taskzustand ist WAITING_T Erreicht der jeweilige Zeitzähler Null, wird die Task laufbereit (READY).

5.4. mtpin

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

5.5. 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 os.*signal* zu Semaphore-Ereignissen (semaphore event). Darauf kann eine andere Task mit mtsema warten, wobei sie den Zustand WAITING_S einnimmt. Erscheint das Semaphore-Ereignis oder ist es bereits vorhanden, wenn mtsema aufgerufen wird, wird die Task laufbereit (READY).

Selbst ein timeout event oder ein pin event kann zu einem oder zu mehreren semaphore events umgewidmet werden.

Hinter einer Semaphore verbirgt sich eine Zahl, für die ein Name definiert sein sollte. Diese Zahl ist nicht die Semaphore selbst, sondern steht für den Speicherort der Semaphore, die vorhanden sein kann oder auch nicht. Sie entsteht durch Aufruf von os.signal, und sie verschwindet, wenn eine Task, die mit mtsema darauf wartet, sie bekommt. Beim nächsten Aufruf von os.signal entsteht sie wieder usw.

Verschiedenartigste Ereignisse können durch die Bindung an Semaphoren durch einen einzigen Standard behandelt werden.

5.6. 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 Kernel dazu, unmittelbar die als Argument übergebene Taskfunktion aufzurufen. mtcoop ermöglicht den Übergang von einer Taskfunktion auf eine andere ohne ein wirkliches Ereignis. Das kann bei der Programmierung von Tasks erforderlich sein, dient aber insbesondere zur Kooperation. Beim Aufruf von mtcoop muss die Task im Zustand RUNNING sein, um sofort wieder laufbereit (READY) zu werden.

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. Eine solche Task begleitet keinen technologischen Prozess, sondern führt eine langwierige Berechnung aus (4.12). Sie muss mtcoop verwenden, um kooperativ zu sein. Kooperation heißt, die Kontrolle freiwillig und rechtzeitig mit mtcoop an den *runner* zurückzugeben, damit andere Tasks zum Zug kommen.

5.7. Mehrfachaufrufe von mt-Funktionen

Die beiden Mehrfachaufrufe mtpin und mtdelay sowie mtsema und mtdelay bilden den sehr wichtigen Timeout. Andere Mehrfachaufrufe haben keine funktionale Bedeutung, aber weil sie möglich sind, müssen sie definiert sein, um das System stabil zu halten.

Es spielt keine Rolle, an welcher Stelle in einer Taskfunktion eine mt-Funktion aufgerufen wird. Es kommt jedoch der Vorstellung von einem schlüssigen Programmablauf zugute, wenn das am Ende geschieht.

- Folgt mtsema, mtdelay oder mtpin auf mtcoop, wirkt mtcoop.
- Folgt mtcoop auf mtsema, mtdelay oder mtpin, wirkt mtcoop nicht.
- Bei den mt-Funktionen mtsema und mtpin gilt die zuletzt aufgerufene, auch in Kombination mit mtdelay (Timeout). Ausnahme: mtsema gilt, wenn beim Aufruf die Semaphore vorhanden ist.

6. Serielle User Kommunikation

Eine ereignisorientierte Multitasking Umgebung erfordert einen asynchronen User-Input (vergleichbar mit einem Ereignis), auf den echtzeitgerecht zu reagieren ist. (Ob das auch mit Serial-Methoden von Arduino möglich ist, bleibt offen.) Die Bibliothek mtos stellt für den seriellen User-Input eine eigene Lösung bereit, und zwar in Form von Task SUI (Serial User Input). Das bedeutet, der User hat die Initiative. Er kann *jederzeit* etwas eingeben, worauf das Programm reagiert, und zwar, ohne dass andere Tasks davon beeinflusst werden. Das bedeutet auch, dass die User-Eingabe eine Information über ihren Zweck enthalten muss (Bei grafischen Benutzeroberflächen steckt diese Information in der Beschriftung eines Feldes, das mit der Maus angeklickt werden kann.)

Der serielle Output, den Arduino mit Serial.print (als Kurzform PR definiert) bietet, ist dagegen überall in einem mtos-Programm möglich. Man braucht den seriellen Output als Antwort auf User-Inputs, zum Monitoring sowie temporär zur Fehlersuche. Zu beachten ist, dass die auszugebenden Zeichen in den 64 Byte Ausgabepuffer passen. Falls nicht, entstehen Wartezeiten, die die Echtzeitfähigkeit gefährden könnten. Deshalb geschieht insbesondere die Ausgabe des Menüs mit Zeitunterbrechungen.

