CoopOS_Stack_MT_Nano

Generated by Doxygen 1.8.11

Contents

1 What does this program ?

Arduino Multitasking Stackchange Beta 1.0

CoopOS with Stackframes

You never thought that multitasking can be so easy:

This is a simple and fast approach to multitasking.

- Very easy to use even for beginners - nothing else comes close
- Usable for professionals
- Reliable timings
- Tasks have priorities
- Full documentation
- Compatible with all Arduino Libraries
- Not a librariy but only one file to include
- Could be combined with RTOS's as Idle-Task
- Easy to port to other processors • Valuable tools for development and tests

- 25000 (40 µs) TaskSwitches per second on Arduino-UNO / -NANO are possible
- Breakpoints
- Up to 200000 Interrupts/s !

More than 50 pages documentation !

>> Introduction

1.1 All Demos as ZIP-files:

Demos

2 Doxyfile

This is the external doxygen documentation

3 Introduction

If you are a serious programmer you may have missed multitasking in your Arduino programs.
You want to send some commands to your program - but all the other actions should go on undisturbed:
Blinking the LED, measuring temperatures, moving the stepper motor ...

That could be done by clever programming - but it is a nightmare.
And sometimes you want to include a function as snippet from another sketch - with easy integration.

Then >b>Multitasking is what you want!
The Arduino IDE has a very simple Scheduler.h and yield() for the Arduino Due, but for NANO and UNO there is nothing.
But in the meantime you will find countless examples for multitasking - preemptive and cooperative.
It is not easy to choose one !

Why do you may want to use this one?
Because it is very easy to learn, fast and full documented with enough explanation for beginners,
but flexible enough for professionals.
And it contains the additional tools you need !

My **CoopOS** is faster, has a smaller footprint and has more functions. But you have to learn a bit more.

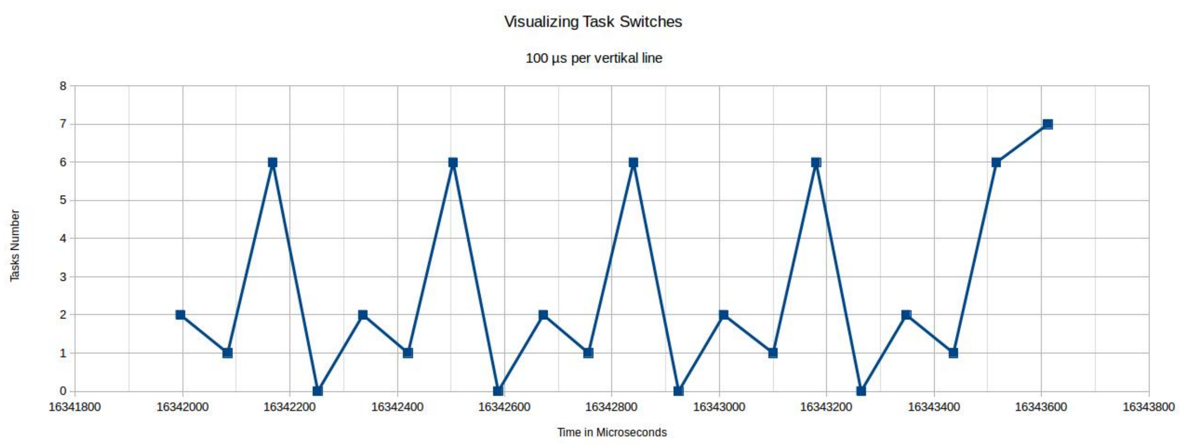
Using multitasking you have different functions which seem to run simultaneously:
At the first glance the following sketch seems to be a normal Arduino sketch.
But at the second view you see the **while(1) {}** loops in the tasks.
With that construction a normal Arduino sketch will get captured in one of the tasks.

But some additional lines will make it work !
And all files are tabs in your Arduino sketch - no library. So it is easier to see what is done behind the scenes and you are encouraged to make your own modifications - and maybe build your own customized library.
It also is a good introduction to multitasking systems. Though this is cooperative it should you give a glimps of what to do to make it coopeartive: Yield() and TaskSwitch() then must be done additionally by a timer interrupt.

