CoopOS Part-1 What is offered ?

- Why this series?
- Why cooperative multitasking?
- Not just a given library, but understanding
- Examples of designing your own code
- Very vivid simple examples

First, PPT lecture
Then development of small boxed demos for Arduino IDE
With sources.

At the end of the video series:

Complete CoopOS: simple, fast, space-saving
Easily customizable

CoopOS Part-1 What is offered ?

We begin with **very simple** examples – even beginners should understand them !

- What is a **State Machine** ?
- What are **Coroutines** ?
- How to build a **Scheduler** ?
- **Interrupts** (Timer and External) starting Tasks

At the end (demo-8) we have a Multitasking System (**CoopOS**):

- 7 Tasks running
- 2 Interrupt Routines (Timer and External)
- 24% of program space
- 41% of dynamic memory

CoopOS Part-1 What is offered ?

It uses an **ESP8266**, but the Lessons demo-0 to demo-6 should work on all processors from **AtTiny45 to Supercomputers.**
(New: Demo7,8 for Arduino Nano added)

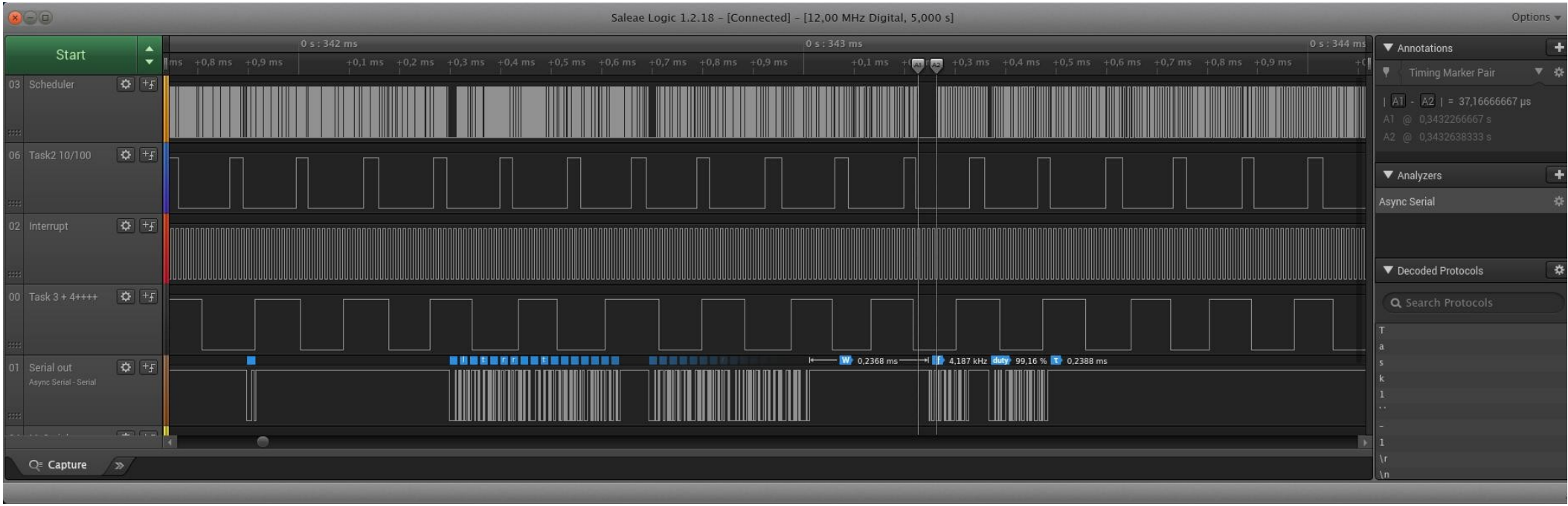
At the end is demo-8 (Esp8266), with: (per second !)

- **500.000 Scheduler-calls/s** (every 2 µs on average !)
  (*Arduino Uno,Nano: 44525 Scheduler-calls/s ~ µs per call 22.45*)
- **100000 Timer Interrupts/s** (every 10 µs, 5 µs works also !)
- Task-call from Interrupt (per Signal <5 µs)
- Signals, waiting for signals, mutex for resources, ..
- CoopOS.h with CoopOS Class: **< 200 Lines**

It may start boring for the professional - but then ...;)

CoopOS Part-1 Understanding

I would like to understand such pictures:



CoopOS

What are Coroutines ?

