CoopOS Part-1 Was wird geboten ?

- Warum diese Serie ?
- Warum kooperatives Multitasking ?
- Nicht nur vorgegebene Bibliothek, sondern Verständnis
- Beispiele zur Gestaltung von eigenem Code
- Sehr anschauliche einfache Beispiele

Zunächst PPT-Vortrag
Dann Entwicklung kleiner eingestreuter Demos für Arduino-IDE
Mit Sourcen.

Am Ende der Video-Reihe:

Komplettes CoopOS: Simpel, Schnell, Platzsparend
Leicht anpassbar!

CoopOS Part-1 Was wird geboten ?

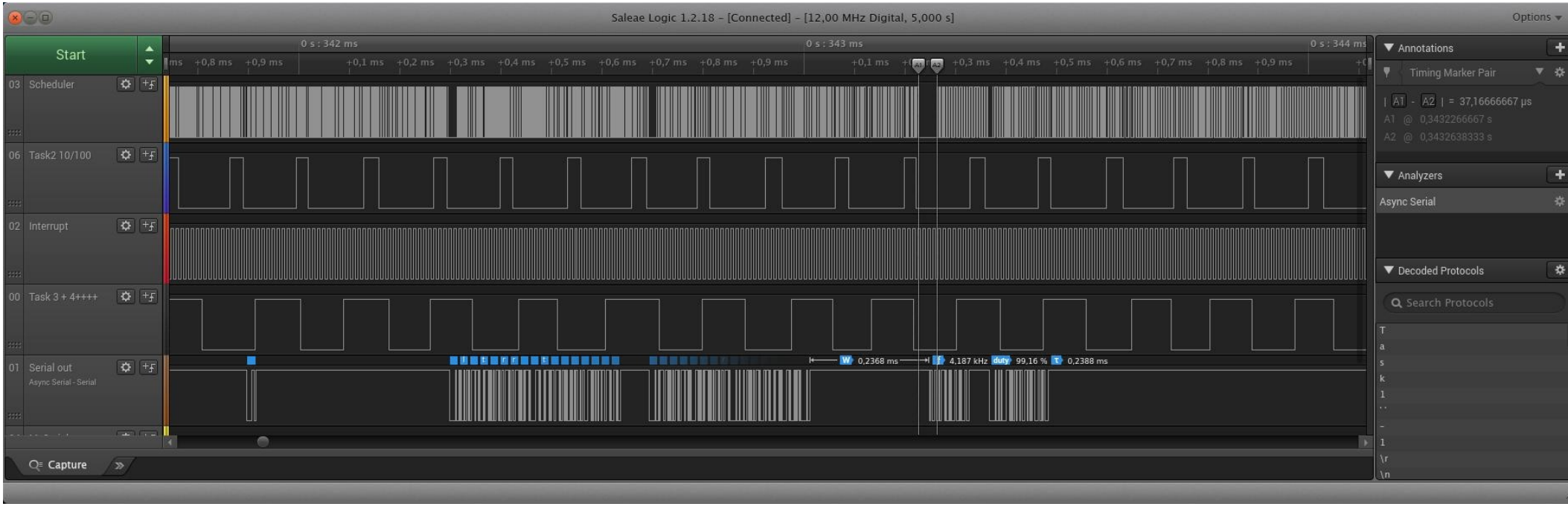
Benutzt wird hier ein **ESP8266**, aber der Kurs Demo-0 bis Demo-6 sollten auf allen Prozessoren ab **AtTiny45** funktionieren.
Am Ende steht eine Demo (Esp8266), mit:
(pro Sekunde !)

- **430000 Scheduler-calls/s** (alle 2.3 µs im Schnitt !)
- **100000 Timer Interrupts/s** (alle 10 µs, 5 µs geht auch !)
- Task-Aufruf vom Interrupt (per Signal <10 µs)
- Signale, Warten auf Signale, Mutex für Resourcen, ..
- CoopOS.h mit CoopOS Class: **< 200 Zeilen**

Es fängt für den Profi vielleicht langweilig an – aber dann … ;)

CoopOS Part-1 Verständnis

Ich möchte, dass solche Bilder verstanden werden:



CoopOS

Was sind Coroutines ?

Coroutines: Begriff von Melvin Conway 1963

Donald Knuth: Funktionen sind Sonderfall von Coroutines 1967
(„*The Art of Programming*“)

In vielen Programmiersprachen implementiert oder als Bibliothek vorhanden.

Was sind Coroutines ?

In der Informatik sind Koroutinen (auch Coroutines) eine Verallgemeinerung des Konzepts einer Prozedur oder Funktion. Der prinzipielle Unterschied zwischen Koroutinen und Prozeduren ist, dass Koroutinen **ihren Ablauf unterbrechen und später wieder aufnehmen können, wobei sie ihren Status beibehalten.** (Wikipedia)

Was sind Coroutines ?

Jede Coroutine ist eine „state machine“. Sie speichert ihren Status beim freiwilligen Verlassen, um diesen Status beim nächsten Aufruf wieder anzunehmen.
So werden bei jedem Aufruf andere Teile der Funktion ausgeführt.

Was sind Coroutines ?

Was sind Coroutines ?