Prog 0

```
 //// CoopOS_Stack_MT - Demo   (C) 2019 Helmut Weber
//// Demo0

 #include "TaskSwitchDemo.h"
#define LED 13

 void Task1() {                                          // Blink LED
    while(1) {                                          // <<< You will find these while(1) loops i
        digitalWrite(LED,HIGH);                         // if you call this in a normal Arduino ske
        Delay(50000);
        digitalWrite(LED,LOW);
        Delay(50000); // microseconds
    }
}

 void Task2() {                                          // Write to Serial Line
    while(1) {                                          // Another infinit loop !
        Serial.println("Hello World");
        Delay(1000000);
    }
}

 void setup() {
    Serial.begin(500000);
    StackInit();                                        // Init the stacks for all tasks

     TaskInit("T1", Task1, 90, 100,   0, READY);     // define 2 tasks for multitasking:
    TaskInit("T2", Task2, 90, 100,   0, READY);

     StartMultiTasking();                               // start the system:
}

 void loop() {                                          // loop is never called
    //// this is never called !!!
    Serial.println("Hoops? How did you come to this line ???");
}
```

Breakpoints and visualizing task switching in the advanced version:



Read more in:

>> Overview

4 Overview

Realtime Operating systems are the prefered tools for most measurements using embedded systems.
ChibiOS and others are running on the Arduino UNO.

But sometimes a TickTime of 1 ms is too long.
And you have to learn a lot (as a beginner) to write your first real programs.

CoopOS is faster, but there are some drawbacks:
• Tasks must be surrounded with CoopOS_Begin, COOPOS_End
• No SWITCH statement in tasks
• Local variables must be static
• Task switch { Yield(), Delay(), ...} are allowed only inside tasks - not in called functions

CoopOS_Stack_MT is so simple even a beginner will be able to use multitasking in 5 minutes.

If you have never used multitaskingyou have to turn a switch in you head now.
Up to now your program was a long worm. After you have done setup you may look at loop()
and you can follow all instructions until the end - then loop is ready and exits - just to start again

And then you get the problems:
While you print messages you want to read the serial line and wants to stop scrolling the output, if a character is sent.
No problem. But how can you manage, that the LED blinks without stopping at the same rate?
And what can you do to continue runninf the stepper motor and ...

It is obvious: you need some kind of multitasking!

Now you have another problem. Which kind of multitasking do you need?
Some are telling you, that multitasking only works correct with an RTOS (Real Time Operating System).
This is the one and only way to write professional multitasking programs. Period.
You have to believe or you are a silly Beginner. Period.
But first of all there where not so much real RTOSs for Arduino.

And when you got one, you have other problems:

Every task nedds an own stack. But how big is big enough? And then you have to nearn a lot about mutexes, semaphores, signals and so on. You have to study a lot just to get 3 or 4 task to run.

Some are telling you, cooperative multitasking is much easier to learn, has less footprint and does faster task switching.

What is the truth? Answer: Both - or - it depends on.

I could not tell you how YOU should handle these problems, but I can tell you, how I do it:
At first I look, if there is a prefered RTOS for an embedded system/microcontroller
For instance the ESP32 comes with freeRTOS. You may try other RTOS with this processor but it is not advisable.
If I need fast reaction I use my CoopOS on the 2. processor.

For STM devices chibiOS/RT is great and is under active development.

For Arduino I use chibiOS/Nil or my CoopOS. ChibiOS deliver timings with very small jitter. CoopOS therefore is faster AND is nearly independent of the hardware. All you to change the call to micros() and the connections to the pins and attach interrupts (only, if you use them).

The advantages are:

- Fast easy to transport to other systems - it runs on an AVR Tiny as well as on a big machines (pure Ansii-C) all

- libraries are usable without a change

Why should you use CoopOS_Stack_MT ?

Because it is so easy to use!

You will be able to run multitasking programs within 5 minutes.

It is good enough to make your home projects running.

It is easy to combine different projects you have done before.