Task SUI reagiert auf einen User-Input und erledigt alles, was im Zusammenhang damit steht. Um die Komplexität der Task in Grenzen zu halten, ist der User Input standardisiert. Er besteht aus dem Kommando, dem eine oder zwei Zahlen folgen können sowie dem Abschluss mit der Enter-Taste. Die Zahlen können verschiedene Formen haben, siehe unten. Wird nur die Enter-Taste gedrückt, wird die Liste aller Kommandos der jeweiligen Applikation ausgeben, die als mtos-Menü anzusehen ist.

```
Format: (cmd (number) (number)) LF       ( ) -> optional
cmd:    command string 1 to 6 characters
number: # || 123 || -123 || 001234 || 1.23 (~ 123) || -1.23 (~ -123)
        digits 1234 as example,   # ~  NONB (no number),
        001234 used as time format hhmmss, 1.2 or 1.234 -> error,
LF:     Enter
```

Als Task der Bibliothek mtos kennt Task SUI weder die Kommandos der jeweiligen Applikation noch die Aktivitäten, die bei einem bestimmten Kommando auszuführen sind. Deshalb ist Task SUI zu einem kleineren Teil in die Applikation ausgelagert, den der Programmierer ausgestalten muss. Insbesondere sind die Funktionen, die mit den Kommandos verbunden sind, Taskfunktionen von Task SUI, die mit ihren mt-Funktionen auf die Taskfunktion sui_clear von Task SUI verweisen müssen, z.B. durch
mtcoop (sui_clear);

Die Einzelheiten gehen aus dem Beispiel-Sketch userinput.ino hervor. Es besteht aus den Tabs SER1 bis SER5, die, wie im Beispiel elements.ino, einzeln zu compilieren sind, und die jeweils einen bestimmten Aspekt des seriellen User Inputs von Task SUI demonstrieren. Task SUI wird gestartet, indem sr.initserinp(); in das setup aufgenommen wird.

Im Objekt sr der Klasse Sui sind die Daten von Task SUI zu Hause sowie die Funktionen, die keine Taskfunktionen sind. Taskfunktionen können kein Mitglied einer Klasse sein, aber es ist vorteilhaft, ihren Inhalt oder ihre Substanz zu Methoden der Klasse zu machen. Die Klasse Sui geht diesen Weg, was die umfangreichen Taskfunktionen schlanker und ihr Zusammenspiel transparenter macht. Task SUI ist daher eine gute Vorlage für die Programmierung komplexer Tasks.

6.1. SER1 Kommando ohne Zahlen

```
const char ye [] PROGMEM = "ye -- YELLOW on/off";
const char gr [] PROGMEM = "gr -- GREEN on/off";
const char bl [] PROGMEM = "bl -- BLUE on/off";
struct cmdstruct cmds [] = { {ye, cmd_ye},
                             {gr, cmd_gr},
                             {bl, cmd_bl} };
char maxcmds = sizeof (cmds) / sizeof (cmdstruct) ;

//command functions
void cmd_ye () {
  WPIN(YELLOW, !RPIN(YELLOW));
  prtstate (RPIN(YELLOW) );
  mtcoop (sui_clear);      //end of serial user input activity
}
void cmd_gr () {
  WPIN(GREEN, !RPIN(GREEN));
  prtstate (RPIN(GREEN) );
  mtcoop (sui_clear);
}
void cmd_bl () {
  WPIN(BLUE, !RPIN(BLUE));
  prtstate (RPIN(BLUE) );
  mtcoop (sui_clear);
}
void prtstate ( bool s) {
  if (s) PRln("on"); else PRln("off");
}
```

Der gesamte Code dieses Tab-Beispiels, das keine eigene Task hat, gehört programmtechnisch zu Task SUI.

Das Array cmds ist in der Bibliothek mtos deklariert und wird in der Applikation definiert, das heißt, mit Daten ausgestattet. Es handelt sich um ein Array von Strukturen. Die erste Komponente ist der Kommando-String, und die zweite ist der Funktionszeiger der dem Kommando zugeordneten Funktion. Sie wird ausgeführt, wenn das Kommando eingegeben wurde. Ein Kommando kann maximal 8 Zeichen haben.

Der Kommando-String enthält nicht nur das Kommando, sondern auch Text der keine funktionale Bedeutung hat. Dieser Text soll den User unterstützen. Die Kennung aus zwei Zeichen nach dem Kommando weist auf die Art und Weise der Zahleneingabe hin, die von der Applikation erwartet wird.

Wenn der Sketch hochgeladen wird, gibt die nicht dargestellte Allzweck-Task GPCYCLE eine Begrüßung auf dem seriellen Monitor aus und auch den Hinweis, dass mit der Enter-Taste (Symbol LF) das Menü erscheint.