Coroutines: Term of Melvin Conway 1963:

Donald Knuth: Functions are special cases of Coroutines 1967
(„*The Art of Programming*“)

Implemented in many programming languages
or as a library available.

What are Coroutines ?

In computer science, coroutines are one
Generalization of the concept of a procedure or function. The principal difference between coroutines and procedures is that **coroutines can interrupt their flow and resume later, maintaining their status..** (Wikipedia)

What are Coroutines ?

Each coroutine is a **state machine**. She saves her Voluntary exit state to resume this state at the next call. This will **cause other parts of the function to be executed** each time it is called.

What are Coroutines ?

Coroutines are more of a philosophy than a library.
CoopOS provides the scheduler to manage the coroutines. They run from the AtTiny to the fast PC.
They are not a contrast to an RTOS, but often a good match.
Example: The ESP32 has two cores. This makes it possible to run freeRTOS on one core and CoopOS on the second core. This gives in amazing results! (More about this in another project)
But even on a Linux_preempt with a reserved processor core, CoopOS enables extremely fast task switches – for instance for cooperation with an external processor.

What are Coroutines ?

I pushed it to the extreme with a Rasperry Pi 3+. No Linux, "bare metal".
Only 1 core was used - 3 are still available. Values with CoopOS:

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

http://helmutweber.de/Microcontroller.html

CoopOS

State Machine

```
// Example of a cooperative function


int F1_State=0;

void Task1() {
    Switch (F1_State) {
        case (0):
                printf(„Task1 – Part1\n“);
                F1_State=1;                    // State at the next call
                return;                        // cooperative exit
        case (1):
                printf(„Task1 – Part2\n“);
                F1_State=0;
                return;
    }
}
```

Before the first call

CoopOS

State Machine

```
// Example of a cooperative function


int F1_State=0;

void Task1() {
    Switch (F1_State) {
        case (0):
                printf(„Task1 – Part1\n“);
                F1_State=1;                    // State at the next call
                return;                        // cooperative exit
        case (1):
                printf(„Task1 – Part2\n“);
                F1_State=0;
                return;
    }
}
```

First call

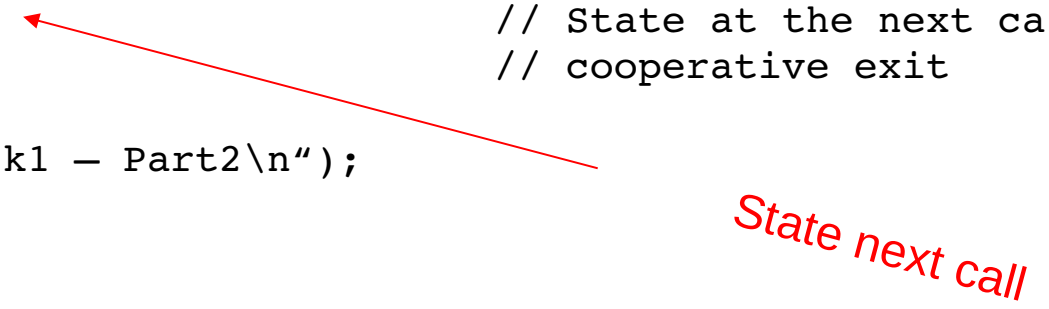
CoopOS

State Machine

```
// Example of a cooperative function


int F1_State=0;

void Task1() {
    Switch (F1_State) {
        case (0):
                printf(„Task1 – Part1\n“);
                F1_State=1;                    // State at the next call
                return;                        // cooperative exit
        case (1):
                printf(„Task1 – Part2\n“);
                F1_State=0;
                return;
    }
}
```



CoopOS

State Machine

```
// Example of a cooperative function


int F1_State=0;

void Task1() {
    Switch (F1_State) {
        case (0):
                printf(„Task1 – Part1\n“);
                F1_State=1;                    // State at the next call
                return;                        // cooperative exit
        case (1):
                printf(„Task1 – Part2\n“);
                F1_State=0;
                return;
    }
}
```

Second call

CoopOS

State Machine

```
// Example of a cooperative function


int F1_State=0;

void Task1() {
    Switch (F1_State) {
        case (0):
                printf(„Task1 – Part1\n“);
                F1_State=1;                    // State at the next call
                return;                        // cooperative exit
        case (1):
                printf(„Task1 – Part2\n“);
                F1_State=0;
                return;
    }
}
```