Look at the program below containing this task:

Prog 1

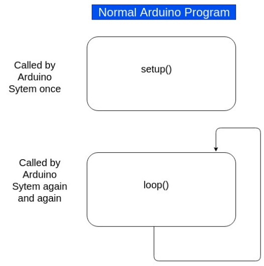
```
  void Task() {
  // do some initialisation for this task like you know it from Arduino-setup()~

  while(1) {
        running forever
        here comes your task loop running in a circle
        ...

        Delay(1000);

        <<<<<<<<<< In this delayed time the other tasks are running
        use Yield(0) or Delay(microseconds) for cooperative tasks giving back the control to the
        <<< Here it is your turn again - after the time Dely(...)-time or, if Yield() is used , a
        of some microseconds

  } // end while
}
```

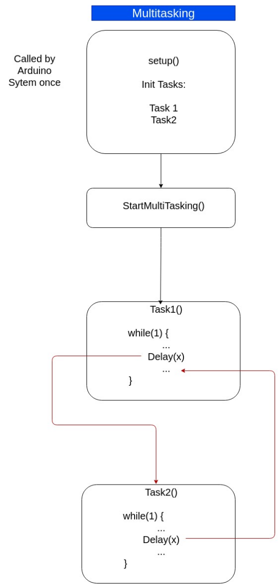
In the Arduino-IDE you normallay have loop() to start your program again and again.

What you normally wrote into the loop function is a task now.
It loops forever using while(1) (or for(;;))

But how the other tasks get started?
The miracle happens when Delay() - not delay() - is called.

While this task has to pause for some time other tasks are running.
In the ideal case Task() is restarted after the Delay(–) time and did not know, what happend
in the meantime.





<< Introduction

>> Getting started

5 Getting started

Let s take a look on a real program. But first of all try to learn, how multitasking programming changes your methods of
thinking. You know you have to write programs for an Arduino that never end. There is nothing like a linux terminal window where
you can start another program.
The program for an Arduino normally runs forever.
For a multitasking system that is almost true for the different tasks. Nearly all tasks run in a loop

New Thinking

CoopOS_Stack_MT is a cooperative multitasking system
Think of each task you start as if it has it s own processor. And here is the "hello world" for embedded systems:
Here is our first program:

Each task seems to be a whole program:

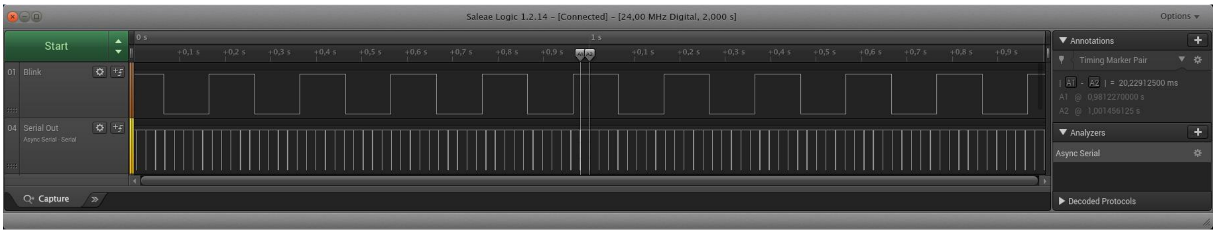
Prog 2

```
//// Simplest Multitasking ever ! - Demo 0
//// (C) 2019 Helmut Weber

#include "TaskSwitchDemo.h"

#define LED 13

//// --------------------- Definition of Tasks ------------------------

void Task1() {
  while(1) {
    digitalWrite(LED,HIGH);              // LED on
    Delay(100000);                       // Return to Scheduler - Delay 100.000 µs
    digitalWrite(LED,LOW);               // LED off
    Delay(100000);                       // Return to Scheduler - Delay 200.000 µs
  }
}

void Task2() {
int count=0;
  while(1) {
    Serial.print(micros()/1000); Serial.print(" ");
    Serial.print("This is Task 2: #");
    Serial.println(++count);
    Delay(20000);                                    // Every 20 milliseconds
  }
}

//// An example to get real 20 ms cycle time
//// void Task2() {
//// int count;
//// unsigned long m,m2;
////   m=micros();
////   while(1) {
////     Serial.print(micros()/1000); Serial.print(" ");
////     Serial.print("This is Task 2: #");
////     Serial.println(++count);
////     m2=micros();
////     Delay(20000-(m2-m)-83);                        // Every 20 milliseconds
////     m=micros();
////   }
//// }

void Task3() {
  while(1) {
    while(Serial.available()) {
      Serial.print(micros()); Serial.print(" ");
      Serial.println( Serial.read());            // Get all characters as fast as possible
      Yield(0);                                  // But check, if a task with higher priority i
    }
    Delay(10000);
  }
}
```

```
 void setup() {
   Serial.begin(500000);

   // Init the some space to use as stack:
   StackInit();

   // Tell the system, which functions are tasks
   TaskInit("T1", Task1, 100, 100,    0, READY);
   TaskInit("T2", Task2, 100, 100,    0, READY);
   TaskInit("T3", Task3, 100, 100,    0, READY);

   // And start the tasks:
   StartMultiTasking();
 }

void loop() {  // never reached

}

OUTPUT:

Stack allocated: 765
Free Ram now   : 776

T1: Stack free for next task: 490
T2: Stack free for next task: 390
T3: Stack free for next task: 290
2 This is Task 2: #1
23 This is Task 2: #2
43 This is Task 2: #3
64 This is Task 2: #4
85 This is Task 2: #5
105 This is Task 2: #6
126 This is Task 2: #7
146 This is Task 2: #8
167 This is Task 2: #9
188 This is Task 2: #10
208 This is Task 2: #11
229 This is Task 2: #12
250 This is Task 2: #13
270 This is Task 2: #14
291 This is Task 2: #15
312 This is Task 2: #16
332 This is Task 2: #17
353 This is Task 2: #18
374 This is Task 2: #19
394 This is Task 2: #20
...

12044 This is Task 2: #581
12065 This is Task 2: #582
12066660 49                     // <<<< Serial Monitor: send characters
12067260 50
12067860 51
```

```
12068452 52
12069052 53
12069644 54
12070244 55
12070836 56
12071436 57
12086 This is Task 2: #583
12107 This is Task 2: #584
```