Die Kennung -- im Array cmds bedeutet, dass keine Zahlen bei der Eingabe erwartet werden. Es genügt z.B. das Kommando ye, um die Funktion cmd_ye aufzurufen, die die LED YELLOW umschaltet. (Gibt man trotzdem Zahlen ein, werden sie nicht beachtet.)

Als Antwort gibt Task SUI im seriellen Monitor generell das jeweilige Kommando aus. Für den Rest ist die Applikation zuständig. Im Beispiel-Tab SER1 wird mit der Funktion prtstate der Zustand der jeweiligen LED ausgegeben (on oder off).

6.2. SER2 Zahleneingabe

```
Tnum ye (100,0);
Tnum bl (50, 0 );    //50 ~ 0.50
Tnum gr (500,1000);

////////Task TELLOW      symmetric flashing of LED YELLOW
void tyellow1 () {
    WPIN(YELLOW, !RPIN(YELLOW) );
    os.mtdelay (ye.OA, tyellow1);
}

///////////Task TBLUE    symmetric flashing of LED BLUE
void tblue1 () {
  WPIN (BLUE, !RPIN(BLUE) );
  os.mtdelay ( bl.OA * 10 , tblue1);   //e.g.  50 * 10 = 500 ms = 0.5 s
}

/////Task TGREEN    asymmetric flashing of LED GREEN
void tgreen1 () {
  WPIN(GREEN, HIGH);
  os.mtdelay (gr.OA, tgreen2 );
}
void tgreen2 () {
  WPIN(GREEN, LOW);
  os.mtdelay (gr.OB, tgreen1);
}

const char yellow [] PROGMEM = "yellow i-  on/off ms";
const char blue   [] PROGMEM = "blue   f-  on/off x.xx s";
const char green  [] PROGMEM = "green  ii  on ms /off ms";
struct cmdstruct cmds [] = { {yellow, cmd_yellow},
                             {blue, cmd_blue},
                             {green, cmd_green}  };
char maxcmds = sizeof (cmds) / sizeof (cmdstruct) ;

void cmd_yellow() {
  switch (sr.inum) {
    case 0: ye.outp ( "$g$i ms$n");   //get task data
            break;
   default: if (IA <= 0)
               mprint (range_error);
            else {
               ye.OA = IA;           //set task data
               ye.outp ( "$s$i ms$n");
               os.stop (TYELLOW,0);          //Timer should be killed
               os.start (TYELLOW, tyellow1);
            }
  }
  os.mtcoop (sui_clear);
}

void cmd_blue() {
  switch (sr.inum) {
      case 0: bl.outp ( "$g$f s$n");   //get task data
              break;
     default: if (IA < 5 || IA > 500)    // ~  0.05 to 5 s
                mprint (range_error);
              else {
                bl.OA = IA;
```

```
                bl.outp ( "$s$f s$n");
              }
  }
  os.mtcoop (sui_clear);
}
void cmd_green() {
  switch (sr.inum) {
      case 0: gr.outp ( "$g$i ms $I ms$n");
              break;
      case 1: mprint (num_missed);
              break;
     default: if ( IA < 10  || IA > 3000 || IB < 10  || IB > 3000  )
                mprint (range_error);
              else {
                gr.OA = IA;
                gr.OB = IB;
                gr.outp ( "$s$i ms $I ms$n");
                eewrite ( gr.nb, sizeof(gr.nb), 0, sui_clear );  //store gre data to EEMEN
                return;      //subtask, return must follow !!
              }
  }
  os.mtcoop (sui_clear);
}
```

Die Tasks TYELLOW und TBLUE lassen ihre LED symmetrisch blinken und Task TGREEN asymmetrisch. Zu Speicherung der Zykluszeiten wird jeweils ein Objekt der Klasse Tnum benutzt. Der User kann jederzeit die Zykluszeiten durch einen Input im seriellen Monitor ändern.

Das Array cmds enthält neben dem Kommando yellow den Hinweis i-, der bedeutet, dass bei der Eingabe eine Integerzahl eingegeben werden kann oder auch nicht. Deshalb ist die Hauptaufgabe von Kommando-Funktion cmd_yellow, diese Zahl entgegenzunehmen, sie zu prüfen, zu speichern und dem User eine Antwort zu geben.

Mit sr.inum stellt Task SUI die Anzahl der eingegebenen Zahlen bereit. sr.inum = 0 (keine Zahl) wird von cmd_yellow als Aufforderung verstanden, die aktuell gespeicherte Zykluszeit auszugeben.