Coroutines sind eher eine Philosophie statt einer Library. CoopOS liefert dazu den Scheduler, um die Coroutines zu verwalten. Sie laufen vom AtTiny bis hin zum schnellen PC. Sie sind auch kein Gegensatz zu einem RTOS, sondern oft eine gute Ergänzung.
Beispiel: Der ESP32 hat zwei Kerne. Damit ist es möglich, freeRTOS auf einem Kern laufen zu lassen und CoopOS auf dem 2. Kern. Damit sind erstaunliche Ergebnisse zu erzielen! (Dazu mehr in einem anderen Projekt)
Aber auch auf einem Linux_preempt mit einem reservierten Prozessor-Kern sind mit CoopOS extrem schnelle Taskswitches möglich – z. Bsp. für die Zusammenarbeit mit einem externen Prozessor.

Was sind Coroutines ?

Mit einem Rasperry Pi 3+ habe ich es mal auf die Spitze getrieben. Kein Linux, „bare metal“.
Nur 1 Kern wurde benutzt – 3 stehen noch zur Verfügung. Werte mit CoopOS

- **> 3.000.000 Interrupts/s**
- **10.000.000 Taskswitches/s**

http://helmutweber.de/Microcontroller.html

CoopOS

```
// Beispiel für eine kooperative Funktion


int F1_State=0;

void Task1() {
    Switch (F1_State) {
        case (0):
                printf(„Task1 – Part1\n“);
                F1_State=1;                     // Zustand beim nächsten Aufruf
                return;                         // kooperatives Verlassen
        case (1):
                printf(„Task1 – Part2\n“);
                F1_State=0;
                return;
    }
}
```

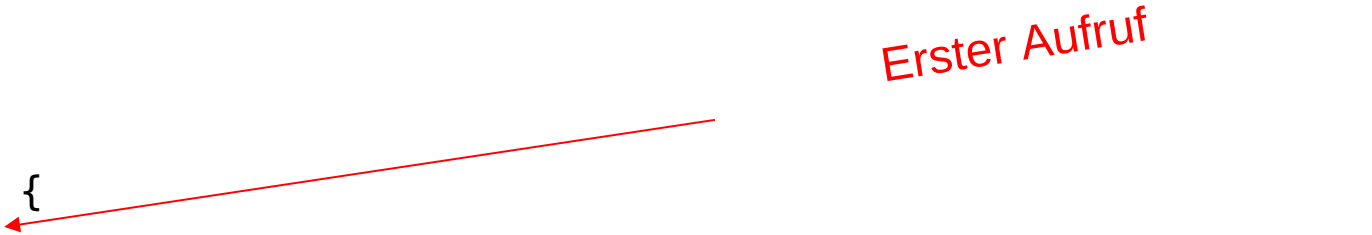
Vor erstem Aufruf

CoopOS

```
// Beispiel für eine kooperative Funktion


int F1_State=0;

void Task1() {
    Switch (F1_State) {
        case (0):
                printf(„Task1 – Part1\n“);
                F1_State=1;                    // Zustand beim nächsten Aufruf
                return;                        // kooperatives Verlassen
        case (1):
                printf(„Task1 – Part2\n“);
                F1_State=0;
                return;
    }
}
```

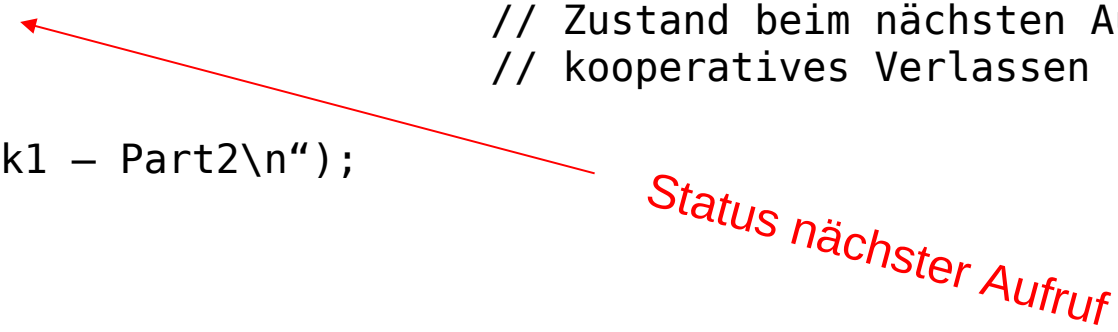


CoopOS

```
// Beispiel für eine kooperative Funktion


int F1_State=0;

void Task1() {
    Switch (F1_State) {
        case (0):
                printf(„Task1 – Part1\n“);
                F1_State=1;                    // Zustand beim nächsten Aufruf
                return;                        // kooperatives Verlassen
        case (1):
                printf(„Task1 – Part2\n“);
                F1_State=0;
                return;
    }
}
```

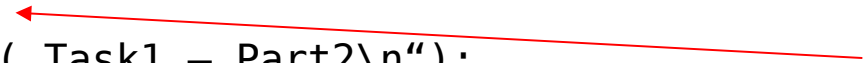


CoopOS

```
// Beispiel für eine kooperative Funktion


int F1_State=0;

void Task1() {
    Switch (F1_State) {
        case (0):
                printf(„Task1 – Part1\n“);
                F1_State=1;                    // Zustand beim nächsten Aufruf
                return;                        // kooperatives Verlassen
        case (1):
                printf(„Task1 – Part2\n“);
                F1_State=0;
                return;
    }
}
```



Zweiter Aufruf

CoopOS

```
// Beispiel für eine kooperative Funktion


int F1_State=0;

void Task1() {
    Switch (F1_State) {
        case (0):
                printf(„Task1 – Part1\n“);
                F1_State=1;                    // Zustand beim nächsten Aufruf
                return;                        // kooperatives Verlassen
        case (1):
                printf(„Task1 – Part2\n“);
                F1_State=0;
                return;
    }
}
```