Let's analyse the results:



1) Tasks1 does the blinks with a cycletime of 200.2 ms

2) The text output of Tasks comes with a cycle time of 20,2 ms. 20 ms for Delay(20000) and 200µs for the Serial.print(...);

3) A String sended from Serial Monitor are kept ever 600 µs (includiing taskswitch!)

You can try it: make a single Arduino tasks from each task in the program and you will get very comparable Timings.

BTW: Noboy is talking about: For every program is true that a Delay(x) delays just that time x. If you have a while(1) ..
or the normal Arduino loop() that mean, you have to add the time for all other instructions to get the total cycle time

BTW: Noboy is talking about: RTOS's are cooperative too. A Delay(x) or Yield() will immediately initiate a task switch!
If the only switch a task after a tick time they would not be usable. So the tick is some kind of safty: A taskswitch
even occures, if the task is NOT cooperative. That is the main difference.<br Non of our tasks needs 1ms or more for a total cycle.

CoopOS_Stack_MT delivers precise deterministic resuls.

But tasks normally are not running for it's own. They have to communicate with each other:

<< **Introduction**

>> **Intertask Communication**

6 Intertask Communication

Here is another Example:

Prog 3

```
//// CoopOS_Stack_MT - Prog 3
//// (C) 2019 Helmut Weber

//// ============================================================================= For Multita
#include "TaskSwitchDemo.h"
//// ============================================================================= For Multita

volatile int PosX=1;
volatile int One=1;
volatile int DelConst=10000;
volatile int Del=10;

#define LED 13

void Task1() {
  while(1) {
    digitalWrite(LED,HIGH);
    Delay(50000);
    digitalWrite(LED,LOW);
    Delay(50000);
  }
}

void Task2() {
  while(1) {
   PosX+=One;
   for (int i= 0; i<PosX; i++) Serial.write(" ");
   Serial.println("*");
   Delay(DelConst);
  }
}

void Task3() {
  while(1) {
    if (PosX==0) One=1;
    if (PosX==80) One=-1;
    Delay(DelConst-(DelConst/10));        // must be  a little bit faster than Task1 !!!
  }
}

void Task4() {
  while(1) {
    if (DelConst>1000) if (PosX==1) DelConst-= DelConst/10;;
    Delay(DelConst-100);
    if (DelConst<=1010) DelConst=10000;
```

```
  }
}

void setup() {

  Serial.begin(500000);

  //// ================================================================= For Multi
  StackInit();
  //// ================================================================= For Multi

  //// ================================================================= For Multi
  TaskInit("T1", Task1, 90, 100,   0, READY);
  TaskInit("T2", Task2, 90, 100,   0, READY);
  TaskInit("T3", Task3, 90, 100,   0, READY);
  //TaskInit("T4", Task4, 90, 101,   0, READY);

  //// ================================================================= For Multi

  //// ================================================================= For Multi
  StartMultiTasking();
  //// ================================================================= For Multi

}

void loop() {
  // this is never called !!!
  Serial.println("Hoops? How did you come to this line ???");
}
```