Die erste eingegebene Zahl wird von Task SUI in sr.lnum[0] bereitgestellt und die zweite in sr.inum[1], wofür die Kurzformen IA und IB definiert sind. Für die Zykluszeit von Task TYELLOW sind Zahlen >= 0 zugelassen. Eine negative Zahl führt zur Fehlermeldung durch die Funktion mprint, die Strings ausgeben kann, die im ROM gespeichert sind. Eine korrekte Zahl wird im Objekt ye gespeichert. Diese Zahl und damit die Zykluszeit kann sehr groß sein. Eine neu eingegebene Zykluszeit würde erst wirksam, wenn die alte abgelaufen ist. Um das zu vermeiden, wird Task TYELLOW gestoppt und sofort wieder gestartet.

Weil die in Objekten der Klasse Tnum gespeicherten Zahlen der Hauptbestandteil der Antwort sind, die der User erwartet, gibt es die Methode outp von Tnum. Mit nur einem Aufruf von outp ist es in den meisten Fällen möglich, die vollständige Antwort mit Hilfe eines Strings auszugeben, in den Formate eingebettet werden können. Ein Format besteht aus dem Zeichen $ und einem Kennbuchstaben.

```
$i   Ausgabe von OA als Integerzahl                  $I   dto OB
$f   Ausgabe von OA dezimal mit 2 Nachkommastellen   $F   dto OB
$t   Ausgabe von OA als Uhrzeit hhmmss               $T   dto OB
$g   Ausgabe von ">> "   (Symbol für get data)
$s   Ausgabe von "<< "   (Symbol für set data)
$n   Ausgabe von LineFeed
$$   Ausgabe von “$“
```

Beispielsweise enthält der Format-String “si ms$n“ drei der angegebenen Formate und die Zeichenfolge ‘ ms’, was Millisekunden bedeutet.

Im Unterschied zu Task TYELLOW wird die Zykluszeit von Task TBLUE als Dezimalzahl in Sekunden eingegeben, wobei nur Werte im Bereich von 0.05 bis 5 s erlaubt sind. Bei diesen relativ kurzen Zykluszeiten muss Task TBLUE nicht neu gestartet werden.

Task SUI kann zwar 2stellige Dezimalzahlen entgegennehmen, aber weiter behandelt werden sie als long-Zahlen mit dem 100fachen Wert der Dezimalzahl. Das heißt auch, Dezimalzahlen mit einer Stelle oder drei und mehr Stellen führen zu einer Fehlermeldung.

Der zulässige Bereich in Kommando-Funktion cmd_ blue ist dann 0.05*100 bis 5*100, also 5 bis 500 . Die Zykluszeit in Millisekunden ist das 10fache. Viele technische Rechnungen lassen sich auf diese Weise ausführen. Es ist nur folgerichtig, eine Zahl, die als Dezimalzahl eingegeben wurde, auch als Dezimalzahl auszugeben. Das erledigen die Formate $f und $F in Verbindung mit der Funktion outp,

Wenn eine Rechnung mit float-Zahlen ausgeführt werden muss, wird die Ganzzahl in IA mit (float) IA / 100 in eine float-Zahl verwandelt und weiter verarbeitet. (Für IB gilt das gleiche.) Auf die Funktion outp muss dann verzichtet und die Antwort anders gestaltet werden.

Task TGREEN blinkt asymmetrisch. Um den Zyklus korrekt zu ändern, müssen zwei Zahlen gleichzeitig eingegeben und wirksam werden. Die Kommando-Funktion cmd_green wertet es als Fehler, wenn nur *eine* Zahl eingegeben wurde. Die Bereichsprüfung ist aufwändiger.

Die Besonderheit ist, dass die aktuellen Daten dauerhaft im EEPROM gespeichert werden, damit zum Beispiel bei einem Reset durch Stromausfall die Daten nicht verloren sind. Das Schreiben eines Bytes in den EEPROM dauert etwa 4 ms. Weil darauf nicht herkömmlich gewartet werden kann, bietet mtos die Subtask eewrite. die den gesamten Schreibvorgang echtzeitverträglich erledigt. Sie hat vier Argumente und spiegelt einen RAM-Bereich im EEPROM mit Hilfe von Adressen.

```
eewrite ( gr.nb, sizeof(gr.nb), 0, sui_clear );
```

Bei dem Aufruf dieses Beispiels ist die Adresse des RAM-Bereichs der Zeiger auf das Array nb der Klasse Tnum. Der Name des Arrays ist bereits der Zeiger. Die Adresse des EEPROM-Bereichs ist die Zahl 0 (maximal 1023). Beide Bereiche haben die gleiche Größe, nämlich die des Arrays nb. Nicht zu vergessen ist der Funktionszeiger sui_clear zur Fortsetzung der Task.