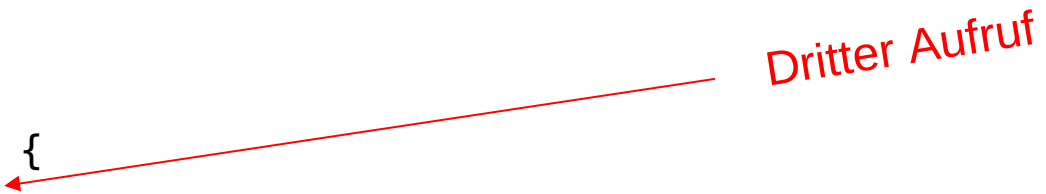
Status nächster Aufruf

CoopOS

```
// Beispiel für eine kooperative Funktion


int F1_State=0;

void Task1() {
    Switch (F1_State) {
        case (0):
                printf(„Task1 – Part1\n“);
                F1_State=1;                    // Zustand beim nächsten Aufruf
                return;                        // kooperatives Verlassen
        case (1):
                printf(„Task1 – Part2\n“);
                F1_State=0;
                return;
    }
}
```



CoopOS

```
// Beispiel für eine kooperative Funktion


int F1_State=0;

void Task1() {
    Switch (F1_State) {
        case (0):
                printf(„Task1 – Part1\n“);
                F1_State=1;                          // Zustand beim nächsten Aufruf
                return;                              // kooperatives Verlassen
        case (1):
                printf(„Task1 – Part2\n“);
                F1_State=0;
                return;
    }
}
```

Warum diese Unterbrechungen?

CoopOS

```
// Beispiel für eine kooperative Funktion

void Task1() {
static int F1_State=0;

Switch (F1_State) {
        case (0):
                printf(„Task1 – Part1\n“);
                F1_State=1;
                return;
        case (1):
                printf(„Task1 – Part2\n“);
                F1_State=0;
                return;
    }
}
```

static: Erfüllt den gleichen Zweck
Wie GLOBAL

```
// Zustand beim nächsten Aufruf
// kooperatives Verlassen
```

Um jedesmal nur einen kleinen Teil der Funktion **sehr schnell** abzuarbeiten.

Beispiel Arduino Kurs-0

Bei diesen Hinweisen bitte die entsprechenden Sourcen laden, ansehen und ausprobieren, ändern ...

Cooperative Multitasking

- (**Zunächst**) keine zeitliche Steuerung
- Tasks bestimmen selbst, wann sie kooperativ sind
- Genauer Punkt (im Programm) der Unterbrechung vorgegeben
- Problemloser Zugriff auf (globale) Variablen
- Alle vorhandenen Bibliotheken nutzbar
- Nur **ein Stack** (Speicher sparend)
- Schnelles Taskswitching

CoopOS

Cooperative / Preemptive

Preemptive Multitasking (RTOS)

- Tasks werden per Timer-Interrupt unterbrochen
- Scheduler schaltet zwischen Tasks um
- Genauer ZEIT-Punkt der Unterbrechung vorgegeben
- Zugriff auf (globale) Variablen muss geregelt werden
- Teilweise neue Bibliotheken notwendig
- Jeder Task hat eigenen Stack (Speicherintensiv)
- Taskumschaltung rettet Register ==> kostet Zeit

Was sind Coroutines ?

Beispiel Arduino Kurs-1

CoopOS Part2 Was wird geboten ?

- Warum diese Serie ?
- Warum kooperatives Multitasking ?
- Nicht nur vorgegebene Bibliothek, sondern Verständnis
- Beispiele zur Gestaltung von eigenem Code
- Sehr anschauliche einfache Beispiele

Zunächst PPT-Vortrag
Entwicklung von TaskControlBlock (TCB) und Scheduler

Am Ende der Video-Reihe:

Komplettes CoopOS: Simpel, Schnell, Platzsparend
Leicht anpassbar!

TaskControlBlock TCB

Der TCB enthält alle Parameter, die einen Task definieren

- Task-Funktion
- Status vom Task (READY, DELAYED)
- Zeitpunkt des letzten Aufrufs
- Delay (Pause bis zum nächsten Aufruf)

Schnelles Taskswitching – aber **WANN** ?

TaskControlBlock TCB

Definieren einer Struktur:

```
struct tcb {
  void          (*Func)();          // Name des Tasks
  uint32_t      LastCalled;         // Zeit: letzter Aufruf
  uint32_t      Delay;              // Verzögerung für Task
  char          State;              // Status: READY, DELAYED
};

#define MAXTASKS 10
int      numTasks;                  // Anzahl der definierten Tasks

struct tcb    Tasks[MAXTASKS];     // Array von TCBs

enum state  { READY, DELAYED };
```

Scheduler

Zeitliche Steuerung:

```
void Scheduler() {
uint32_t m;
    for (int i=0; i<numTasks; i++) {
        m=micros();
        if ((m-Tasks[i].LastCalled) >= Tasks[i].Delay) {
          Tasks[i].State = READY;
        }
        else Tasks[i].State=DELAYED;

        if (Tasks[i].State == READY) {
            Tasks[i].Func();              // Aufruf vom Task
            Tasks[i].LastCalled = m;
        }
    } // for
} // Scheduler
```

CoopOS defines

Scheduler

Für bessere Lesbarkeit:

```
#define taskBegin() static int _mark = 0; \
  switch (_mark) {\
  case 0:

#define taskEnd()    _mark=0;\
    return;  \
}

#define taskSwitch() _mark=__LINE__; \
return;  case __LINE__:
```

CoopOS

Verwendung der **defines**

Für bessere Lesbarkeit:

```
void Task1() {
    taskBegin();
    while(1) {
      Serial.println("Task1 -1");
      taskSwitch();
      Serial.println("Task1 -2");
      taskSwitch();
      Serial.println("Task1 -3");
      taskSwitch();
      Serial.println("Task1 -4");
    }
    taskEnd();
}
```

Was sind Coroutines ?