State next call

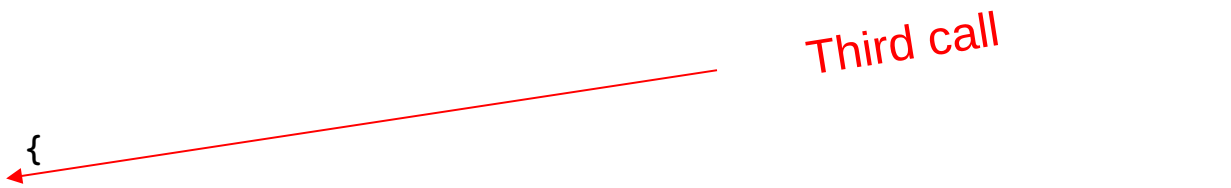
CoopOS

State Machine

```
// Example of a cooperative function


int F1_State=0;

void Task1() {
    Switch (F1_State) {
        case (0):
                printf(„Task1 – Part1\n“);
                F1_State=1;                    // State at the next call
                return;                        // cooperative exit
        case (1):
                printf(„Task1 – Part2\n“);
                F1_State=0;
                return;
    }
}
```



CoopOS

State Machine

```
// Example of a cooperative function


int F1_State=0;

void Task1() {
    Switch (F1_State) {
        case (0):
                printf(„Task1 – Part1\n“);
                F1_State=1;                          // State at the next call
                return;                              // cooperative exit
        case (1):
                printf(„Task1 – Part2\n“);
                F1_State=0;
                return;
    }
}
```

Why these interruptions ?

CoopOS

State Machine

```
// Example of a cooperative function

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: fulfills the same purpose like GLOBAL

```
  // State at the next call
  // cooperative exit
```

To each time only a small part of the function to work very fast !

Example Arduino Lesson-0

With these hints please load, view and try out the appropriate sources, change ...

Cooperative Multitasking

- (**First**) no timing
- Tasks themselves determine when they are cooperative
- Exact point (in the program) of the interruption given
- Easy access to (global) variables
- All existing libraries usable
- Only one stack (saving memory)
- Fast task switching

Preemptive Multitasking (RTOS)

- Tasks are interrupted by a timer interrupt
- Scheduler switches between tasks
- Exact TIME point of interruption given
- Access to (global) variables must be regulated
- **Partly new libraries necessary**
- Each task has its own stack (memory intensive)
- Task switching saves register ==> takes time

Example Arduino Lesson-1

CoopOS Part2 What is offered ?

- Why this series?
- Why cooperative multitasking?
- Not just a given library, but understanding
- Examples of designing your own code
- Very vivid simple examples

First, PPT lecture
Development of TaskControlBlock (TCB) and Scheduler

At the end of the video series:

Complete CoopOS: simple, fast, space-saving
Easily customizable!

TaskControlBlock TCB

The TCB contains all the parameters that define a task

- Task-Function
- State of Task (READY, DELAYED)
- Time of last call

Delay (Pause until the next call)

Fast task switching - but WHEN ?

TaskControlBlock TCB

Defining a structure:

```
struct tcb {
  void          (*Func)();          // Name of Task
  uint32_t      LastCalled;         // Time: last called
  uint32_t      Delay;              // Delay of task
  char          State;              // State: READY, DELAYED
};

#define MAXTASKS 10
int       numTasks;                 // Max. number of Tasks

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

enum state  { READY, DELAYED };
```

Scheduler

Scheduler = Timing of Tasks:

```
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();              // call of Task Function
            Tasks[i].LastCalled = m;
        }
    } // for
} // Scheduler
```

CoopOS defines

Scheduler

For better readability:

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

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

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

CoopOS Usage of **defines**

For better readability:

```
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();
}
```

Example Arduino Lesson-2

CoopOS Part3 Time control

It is time to introduce different timing for each task.

First PPT-Presentation

- **thisTask** – global Pointer to running Task
- **#define taskDelay(microseconds)**
- **taskDelay(val) in Tasks**

At the end of the video series:

Complete CoopOS: simple, fast, space-saving
Easily customizable!

Scheduler

thisTask = global Pointer to running 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 = global Pointer to running 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__:
```

Example Arduino Lesson-3

Look at Demo3.jpg, Serial.jpg, TaskSwitch.jpg

BTW

At the end of Lesson-3 you can find this sketch as
a normal Linux C-program (in the comment)
With my Laptops I get **40.000.000** taskswitches/s !