Da *eewrite* eine Subtask ist, muss die aufrufende Task auf das Ende von eewrite warten, was Für Task SUI keine Bedeutung hat.

Gibt es mehrere EEPROM-Speichervorgänge in einem Projekt, muss der Programmierer die EEPROM-Adressen festlegen. Das erscheint mühsam, aber es verhindert, dass sich während der Programmentwicklung die Adressen ändern. Das könnte geschehen, wenn der Compiler mit dem Zusatz EEMEM bei der Deklaration die Adressen bestimmt.

Bei jedem Reset müssen die Daten aus dem EEPROM gelesen und in den RAM gespeichert werden. Das geschieht mit der Funktion eeread, die echtzeit-verträglich ist. eeread benutzt die gleichen Adressen wie der korrespondierende Aufruf von eewrite, nur vertauscht.

```
eeread (0 , sizeof(gr.nb) , gr.nb );
```

Die Funktion eeread wird in der nicht dargestellten Allzweck-Task GPCYCLE aufgerufen.

Weder eeread noch eewrite dürfen aufgerufen werden, wenn eewrite arbeitet. Wenn diese Situation nicht ausgeschlossen werden kann, muss die für diesen Zweck bereitgestellte Semaphore EESEM mit mtsema (EESEM,....) für einen Ausschluss genutzt werden.

6.3. SER3 System Monitor

```
Tnum idl (NONB,0);  //idle session data

const char idle [] PROGMEM =    "idle  --  CPU";
const char runti   [] PROGMEM = "runti --  system run time";
const char clock   [] PROGMEM = "clock t-  get/set clock";
struct cmdstruct cmds [] = { { idle, cmd_idle},
                             {runti, cmd_runti},
                             {clock, cmd_clock},  };
char maxcmds = sizeof (cmds) / sizeof (cmdstruct) ;

void cmd_idle() {      //Can find minimal idle (~ max cpu load) during a session
    if (idl.OA == NONB) {        //begin of session
      idl.OA = os.idle;          //actual idle
      idl.OB = os.bclock;        //actual time
      idl.outp ("$i begin $T$n");
      os.mtdelay (1100, cmd_idle1);  //influence of idl.outp should be avoided
      return;
    }
    else {                           //end of session
      idl.outp ( "$i $T end ");      //result
      idl.OA = os.bclock;
      idl.outp ( "$t$n");
      idl.OA = NONB;                 //stop
    }
    os.mtcoop (sui_clear);
}
void cmd_idle1() {                   //real start of idle session
  idl.OA = 10000;                    //see gpcycle2
  os.mtcoop (sui_clear);
}

void cmd_runti() { //get program running time
  PR ( (float) os.get_clock() / 3600000 );
  PRln (" h");
  os.mtcoop (sui_clear);
}

void cmd_clock() {        //adjust base clock manually or
  switch (sr.inum) {      //let synchronize base clock by PC cronjob,
    case 0: PR (">> ");   //see following description
            prtime (os.bclock);
            break;
    default: if ( trtime (IA) != SECDAY ) {
               os.bclock = trtime (IA);
             }
  }
  PRln ();
  os.mtcoop (sui_clear);
}

////////Task GPCYCLE    general purpose
void gpcycle1 () {      //to add own and other inits
```

```
  PR ("SER3 Hello, ");
  mprint (LF_menu);
  os.mtcoop (gpcycle2);
}

void gpcycle2 () {     //general purpose cycle
  WPIN(LED_BUILTIN, !RPIN(LED_BUILTIN));    //toggled LED as sign of life,
  if (idl.OA != NONB) {    ////idle session
    if ( os.idle < idl.OA ) {
      idl.OA = os.idle;
      idl.OB = os.bclock;
    }
  }
  os.mtdelay (1000, gpcycle2);
}
```

In Tab SER3 geht es um die Systemüberwachung und die damit verbundene User-Kommunikation.

Mit dem Kommando idle wird eine idle session gestartet und gestoppt. Währenddessen wird jede Sekunde der aktuelle Wert von os.idle (CPU-Leerlauf) geprüft und, falls er minimal ist, als neues Minimum zusammen mit der Uhrzeit gespeichert. Das erledigt die Taskfunktion gpcycle2 von Allzweck-Task GPCYCLE. Auf diese Weise wird die Spitzenlast (gleich minimaler Leerlauf) während der session festgehalten, zum Beispiel wenn während der session das Kommando runti oder LF eingegeben wird.