Beispiel Arduino Kurs-2

CoopOS Part3 Zeitliche Steuerung

Es wird Zeit, für jeden Task eine unterschiedliche zeitliche Steuerung einzuführen.

Zunächst PPT-Vortrag

- **thisTask** – globaler Pointer auf aktuellen Task
- **#define taskDelay(microseconds)**
- **taskDelay(val) in Tasks**

Am Ende der Video-Reihe:

Komplettes CoopOS: Simpel, Schnell, Platzsparend
Leicht anpassbar!

Scheduler

thisTask = globaler Pointer auf Task:

```
struct tcb      *thisTask;

void Scheduler() {
uint32_t        m=micros();
    for (int i=0; i<numTasks; i++) {
        thisTask = &Tasks[i];
        if ((thisTask->LastCalled) >= thisTask->Delay) {
          thisTask->State = READY;
        }
        else thisTask->State=DELAYED;

        if (thisTask->State == READY) {
            thisTask->Func();
            thisTask->LastCalled = m;
        }
    } // for
} // Scheduler
```

taskDelay

thisTask = Pointer auf Task:

```
#define taskSwitch() _mark=__LINE__; return;  case __LINE__:

extern struct tcb *thisTask;

#define taskDelay(val) _mark=__LINE__; return; \
thisTask→Delay =val; thisTask→State=DELAYED; case __LINE__:
```

CoopOS

taskDelay & Scheduler

Beispiel Arduino Kurs-3

Interpretation der Ergebnisse

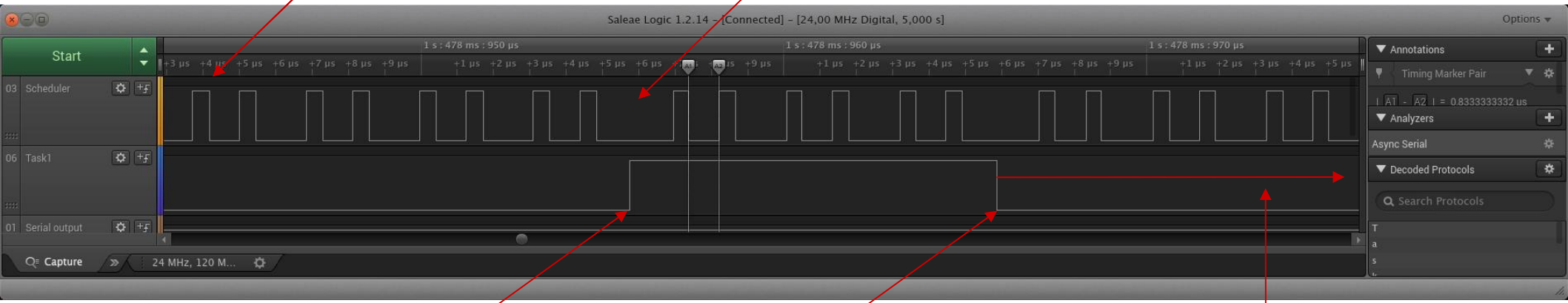
Taskswitch-Zeiten: (ESP8266, 160 MHz)

- Kein Task READY: 0.83 µs
- Task1 READY und Ausführung: 2,51 µs

CoopOS ESP8266 – 160 MHz

Leerlauf

Task1 wird ausgeführt



Task1: 10.125 µs HIGH

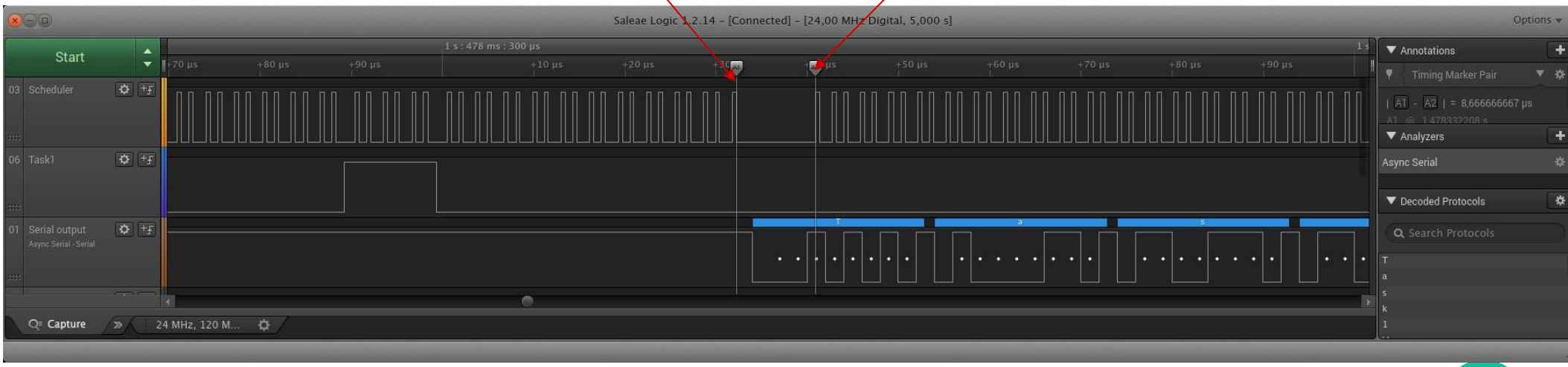
Task1: 100.71 µs LOW

CoopOS ESP8266 – 160 MHz

Ein Kritikpunkt bei kooperativen Systemen ist:
Die längste Ausführungszeit eines Tasks ist stets einzukalkulieren.
Und das ist wahr: Hier die serielle Ausgabe mit ~9µs !
ABER:
Damit finden nach HARD-REALTIME Bedingungen immer noch mindestens alle 10 µs ein TaskSwitch statt.