Interpretation of the results

Taskswitch-times: (ESP8266, 160 MHz)

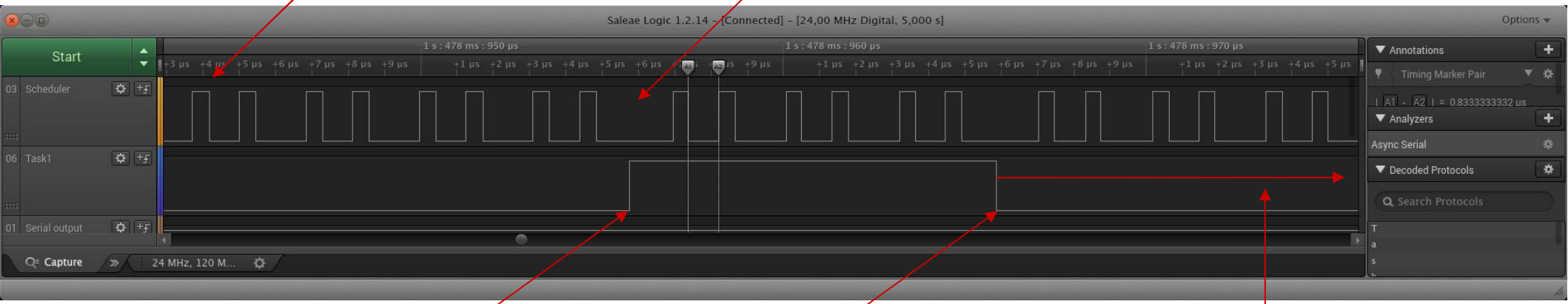
- No Task READY: 0.83 µs
- Task1 READY calling, executing: 2,51 µs

CoopOS

ESP8266 – 160 MHz

Idle

Task1 is running



Task1: 10.125 µs HIGH

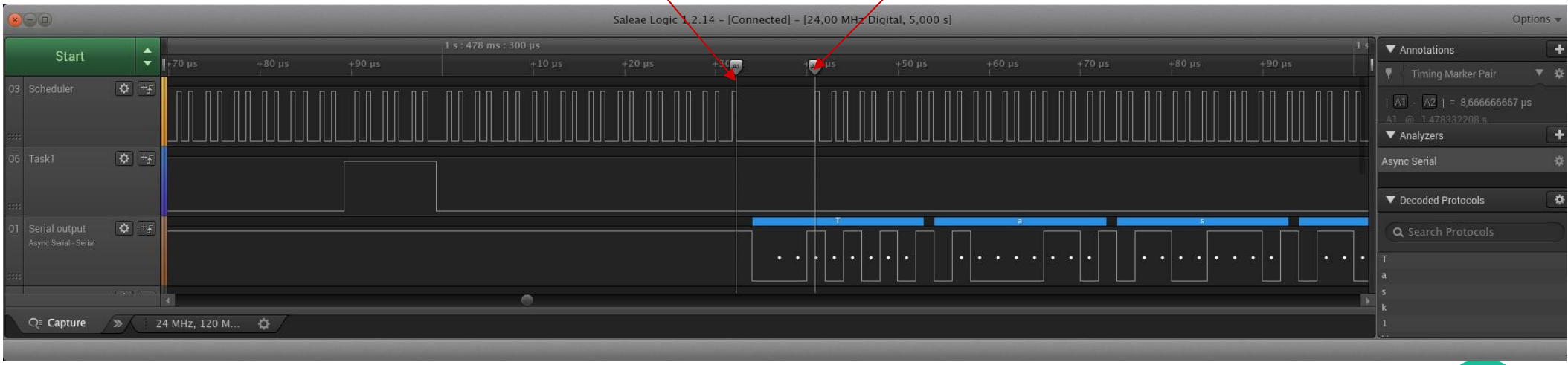
Task1: 100.71 µs LOW

CoopOS ESP8266 – 160 MHz

One criticism of cooperative systems is:
The longest execution time of a task must always be taken into account.
And that's true: Here's the serial output with ~ 9µs!
BUT:
Thus, after HARD-REALTIME conditions still at least all
10 µs a TaskSwitch instead.