Some comments concerning global variables:

One of the biggest advantages of a cooperative system compared to RTOS's:
RTOS tasks never know, when the get switched out. If a task A increments a global vaiable it could be possible that the scheduler switches the task A out just in that moment and start another task B, which also writes to this variable. When task A continues it's work, it got a problem.
Some mechanism are neccessary to solve this problem.>br>
A cooperative task knows, that it will not get disturbed until it reliquishes the control to the scheduler! (Well - another thing are interrupts).
So it is much easier for cooperative systems the communicate throug global variables.

Task 1 does the blining as before. It demonstrates the independence from the other tasks

Task 2 is very simple: It writes PosX spaces and then a '*'. After that it increments PosX.
Than it Delay() for 10 ms . That means PosX will be incremented until it overflows and the lines will get longer and longer!

Task 3 prevents PosX the reach infinity. It has a caycle time 10% less than Task 2. So it will be able to see allways the changes Task 2 made.
If PosX reaches 80 Tasks 3 changes the sign of the increment of Task2. The the is true if Task 2 reaches 0:

This is the result: (Attention: Init of Task 4 is commented out!)

```
x
 x
  x
   x
    x
     x
      x
       x
        x
         x
          x
           x
            x
             x
              x
               x
                x
                 x
                  x
                   x
                    x
                     x
                      x
                       x
                        x
                         x
                          x
                           x
                            x
                             x
                              x
                               x
                                x
                                 x
                                  x
                                   x
                                    x
                                     x
                                      x
                                       x
                                        x
                                         x
                                          x
                                           x
                                            x
                                             x
                                              x
                                               x
                                                x
                                                 x
                                                  x
                                                   x
                                                    x
```

```
    x
     x
      x
       x
        x
         x
          x
           x
            x
             x
              x
               x
                x
                 x
                  x
                   x
                    x
                     x
                      x
                       x
                        x
                         x
                          x
                           x
                            x
                             x
                            x
                           x
                          x
                         x
                        x
                       x
                      x
                     x
                    x
                   x
                  x
                 x
                x
               x
              x
             x
            x
           x
          x
         x
        x
       x
      x
     x
    x
   x
  x
 x
x
```

```
                                                  x
                                                 x
                                                x
                                               x
                                              x
                                             x
                                            x
                                           x
                                          x
                                         x
                                        x
                                       x
                                      x
                                     x
                                    x
                                   x
                                  x
                                 x
                                x
                               x
                              x
                             x
                            x
                           x
                          x
                         x
                        x
                       x
                      x
                     x
                    x
                   x
                  x
                 x
                x
               x
              x
             x
            x
           x
          x
         x
        x
       x
      x
     x
    x
   x
  x
 x
x
 x
  x
   x
    x
```

Well, the result is not astounding.
But you may ask: ""Why, the hell, do I need a second task to test the limits? That could be done in Task 2 itsself."

Let's assume we want to control a robot. Task 2 has to control the stepper motor(ś). It has to know:
1) the direction to move
2) the speed for stepping
And these are parameters, which are produced from the rest of the program.
And thats is the main goal of multitasking! Making modules with a very concrete task.
Here Task 2 is responsible for moving using the parameters PosX, direction (One) and speed (DelConst).

Now it's time to uncomment Init(Task4). Task 4 is responsible for the speed. Try it!
And why the global variables are volatile? It tells the compiler to read the variable from memory an do not save them in registers. Here it is not neccessary, but it is a good habit do use volatile to be safe.

Now you know all to do own small experiments, Yes, I recommend to do so!

But then you will come to a point where you want to know the meanings of the parameter of Init(), how to check stack sizes and more.

It follows a program - for the experienced - with the explainations you may want:

<< Getting started

>> Initializing Tasks

7 Initializing Tasks

If you look at the source if is not obvious, which functions are just pure functions and wich functions should act as tasks.
Therefore it is advisable the name functions which should be tasks with a "task" in it's function name.
But that is for readablity only!