Eine Zahleneingabe wird vom User nicht verlangt. Dennoch braucht der Vorgang Daten, die im Objekt idl von Klasse Tnum gespeichert sind. Insbesondere wird die Zahl NONB verwendet (kleinste long-Zahl), die für Ausnahmesituationen eingesetzt werden kann. Mit NONB wird signalisiert, dass die idle session *nicht* läuft.

Beim Stopp der session werden die Ergebnisse ausgegeben.

Mit dem Kommando runti wird die Laufzeit des Programms in Stunden ausgegeben. Dazu wird der Millisekunden-Zähler clock des Systems benutzt, der von os.get_clock zurückgegeben wird. Beim overflow wird clock negativ, was bei der üblichen Verwendung von clock nicht stört, aber bei der Laufzeitberechnung ist eine Anpassung erforderlich. Mit dieser einfachen Methode beginnt die Zählung nachi etwa 596 Stunden wieder bei Null. Allerdings ist der Wert sehr ungenau, weil der Taktgeber des Nano kein Quarz ist.

Das System stellt eine einfache Uhr in Form des Zählers os.bclock bereit, der Sekunden von 0 bis 86399 umlaufend zählt (ein Tag). Die (Zeit)zahl in os.bclock, gibt die Zahl der Sekunden an, die seit 0 Uhr vergangen sind, falls die Uhr gestellt wurde,.

Kommando-Funktion cmd_clock kann die Uhrzeit abfragen und die Uhr stellen, und zwar im Format hhmmss, das einer long-Zahl entspricht. Bei der Verarbeitung der Eingabe muss diese Zahl mit der Funktion trtime in die Zeitzahl os.bclock umgewandelt werden.

Zur Ausgabe der Uhrzeit wird die Funktion prtime benutzt und auf eine Antwort verzichtet. Alternativ könnte die Uhrzeit mit der Funktion outp und den Formaten $t und $T ausgegeben werden, wenn zur Speicherung der Zeitzahl ein Objekt von Tnum benutzt wird.

Man erhält eine genau gehende Uhr, die nicht gestellt werden muss, wenn Nano in Verbindung mit einem Raspberry Pi standalone (ohne Arduino) betrieben wird. (Ein Linux-PC zum Testen geht auch, wenn Arduino beendet wird.) Wichtig ist ein 100nF Kondensator zwischen Nano Pin RST und VCC , der Resets verhindert, wenn die serielle

Schnittstelle von Raspi geöffnet wird. Zum **Hochladen** von Sketchen unter Arduino muss er **entfernt** werden.

Der Ordner doc enthält zwei Python Scripte.

Python Script cronclock.py, wird als cronjob auf Raspi jede Minute gestartet, wodurch die Uhrzeit auf Nano synchronisiert wird, so als würde ein User die Uhr stellen. Ein cronjob wird eingerichtet mit dem Aufruf crontab -e auf der Kommandozeile. Im Editor, der sich öffnet, wird die Zeile * * * * * <abs.Pfad>/cronclock.py zur ebenfalls geöffneten Datei hinzugefügt.

Der serielle Monitor von Arduino wird ersetzt durch das Python Script mtcomm.py, das ein zeilenorientiertes Terminalprogramm ist und auf der Kommandozeile von Raspi gestartet wird. Auf diese Weise kann über LAN und sogar aus dem Internet mit Nano gearbeitet werden.

Die Python Scripte müssen zu ausführbaren Dateien gemacht werden. Am Code von SER3 ändert sich nichts.