Eine Prioritäten-Steuerung steht noch aus!

8.67 µs



CoopOS Part4 Was wird geboten ?

Zunächst PPT-Vortrag

Aufräumen
TaskInit(..);

Am Ende der Video-Reihe:

Komplettes CoopOS: Simpel, Schnell, Platzsparend
Leicht anpassbar!

CoopOS

CoopOS.h, TaskInit(..)

Aufräumen:

- Sourcen für das Multitasking wurden ausgelagert → **CoopOS.h**
- Keine Library sondern nur eine #include-Datei mit

weniger als 100 Zeilen

- Ein **TaskInit(...)** wurde zugefügt, um die TCBs am Anfang zu initialisieren.

- Das Hauptprogramm wird extrem kurz und übersichtlich

CoopOS

taskDelay & Scheduler

Beispiel Arduino Kurs-4

CoopOS Part5 Was wird geboten ?

Zunächst PPT-Vortrag

- **TaskHandle**
- **taskStop(id);**
- **taskStopMe();**
- **taskResume(id);**

Priorität von Tasks

Am Ende der Video-Reihe:

Komplettes CoopOS: Simpel, Schnell, Platzsparend
Leicht anpassbar!

CoopOS

TaskHandle, taskStop, taskResume

TaskHandle:
TaskInit(..) liefert einen TaskHandle zurück, mit dem andere Tasks Veränderungen durchführen können.

taskStop(id):
Es wurde der State „**BLOCKED**“ neu eingeführt.
taskStop(id) stoppt den (fremden) Task[id].
Er kann sich nicht selbst wieder zum Leben erwecken!

taskStopMe():
taskStopMe() stoppt denTask, der das benutzt, sofort.
Er kann sich nicht selbst wieder zum Leben erwecken!
Aber ein anderer Task kann ihn mit

taskResume(id)
wieder starten.

CoopOS Priorität

Priorität:
Obwohl im tcb das Feld Priority zugefügt wurde, wird es bisher (und bis auf Weiteres) nicht benutzt.
Der Scheduler durchläuft nach jedem Aufruf alle vorhanden Tasks.
Eine Priorität gibt es nicht (Round Robin).

Task3 könnte Task20 mit taskResume(id) wieder aktivieren.
Aber die Auswirkung wäre erst nach Task4-19 wirksam.
Im config-Teil von CoopOS.h kann
INTERNAL_PRIO zugeschaltet werden und das bedeuted:
Jedesmal, Wenn der Scheduler einen Task gestartet hat wird der Scheduler verlassen (return). Beim nächsten Start sucht der Scheduler wieder ab Task0.
Tasks, die zuerst initialisiert wurden, bekommen so eine höhere Priorität.

CoopOS taskResume

Benutzung:
Betrachten wir in der Demo Task3:

```
void Task3() {
  taskBegin();
  while(1) {
    digitalWrite(12,HIGH);
    taskStopMe();                // ich geh ins Bett ;)
  }
  taskEnd();
}
```

Task3 setzt Pin 12 auf HIGH und legt sich dann „schlafen“.
Sinnlos? Nur in Zusammenarbeit mit Task4 kann das verstanden werden.

CoopOS taskResume

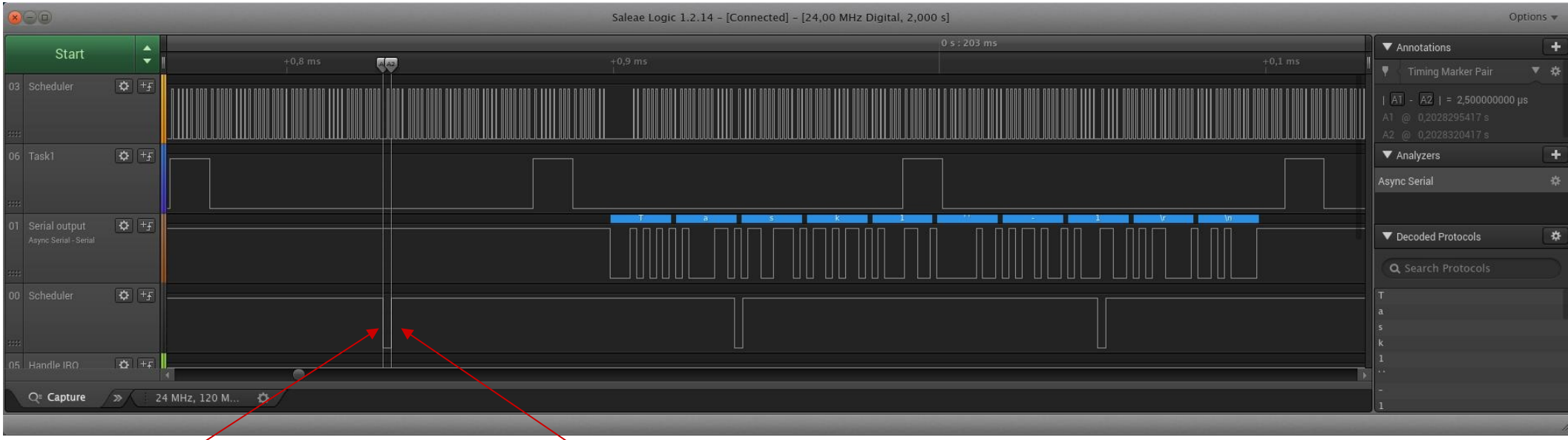
Benutzung:
Betrachten wir in der Demo Task4

```
void Task4() {
  taskBegin();
  while(1) {
    taskDelay(100);
    digitalWrite(12,LOW);
    taskResume(T_ID3); // reagiert "normalerweise" nach 2,5µs (Init: Platz 1
  }
  taskEnd();
}
```