You have to inform the system about the tasks to start. And that is the role of TaskInit:

```
////       NAME   FUNCTION    STCK-LEN  PRIO  DELAY STATE
uint8_t TaskInit(char* _name,
    FuncPt _function,
    int16_t _stackLen,
    uint8_t _prio,
    unsigned long _delay,
    State _state)
```

Parameters:

_name:
You may give the tasks a name as character string. This is used for output only.
You are free to choose a name as you wish.

_function:
This is the function name of a function which should act as a task.

_stackLen:
The length of the stack of this task in bytes. 100 is a good starting point. For optimizing stacksize see "Tools".

_prio:
The priority of the task. Allowed are numbers from 0 to 255. But the prioritie numbers are onls use as relative priorities.
That means it is only important, which task has a higher priority than others.
When a task switch is initiated the program will search through the whole taskslist, which task is READY and has the
highest priority. I f two task have the same priority AND both tasks are READY then the first of these tasks is executed.
Its is a good practice though not mandatory to give all tasks different priorities.

_delay:
In microseconds. If _delay is > 0 than this tasks starts later than all other tasks.
If you have a running system
and then you create an additional task it could be helpful to let the working system run and start the new task seconds or>br> minutes later to see the difference.

_state:
Can be READY or BLOCKED. A Blocked task will not start but is contained in the list. It can be resumed later by one task.

The first task in the TaskInit table will create the task Idle() with the tasknumber 0 automatically; Idle has the priority 0 and is only called when no other task is READY.

The tasks are NOT running after TaskInit()
This is done by:
StartMultiTasking()

<< Intertask Communication

>> Tools

8 Tools

Now we are dealing with "TheProgram"
You will find it as "TheProgram.zip". Unpack it to yout Sketchfolder. (Mine is named "Arduino").
Open the sketch. You will find the tabs:

- CoopOS_Stack_MT_Nano.ino
- MySerial.h
- Pins.cpp, Pins.h
- TaskSwitch.h

  These file are explained in detail below

8.1 Serial Output

In all programs (independently of the processor and system!) the output to a serial line may disturb your timings sensitively, especialy when using cooperative multitasking. The effects get less dramatic through printing into a buffer
and send the buffer character by character through as task. And this is what "MySerial.h" does.
MySerial is a calls inherited from stream. In your program you replace Serial.print(x) through MySerial.print(x).
This will print into the buffer Out[]. And a task like this:

```
void MySer_Task() {
  while(1) {
    if (SerHead!=SerTail) {
      MySerial.toSer(OutBuf[SerTail++]);
      ////Serial.write(OutBuf[SerTail++]);
      if (SerTail==SER_BUF_MAX) SerTail=0;
    }

//// with heavy print-load send 2 or 3 characters:
////    if (SerHead!=SerTail) {
////      MySerial.toSer(OutBuf[SerTail++]);
////      //Serial.write(OutBuf[SerTail++]);
////      if (SerTail==SER_BUF_MAX) SerTail=0;
////    }

    Yield(200);
  }
}
```

is resposible to send the buffer to the serial line.

8.2 Pins

digitalWrite() is painfully slow. To get it much (very much) faster you have to replace it.
The files "Pins.h" and "Pins.c" are such a replacement. To switch the built in LED on an off you write
BITSETD13; BITCLEARD13; insted of digitalWrite(13,HIGH); digitalWrite(13,LOW) and the pulsewidth will reduce

from 3.8 µs to 0.125 µs - 30 times faster!

For me it is extrem important because I use a scope or a logic analyzer to analyze timings of set and cleared bits in the tasks.
With BITSET/BITCLEAR I am able to do interrupts every 5 µs and mark them (see Interrupts) for testing with a scope.