6.4. SER4 Tagesschaltuhr mit drei Zeitbereichen

```
#define  TIMER  TYELLOW     //Rename task TYELLOW

class Tinum : public Tnum {
  public:
  Tinum  (long nb0, long nb1): Tnum (nb0, nb1)  {
  }
  bool dotinum () {
    if (os.get_taskstate (TIMER) == WAITING_S )
      PRln ("clock must be set");
    else
    switch (sr.inum ) {
      case 0:
        outp ("$g$t $T$n");         //get range
        break;
      case 1:
        mprint (num_missed);
        break;
      default:
        if (IA != NONB )  {
             IA = trtime(IA);
             if (IA != SECDAY )
               OA = IA;
             else {
               mprint (range_error);
               break;
             }
        }
        if (IB != NONB )  {
             IB = trtime(IB);
             if (IB != SECDAY )
               OB =  IB;
             else {
               mprint (range_error);
               break;
             }
        }
        outp ("$s$t $T$n");
        return true;
    }
    return false;
```

```
  }
};

//time channels "from","to"
Tinum ti1  (0,0);
Tinum ti2  (0,0);
Tinum ti3  (0,0);

/////////Task TIMER
void timer1() {
  os.mtsema (GPSEMA, timer2);      //wait for base clock is set
}
void timer2() {
  if (os.timeinrange (ti1.OA,ti1.OB)  ||
        os.timeinrange (ti2.OA,ti2.OB)  ||
            os.timeinrange (ti3.OA,ti3.OB) )
    WPIN (YELLOW,HIGH);
  else WPIN (YELLOW,LOW);
  os.mtdelay (1000,timer2);
}
const char cti1   [] PROGMEM = "ti1   tt from to";
const char cti2   [] PROGMEM = "ti2   tt from to";
const char cti3   [] PROGMEM = "ti3   tt from to";
const char clock [] PROGMEM = "clock t- set/get clock";
struct cmdstruct cmds [] = { {cti1, cmd_ti1},
                             {cti2, cmd_ti2},
                             {cti3, cmd_ti3},
                             {clock, cmd_clock}   };
char maxcmds = sizeof (cmds) / sizeof (cmdstruct) ;

void cmd_ti1 () {
  if (ti1.dotinum())
    eewrite (ti1.nb, sizeof(ti1.nb) ,8 , sui_clear);
  else os.mtcoop (sui_clear);
}
void cmd_ti2 () {
  if (ti2.dotinum() )
    eewrite (ti2.nb, sizeof(ti2.nb) ,16 , sui_clear);
  else os.mtcoop (sui_clear);
}
void cmd_ti3 () {
  if (ti3.dotinum())
    eewrite (ti3.nb, sizeof(ti3.nb) ,24 , sui_clear);
  else os.mtcoop (sui_clear);
}

void cmd_clock() {     //adjust base clock manually or
  switch (sr.inum) {   //let synchronize base clock by PC cronjob, see SER3
    case 0: PR (">> ");
            prtime (os.bclock);
            break;
    default: if ( trtime (IA) != SECDAY ) {
               os.bclock = trtime (IA);
               os.signal (GPSEMA);     //signale: base clock is set
             }
  }
  PRln ();
  os.mtcoop (sui_clear);
}
```

Das Herz der Schaltuhr ist Task TIMER, das ist die umbenannte Task TYELLOW. Task Timer prüft jede Sekunde, ob die aktuelle Uhrzeit in einem der drei gespeicherten Zeitbereiche liegt. Falls ja, wird LED YELLOW eingeschaltet, andernfalls ausgeschaltet. Die Funktion os.timeinrange kann auch Zeitbereiche prüfen, in denen die Zeit 0 Uhr liegt. Task Timer arbeitet erst, wenn die Systemuhr gestellt wurde, was durch die Allzweck-Semaphore GPSEMA signalisiert wird.

Die drei Zeitbereiche sind in Objekten von Klasse Tnum bzw. Tinum gespeichert. Die User-Eingabe dieser Zeiten wird entsprechend von drei Kommando-Funktionen abgewickelt. Weil die Verarbeitung der drei User-Eingaben gleich ist, ist die neue Klasse Tinum zweckmäßig, die von Tnum erbt, und die Methode dotinum zur Behandlung der User-Eingaben bekommt. Allerdings kann eine Methode einer Klasse keine mt-Funktionen und keine Subtasks aufrufen. Die Speicherung der Zeitbereiche im EEPROM muss daher in den Kommando-Funktionen erfolgen.

Die Methode dotinum nimmt keine User-Eingaben an, wenn die Uhr noch nicht gestellt wurde, was als Erinnerung anzusehen ist. Wenn eine der beiden Zeiten des Zeitbereichs unverändert bleiben soll, kann dafür das Zeichen # (intern Zahl NONB) eingegeben werden.

Die Kommando-Funktion cmd_clock setzt die Semaphore GPSEMA, um zu signalisieren, dass die Uhr gestellt wurde.

6.5. SER5 Treppenlicht-Timer

```
Tnum al (5000, 0);

////see SER1
const char stli [] PROGMEM = "stli i- time ms";
struct cmdstruct cmds [] = { {stli, cmd_stli},
                           };
char maxcmds = sizeof (cmds) / sizeof (cmdstruct) ;

void cmd_stli () {
  switch (sr.inum) {
    case 0: al.outp ("$g $i ms$n");
            break;
    default:  if (IA > 10000 || IA < 0) {
                mprint (range_error);
                break;
              }
              al.OA = IA;
              al.outp ("$s $i ms$n");
  }
  os.mtcoop (sui_clear);
}

#define STLIGHT TYELLOW     //Rename TYELLOW

////////Task STLIGHT
void stlight1 () {
  os.mtpin (RBUTTON, LOW, stlight2);  //Trigger
}
void stlight2 () {
  WPIN(YELLOW, HIGH);
  os.mtdelay (al.OA, stlight3);      //Retrigger per RBUTTON timeout
  os.mtpin (RBUTTON, LOW, stlight2);
}
```

```
void stlight3 () {
  WPIN(YELLOW, LOW);
  os.mtcoop (stlight1);
}

////////Task GPCYCLE      general purpose
void gpcycle1 () {         //to add own and other inits
  PR ("SER5 Hello, ");
  mprint (LF_menu);
  os.start (STLIGHT,stlight1);
  os.mtcoop ( gpcycle2);
}

void  gpcycle2 () {      //general purpose cycle
  WPIN(LED_BUILTIN, !RPIN(LED_BUILTIN));  //toggled LED as sign of life
  os.mtdelay (1000, gpcycle2);
}
```