Task4 setzt Pin 12 auf LOW, wartet 100µs und startet dann Task3 neu.
Erkennst Du die Auswirkung dieser beiden Tasks ?

CoopOS taskResume, virtueller Interrupt

Virtueller Interrupt: ein Task „weckt“ durch einen Zustand (Pins oder interner Programm-Zustand) einen „schlafenden“ Task.



Task4 „resume“t Task3

Task3 reagiert nach 2.5µs („normalerweise“)

Arduino-Demo Kurs5

CoopOS Part6 Was wird geboten ?

Zunächst PPT-Vortrag

- **taskWaitSig(sig);**
- **taskSetSig(sig);**
- **taskWaitRes(id);**
- **taskFreeRes(id);**

Am Ende der Video-Reihe:

Komplettes CoopOS: Simpel, Schnell, Platzsparend
Leicht anpassbar!

CoopOS taskWaitSig(sig)

taskWaitSig(sig);

Signale können beliebige Werte von 1..255 sein. (Nicht 0 !)
Dieser Befehl sorgt dafür, dass dieser Task sofort aufhört zu arbeiten
Er wartet auf des Senden eines Signals.
Es können mehrere Tasks auf ein Signal warten.

Komplettes CoopOS: Simpel, Schnell, Platzsparend
Leicht anpassbar!

CoopOS taskWaitSig(sig)

taskSetSig(sig);

Signale können beliebige Werte von 1..255 sein. (Nicht 0 !)
Dieser Befehl setzt ALLE Tasks, die auf dieses Signal warten, auf READY.
Der sendende Task macht weiter – kehrt also NICHT mit diesem Befehl zum Scheduler zurück.

Deshalb kann dieser Befehl auch in Interrupt-Routinen benutzt werden.

Er ist besonders nützlich, um Tasks miteinander zu synchronisieren!

CoopOS taskWaitSig(sig)

taskWaitRes(res);

Task wartet auf eine Resource – typisch ist das Display.
Eine Resource kann jeweils nur von einem Task benutzt werden!
Es gibt 3 Fälle:
Die Resource ist frei:
Der Task reserviert die Resource für sich und macht weiter.
Die Resource wurde schon von dem Task reserviert:
Der Task macht weiter.
Die Resource ist besetzt:
Der Task wird unterbrochen. Wenn der Besitzer der Resource die Resource frei gibt, dann wird automatisch der erste Task (Init), der auf die Resource wartet, auf READY gestellt.

Achtung: Resource ist ein Array.
res muss kleiner als MAXRESOURCE sein !!! (config)

CoopOS taskWaitSig(sig)

taskFreeRes(res);

Dieser Befehl markiert eine belegte Resource wieder als frei.
Gleichzeitig wird der erste auf die wartende Resource Task auf READY gestellt.
taskFreeRes wird nur vom Eigentümer-Task akzeptiert!

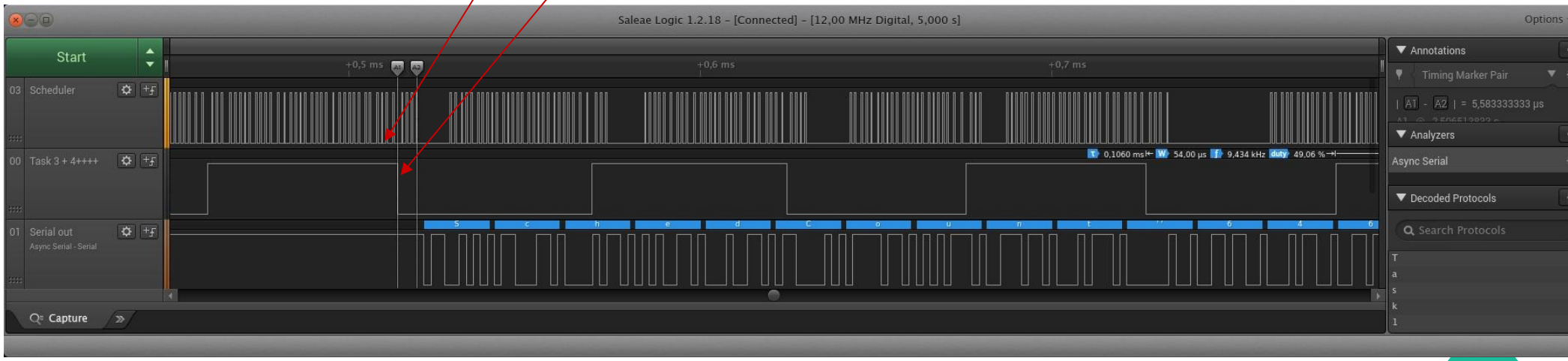
Deshalb nicht im Interrupt nutzbar! Aber es ist gut mit Signalen zu kombinieren.