A priority control is still pending!

8.67 µs



CoopOS Part4 What is offered ?

First: PPT-Presentation

To clean up;
TaskInit(..);

At the end of the video series:

Complete CoopOS: simple, fast, space-saving
Easily customizable!

CoopOS

CoopOS.h, TaskInit(..)

Clean up:

- Sources for multitasking have been outsourced → **CoopOS.h**
- No library but only an #include file with **less than 100 lines**

- A **TaskInit(...)** was added to initialize the TCBs at the beginning.

- The **main program** is extremely short and clear

Example Arduino Lesson-4

CoopOS Part5 What is offered ?

First: PPT-Presentation

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

Priority of Tasks

At the end of the video series:

Complete CoopOS: simple, fast, space-saving
Easily customizable!

CoopOS

TaskHandle, taskStop, taskResume

TaskHandle:
TaskInit (..) returns a **TaskHandle** with which other tasks can make state changes of Tasks with known handle.

taskStop(id):
The state "BLOCKED" was reintroduced.
taskStop(id) stops the (foreign) task [id].
He can not bring himself back to life!

taskStopMe():
taskStopMe () stops the task immediately.
He can not bring himself back to life!
But another task can help him to restart:

taskResume(id)
Resumes another Task: BLOCKED->READY

CoopOS Priority

Priority:
Although the field Priority was added in tcb, so far
(and for the time being) not used.
The scheduler goes through all available tasks after each call.
There is no priority (round robin).

Task3 could reactivate Task20 with taskResume (id).
But the impact would only be effective after Task4-19.
In the config part of CoopOS.h can
INTERNAL_PRIO be switched on and that means:
Each time, when the scheduler has started a task the scheduler will leave (return). At the next start the scheduler searches again from task0.
Tasks that were initialized first get a higher priority.

CoopOS taskResume

Usage:
Consider in the demo Task3:

```
void Task3() {
  taskBegin();
  while(1) {
    digitalWrite(12,HIGH);
    taskStopMe();                // I am going to sleep – good night ;)
  }
  taskEnd();
}
```

Task3 sets pin 12 to HIGH and then goes to sleep.
Pointless? This can only be understood in cooperation with Task4.

CoopOS taskResume

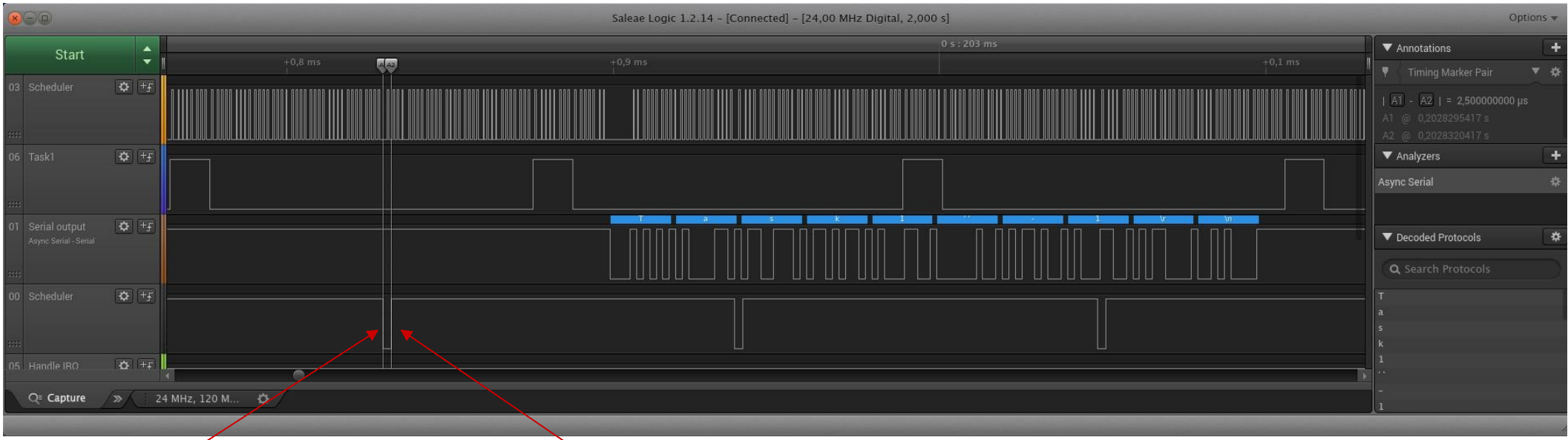
Usage:
Consider in the demo Task4

```
void Task4() {
  taskBegin();
  while(1) {
    taskDelay(100);
    digitalWrite(12,LOW);
    taskResume(T_ID3); // "normally" reacts after 2.5µs (Init: Place 1)
  }
  taskEnd();
}
```

Task4 sets pin 12 to LOW, waits 100µs and then restarts Task3.
Do you recognize the impact of these two tasks?