Task STLIGHT ist die umbenannte Task TYELLOW, die einen Treppenlicht-Timer realisiert. Nach einem Druck auf den Taster an Pin RBUTTON wird in Taskfunktion stlight2 LED YELLOW eingeschaltet. Es beginnt ein Zeitablauf, der zu Taskfunktion stlight3 führt, die die LED abschaltet, um dann in Taskfunktion stlight1 auf einen erneuten Tastendruck zu warten.

Das Besondere ist, dass während des Zeitablaufs ein weiterer Tastendruck erfolgen kann, der den Zeitablauf neu startet, wonach wieder ein Tastendruck erkannt werden kann usw. Diese Retrigger-Funktion ist bei jedem Treppenlicht-Timer zu finden und entspricht programmtechnisch einem mtos-Timeout.

Die Einschaltdauer kann durch den User in gewissen Grenzen verändert werden, was ein weiteres Beispiel des seriellen User-Inputs ist, aber keinen neuen Aspekt bietet.

Die bekannte Totmann-Schaltung könnte genauso realisiert werden, nur wäre die eingeschaltete LED der Normalfall. Das Verlöschen der LED, würde in einen Alarm umgesetzt, oder man vertauscht die LED-Zustände.

7. Beispiel-Sketch *fun*

Dieser Sketch realisiert drei völlig unterschiedliche Vorgänge, die gleichzeitig ablaufen können und ein bisschen Spaß machen sollen.

Die Behandlung der User-Eingaben (programmtechnisch: Vervollständigung von Task SUI) erfolgt in einem eigenen Tab zcomm. Außerdem befindet sich Task GPCYCLE dort. Es ist zweckmäßig, wenn dieser Tab als letzter compiliert wird, weshalb der Name mit z beginnt.

Tab reac enthält einen Reaktionstester in Gestalt von Task REAC, deren Taskfunktion reac2 eine endlose mt-Schleife ist, die die LED BLUE blinken lässt. Durch Drücken des Tasters an Pin RBUTTON wird die LED ausgeschaltet, und der Reaktionstest beginnt. Nach einer zufälligen Zeit wird die LED eingeschaltet und der User muss so schnell wie möglich den Taster drücken. Die Zeit, die der dafür braucht, also seine Reaktionszeit wird gemessen und ausgegeben. Der Reaktionstester kommt ohne User-Eingabe am seriellen Monitor aus.

Task PLAY in Tab play spielt eines von drei Musikstücken. Wenn der User den jeweiligen Kurznamen als Kommando im seriellen Monitor eingibt, startet die Task. Mit dem Kommando stop kann der Song abgebrochen werden.

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 eines Zeigers, der von der jeweiligen Kommando-Funktion gesetzt wird. Für die Daten und Methoden von Task PLAY ist die Klasse Tnum nicht geeignet, weshalb es die Klasse Song gibt.

Task PRIME in Tab prime berechnet fortgesetzt Primzahlen ab 1000000001. Die Task wird von User mit dem Kommando prime gestartet und gestoppt. Beim jedem Start setzt Task PRIME die Berechnung an der Stelle fort, an der der Stopp erfolgte. Mit dem Kommando prime< beginnt die Berechnung ab dem Anfangswert.

Es geht bei Task PRIME um die Demonstration der CPU-Volllast, wofür sich die Primzahlberechnung gut eignet. Zur Kontrolle erscheinen die Ergebnisse im seriellen Monitor. 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.
Mit https://www.arndt-bruenner.de/mathe/scripts/primzahlen.htm können die Ergebnisse kontrolliert werden.

Der User kann die Vorgänge so starten, dass sie gleichzeitig laufen, und er wird sehen bzw. hören, dass sie sich untereinander nicht beeinflussen. Dazu kommt, dass Task PRIME mit der Langzeitrechnung ohne Ereignisse die CPU vollständig auslastet. Der Leerlauf (idle) ist 0, was mit dem Kommando idle kontrolliert werden kann.

8. Zum Schluss

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