CoopOS Demo Kurs6

Synchronisation von Ausgaben mit taskWaitRes, taskWaitSig:
1. Warten auf Resource (Task 1 und 5)
2. Warten auf Signal, um mit einem Task zu synchronisieren.
3. Task4,5 setzen Signal bei Flankenwechsel.

Effekt: Ausgabe dauert maximal 30 µs. Um Task3,4 zu keinem Zeitpunkt verzögern zu können, wird von Task3 und Task4 jeweils an den Flanken ein Signal gesetzt.
Die (max.) 30 µs Ausgabe kann nicht mit Task3,4 kollidieren.
Das sorgt für einheitliche ~50 µs Pulse von Task3,4 auch bei Ausgabe !!!

1. 2., 3.



CoopOS Erweiterungen

Tcb wurde um
uint8_t ***Signal;*** // wartend auf Signal
uint8_t ***Resource;*** // wartend auf Resource
Erweitert.

Ein Task kann zu einem Zeitpunkt nur auf ein Signal und eine Resource warten.

enum state wurde um ***WAITSIG*** und ***WAITRES*** erweitert**.**

Das Array
volatile uint8_t *Resource[MAXRESOURCE];*
Speichert den Eigentümer (Task-ID) einer Resource.

CoopOS Task_3, _4

Task_3 und _4 werden jetzt per Signal synchronisiert (statt mit *taskStopMe* und *taskResume*).

Vorteil:
1. Es können mehrere Tasks auf ein Signal warten und werden beim Eintreffen des Signals alle auf READY geschaltet.
2. Signale können auch von Interrupts aus gesendet werden.

Speichert den Eigentümer (Task-ID) einer Resource.

CoopOS Task_3, _4

Task_5 zählt die Anzahl der TaskSwitches pro Sekunde.

Das Ergebnis von **646000** zeigt erstmal die Mächtigkeit und Geschwindigkeit von CoopOS !

Es entspricht (im Mittel) einen Taskwechsel alle **1.55 µs**.

Arduino-Demo Kurs6

CoopOS Demo Kurs7 - Erweiterungen

Serielle Ausgaben sind immer ein Flaschenhals.
Hier werden die Ausgaben wahlweise mit Task_2 oder Task_4,_4 synchronisiert, um Serial.print mit maximal 37µs so unterzubringen, dass wesentliche Tasks davon nicht beeinflußt werden.
Es ist eine Methode, um Tasks sauber und **deterministisch** ineinander zu verschachteln.

Es wurde bisher (anders als bei einem RTOS) kein Timer-Interrupt benutzt.
Hier wird ein Timer1-Interrupt eingeführt um zu zeigen, wie Tasks von einem Interrupt per Signal angestoßen werden können.

Es gibt **100000 Interrupts/s** entsprechend alle **10 µs** – auch 5 µs sind möglich!

Mit **asm_ccount** werden Clock-Ticks gemessen. Bei 160 MHz sind das
6.15 Nanosekunden pro Tick.

```
// read cpu ticks
static inline uint32_t asm_ccount(void) {
    uint32_t r;
    asm volatile ("rsr %0, ccount" : "=r"(r));
    return r;
}
```

CoopOS Demo Kurs7 - Erweiterungen

Tcp und Scheduler wurden in Klassen (class) umgewandelt.

Das bringt keinen zunächst keinen wesentlichen Nutzen, ist aber für die gedacht, die Erweiterungen gerne von Klassen ableiten.

Um auf Signale noch schneller reagieren zu können, wird der erste Task, der auf das gesendete Signal wartet, besonders schnell ausgeführt.

Vom Timer-Interrupt bis zur Ausführung eines signalisierten Tasks vergehen i.d.R. 2-4 µs.
Das ist die Zeit, die der Scheduler benötigt – nachdem der aktuelle Task fertig bearbeitet wurde (bis taskSwitch, ..), um den passenden Interrupt-Task zu starten.

Das ist nicht zu verwechseln mit der Interrupt-Latenzzeit, denn der Interrupt ist ja schon bei der Interrupt-Routine angekommen.

Arduino-Demo Kurs7

Achtung: Bisher liefen alle Demos auf allen Prozessoren.
Demo-7 beinhaltet ESP8266-spezifischen Code.

Es gibt eine spezielle Version für Arduino-Nano.

Arduino-Demo Kurs7

Achtung:

.