CoopOS taskResume, virtueller Interrupt

Virtual Interrupt: a task "wakes" a "sleeping" task due a state change of pins or internal program state.



Task4 „resumes“ Task3

Task3 reacts 2.5µs („usually“)

Arduino-Demo Lesson-5

Look at Demo-5.jpg, VirtIRQ.jpg

CoopOS Part6 What is offered ?

First: PPT-Presentation

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

At the end of the video series:

Complete CoopOS: simple, fast, space-saving
Easily customizable!

CoopOS taskWaitSig(sig)

taskWaitSig(sig);

Signals can be any values from 1..255. (Not 0!)
This command causes this task to stop working immediately
He is waiting for the sending of a signal.
Several tasks can wait for a signal.

Signals are a mighty tool to start a Task very fast after an External or internal State changed

CoopOS taskSetSig(sig)

taskSetSig(sig);

Signals can be any values from 1..255. (Not 0!)
This command sets ALL tasks waiting for this signal to READY.
The sending task continues - so it DOES NOT return with this command back to the scheduler.

Therefore, this command can also be used in interrupt routines.

It's especially useful for syncing tasks!

CoopOS taskWaitRes(res)

taskWaitRes(res);
Task is waiting for a resource – for instance the display.
A resource can only be used by one task at a time!
There are 3 cases:

The resource is free:
The task reserves the resource for itself and continues.

The resource has already been reserved by this task:
The task continues.

The resource is busy:
The task is interrupted. If the resource owner releases the resource, then the first task (Init) waiting for the resource is automatically set to READY.

Attention: Resource is an array.
res must be smaller than MAXRESOURCE !!! (Config)

CoopOS taskFreeRes(res)

taskFreeRes(res);

This command marks an occupied resource as free again.
At the same time, the first one is set to the waiting resource task on READY.
taskFreeRes is only accepted by the owner task!

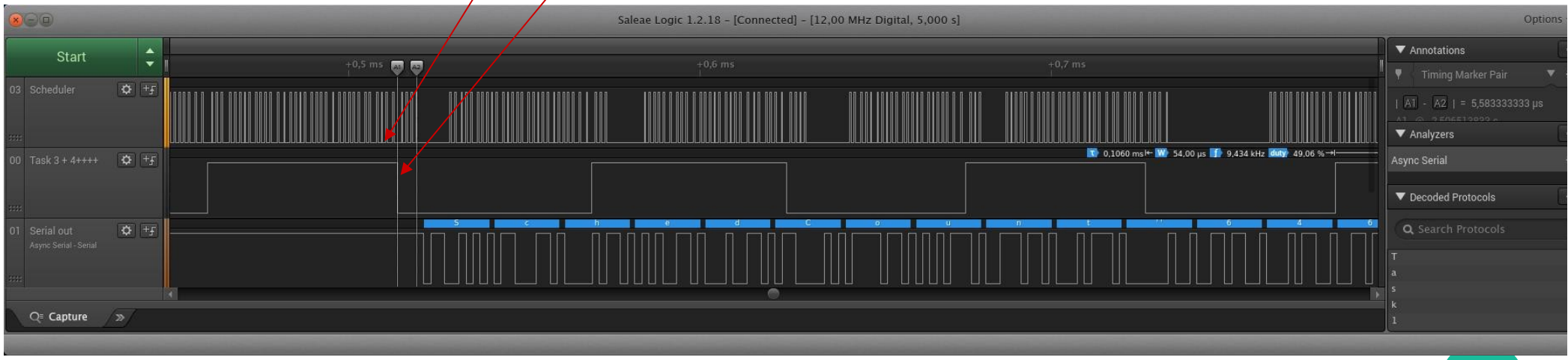
Therefore not usable in the interrupt! But it is good to combine with signals.