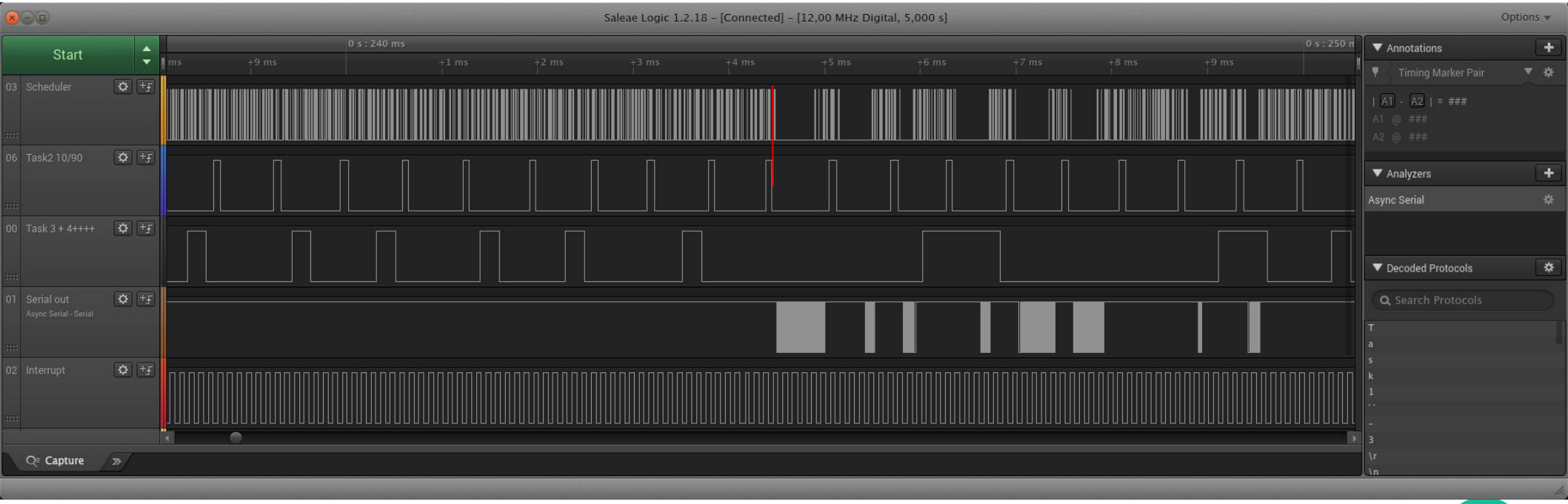
Es gibt eine **spezielle Version für Arduino-Nano**.
Mit **Timer-Interrupt** und **externem Interrupt**.
Der Nano ist etwa um den Faktor 14.5 langsamer als der ESP8266.
Das sollte man bei der Beurteilung der Ergebnisse berücksichtigen.
Trotzdem schlägt er sich tapfer:

- >40000 Scheduler-calls /s
- 20000 Interrupts/s
- Sauberes Task_2-Timing (Synchronisation) trotz serieller Ausgaben
- s. Bild nächste Seite:

CoopOS Demo Kurs7

Arduino-Nano-Demo Kurs-7

Serielle Ausgabe synchronisiert mit Task_2



CoopOS

Demo Kurs8

Arduino-Demo Kurs-7
Arduino-Demo-Nano-Kurs-7

CoopOS Demo Kurs8

Wesentliche Neuerung ist ein externer Interrupt, der ebenfalls per Signal einen Task startet.
Bei prellenden Tastern oder Rotary Dials hat man oft das Problem, dass per Software entprellt werden muss. Das gestaltet such mit CoopOS besonders einfach:

- Interrupt startet einen Task und sperrt sich selbst.
- Der Task macht ein taskDelay(50000) = 50 ms.
- Der Task liest den Input-Pin (entprellt)
- (Rotary Dial: der Task liest den 2. Input Pin)
- Der Task startet die Interrupt-Erfassung neu.

Insbesondere das Auslesen von Rotary Dials wird so sehr vereinfacht !

CoopOS Demo Kurs8

Mit **MySerial.h** wird die Ausgabe von Serial.print (Als **MySerial.print**) in die Ausgabe in einen Puffer umgeleitet.
Dazu gehört der Task **MySer_Task**, der dann die Zeichen im Puffer einzeln ausgibt, wenn es passt.
Das reduziert die Zeit für print-Ausgaben deutlich.

Wer den Kurs bis hier mitverfolgt und seine eigenen Tests gemacht hat, der sollte jetzt über ein sehr solides Grundgerüst verfügen, um CoopOS in der Arduino-IDE, aber auch in jeder anderen C-Umgebung eines jeden Prozessors zu benutzen. Anzupassen sind lediglich **micros()** und ggf. **asm_ccount**.
Ansonsten funktionieren Kurs0-Kurs6 aus der Box vom **AtTiny45** bis zum Supercomputer ! (AtTiny: mit print(F(„...“)) kann noch Platz gespart werden.)

CoopOS kann bei der Entwicklung ein schnelles und mächtiges Werkzeug werden.

Arduino-Demo Kurs8

CoopOS Abschluss-Betrachtung

Modularisierung

Unter CoopOS können vorhandene Tasks schnell zu einem neuen Projekt zusammengefügt werden. Hilfreich ist dabei der problemlose Zugriff auf globale Variablen (bei Interrupts: volatile).
CoopOS findet seine Nische auch gerade da, wo ein RTOS aus Platzgründen nur bedingt oder gar nicht einsetzbar ist.

Verständlich

Gerade bei komplexen Zusammenhängen kommt CoopOS dem menschlichen Denken näher, da stets bekannt ist, wann ein Task unterbrochen wird. Man muss also nicht ständig darüber nachdenken: Was passiert, wenn der Task gerade hier willkürlich unterbrochen wird – ein Gedanke, der einem bei jedem RTOS begleitet.

CoopOS Demo Kurs8

Nachteile

Aber CoopOS hat auch seine Nachteile:

- *taskBegin, taskEnd* dürfen nicht vergessen werden.
- Benutzung von *static* Variablen, die ihren Inhalt über einen taskSwitch hinaus behalten sollen.
- TaskSwitch-Anweisungen nur in Tasks
- TaskSwitch-Anweisungen nicht in Switch/case

Wer es nicht extrem schnell und platzsparend braucht möge sich **CoopOS_Stack_MT** (github) ansehen.

Zusammenfassung

Ein kooperatives Multitasking-System wie **CoopOS**:

- Ist für alle Systeme geeignet, die über ANSI-C verfügen
- Extrem kompakt und übersichtlich
- Kein Assembler-Code
- Sehr platzsparend
- Deterministisch berechenbar
- Extrem schnell
- Leicht zu modularisieren