CoopOS Demo Lesson-6

Synchronization of Output with taskWaitRes, taskWaitSig:
1. Waiting for Resource (Task 1 and 5)
2. Waiting for signal to synchronize with a task.
3. Task4,5 set signal at edge change.

Effect: Output takes a maximum of 30 µs. To delay task3,4 at any time, a signal is set by Task3 and Task4 at the edges.
The (max.) 30 µs output can not collide with Task3,4.
This ensures consistent ~ 50 µs pulses from Task3,4 even at output !!!

1. 2., 3.



CoopOS Expansions

Tcb was expanded:

```
  uint8_t signal;       // waiting for signal
  uint8_t resource;     // waiting for resource
```

A task can only wait for one signal and one resource at a time.

enum state has been extended ***WAITSIG*** and ***WAITRES*** **.**

The Array
volatile uint8_t ***Resource[MAXRESOURCE];***
Saves the owner (task ID) of a resource.

CoopOS Task_3, _4

Task_3 and _4 are now synchronized by signal (instead of using taskStopMe and taskResume).

Advantage:

1. Several tasks can wait for a signal and are all switched to READY when the signal arrives.

2. Signals can also be sent from interrupts.
   Saves the owner (task ID) of a resource.

CoopOS Task_5

Task_5 counts the number of TaskSwitches per second.

The result of **646000** shows for the first time the power and speed of CoopOS!

It corresponds (on average) to a task change every **1.55 µs.**

Arduino-Demo Lesson-6

Look at SIG_SYNC.jpg

CoopOS Demo Lesson-7 – More Extensions

Serial issues are always a bottleneck.
Here the outputs are optionally synchronized with Task_2 or Task_4, to accommodate Serial.print with a maximum of 37µs so that essential tasks are not affected.
It's a way to neatly and deterministically nest tasks together.

So far (unlike an RTOS) no timer interrupt has been used.
Here, a timer1 interrupt is introduced to show how tasks can be triggered by an interrupt sending a signal.

There are **100.000 interrupts**/s corresponding to every **10 µs** - even 5 µs are possible!

With **asm_ccount** Clock ticks are measured. At 160 MHz that is
6.15 Nanosekunden per 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 Lesson-7 – More Extensions

Tcp and scheduler have been converted into classes (class).

This does not initially bring any significant benefit, but is intended for those who like extensions derived from classes.

To be able to respond to signals even faster, the first task that waits for the transmitted signal, runs very fast.

From the timer interrupt to the execution of a signaled task it needs 2-4 µs. This is the time the scheduler needs - after the current task has been finished (until taskSwitch, ..) to start the appropriate interrupt task.

This is not to be confused with the interrupt latency, because the interrupt has already arrived at the interrupt routine.

Arduino-Demo Lesson-7

Look at Lesson7.jpg

Attention: So far, all demos ran on all processors.
Demo Lesson-7 includes ESP8266-specific code.

There is a special version for Arduino Nano.

Arduino-Demo Lesson-7 Nano

Look at Demo-7-Nano.jpg

Caution:

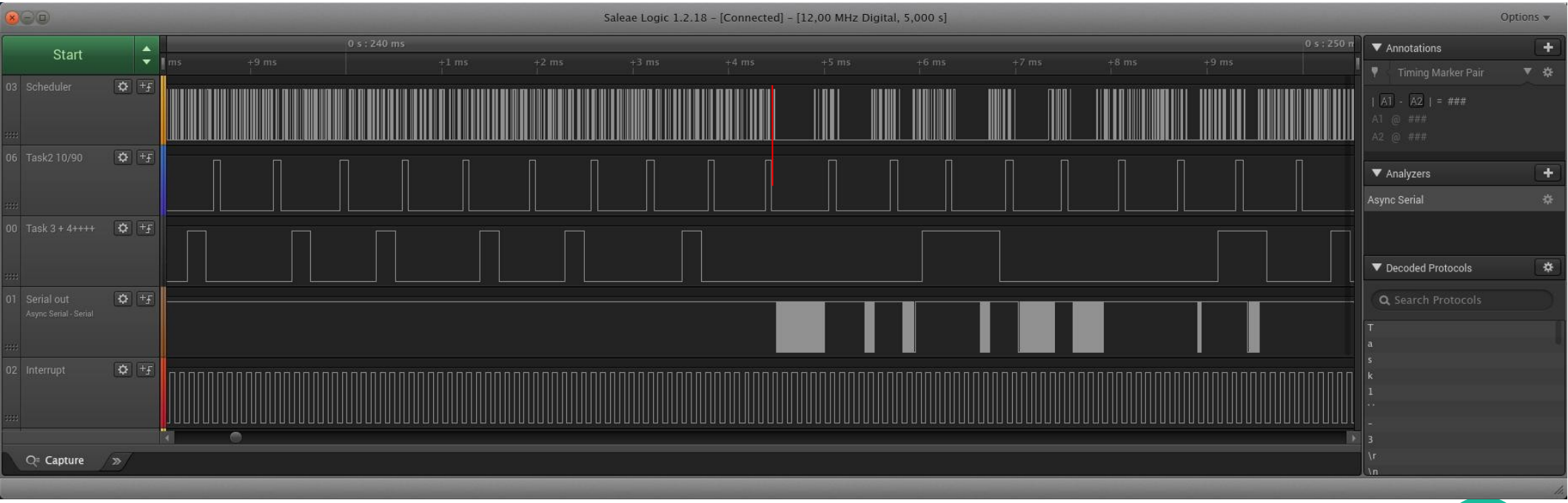
There is a special version for Arduino Nano.
With timer interrupt and external interrupt.
The Nano is about 14.5 slower than the ESP8266.
This should be considered when assessing the results.
Nevertheless, he bravely strikes:

- > 40.000 scheduler calls / s
- 20.000 interrupts / s
- Clean Task_2 timing (synchronization) despite serial output
  s. Picture next page:

CoopOS Demo Kurs7

Arduino-Nano-Demo Lesson-7

Serial output synchronized with Task_2



Arduino-Demo Lesson-7
Arduino-Demo-Nano-Lesson-7

CoopOS Demo Lesson-8

The main innovation is an **external interrupt**, which also starts a task by signal.
With bouncing buttons or Rotary Dials one often has the problem that must debounce by software. That's what makes it so easy with CoopOS:
Interrupt starts a task and locks itself.
The task does a taskDelay (50000) = 50 ms.
The task reads the input pin (debounces)
(Rotary Dial: the task reads the 2nd input pin)
The task restarts the interrupt acquisition.

In particular, the reading of Rotary Dials is so much easier this way!

CoopOS Demo Lesson-8 MySerial.h

MySerial.h redirects the output of Serial.print (as **MySerial.print**) for output to a buffer. The task MySer_Task then outputs the characters from the buffer, when there is time to do so.
MySerial is an instance of the new class mySerial.
This class does the output to the buffer with own functions!
This significantly reduces the time spent on print.

If you've followed the course up here and have done your own tests, you should now have a very solid framework to use CoopOS in the Arduino IDE, but also in any other C environment of any processor. To adjust are only **micros ()** and possibly **asm_ccount**.
Otherwise, Lesson-0 to Lesson-6 work out of the box from the AtTiny45 to the supercomputer! (AtTiny: print (F ("...")) saves space.)

CoopOS can become a fast and powerful tool in development.

Arduino-Demo Lesson-8

>500.000 Scheduler calls/s
100.000 Timer Interrupts/s
From Timer Interrupt to schedule Task 6: <5µs
External Interrupts

7 Tasks running
2 Interrupt Routines
24% of program space
41% of dynamic memory

CoopOS Conclusion

Modularization

Under CoopOS, existing tasks can be quickly merged into a new project. Helpful is the easy access to global variables (for interrupts: volatile). CoopOS also finds its niche just where a RTOS is limited or not usable due to space limitations.

Understandable

Especially in complex contexts, CoopOS is closer to human thinking, as it is always known when a task is interrupted. So you do not have to constantly think about it:
What happens if the task is interrupted arbitrarily right here - a thought that accompanies you on every RTOS.

Disadvantage

But CoopOS also has its disadvantages:

- *taskBegin, taskEnd* must not be forgotten.
- Using **static** on local variables that should keep their contents beyond a taskSwitch.
- TaskSwitch statements only in tasks

Who does not need it extremely fast and space saving may
View **CoopOS_Stack_MT** (github).

Conclusion

A cooperative Multitasking-System like **CoopOS**:

- Is suitable for all systems that have ANSI-C
- Extremely compact and clear
- No assembler code
- Very space saving
- Deterministically calculable
- Extremely fast
- Easy to handle and modularize