8.3 Show Stack

How much stackspace should you give a task? If you reserve too much it will waste precious Ram. But if the stack of
a task is too small the program will crash!
In "TheProgram" ShowStack_Task prints every 5 seconds the stack of all tasks. The stacks are filled with 0x55 during initialisation.
The printout shows the usage of the stacks of the tasks:

```
Idle State  READY , RUN
Free stack space: 25
StackLen: decimal 80
Stack:
0x7bc:  be 0c 75 05 53 49 20 52 56 41 01 10 01 00 00 00 23 01 a1 50 5c 00 00 00 ac 6b e3 00 00 00
0x79c:  00 00 08 00 2d 04 00 00 02 00 00 e3 00 38 00 00 00 00 00 02 02 94 0a 55 55 55 55 55 55 55
0x77c:  55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55

T1  State  BLOCKED , BLK
Free stack space: 35
StackLen: decimal 90
Stack:
0x76c:  66 04 d3 04 18 0d 00 00 c8 16 00 00 40 4b 00 01 04 00 a1 50 a4 00 00 00 8c 3a e5 00 00 00
0x74c:  00 18 08 01 2d 04 00 18 02 03 00 e5 00 39 00 00 00 00 00 05 02 94 0a 55 55 55 55 55 55 55
0x72c:  55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55

T2  State  BLOCKED , DEL
Free stack space: 31
StackLen: decimal 90
Stack:
0x712:  66 04 ea 04 b4 0c 00 00 c8 16 00 00 40 4b 00 02 01 00 a1 50 54 00 00 00 f4 ba e6 00 00 00
0x6f2:  00 00 08 02 2d 04 00 30 02 06 00 e6 00 39 00 9c 00 27 00 75 02 5e 0b 76 02 f9 02 55 55 55
0x6d2:  55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55

T3  State  BLOCKED , DEL
Free stack space: 33
StackLen: decimal 80
Stack:
0x6b8:  66 04 ba 08 00 00 00 00 c8 3a 0c 00 00 00 ff 03 00 02 a1 50 cc 00 00 00 f0 b3 d9 00 00 00
0x698:  78 03 08 03 2d 04 06 6d 00 a1 06 55 00 15 00 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
```

```
0x678:  55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55

T4  State  READY , RDY
Free stack space: 49
StackLen: decimal 90
Stack:
0x668:  66 04 18 02 00 fa 08 00 00 00 00 54 46 55 00 00 00 01 00 63 06 a1 50 bc 00 00 00 20 ee e2
0x648:  00 ff 02 01 18 08 04 2d 04 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
0x628:  55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55

SHS State  READY , RUN
Free stack space: 33
StackLen: decimal 110
Stack:
0x60e:  66 04 18 02 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 00 00 30 44 0a 90 0d 00 00
0x5ee:  01 18 22 00 05 00 f6 05 a1 50 00 00 00 00 d0 63 eb 00 00 00 ff 06 f7 03 08 05 2d 04 00 78
0x5ce:  00 eb 00 3a 00 00 00 00 00 02 02 94 0a 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
0x5ae:  55 55 55 55 55 55 55 55 55 55 55 55 55 55

MyS State  READY , RDY
Free stack space: 33
StackLen: decimal 80
Stack:
0x5a0:  66 04 66 05 00 00 00 00 1c 27 00 00 64 00 00 06 18 02 a1 50 00 00 00 00 d4 d6 ec 00 00 00
0x580:  78 03 08 06 2d 04 00 90 02 ff 00 ec 00 3b 00 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
0x560:  55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55

Dbg State  BLOCKED , BLK
Free stack space: 28
StackLen: decimal 110
Stack:
0x550:  66 04 46 08 f4 65 a6 00 75 02 0c 18 04 00 76 02 c6 02 a1 50 54 00 00 00 34 1a bb 00 00 00
0x530:  14 00 08 07 2d 04 0a c1 04 94 0a 55 55 cb 05 04 00 76 02 42 05 23 05 97 01 02 00 44 05 01
0x510:  94 0a 12 23 00 00 08 00 38 05 39 39 40 04 12 02 94 0a 55 55 55 55 55 55 55 55 55 55 55 55
...
...
```

.
The stackspace for a task is shown with:
Free stack space: xx
This shows, how much stackspace is available.

It helps to finetune the size of the stacks of the tasks.

8.4 Functions to check stackspace

There are two functions to check the stack:
- **myStackFree()** Returns the free stackspce for the running task.

- **stackFree(ID)** Returns the free stackspce for the task with number ID.
Idle has the ID 0 and all other Tasks get incrementing IDs in TaskInit()

So it is easy to check the stackspace even when DBG is not enabled!<br

Warning

Do not leave these functions in the final code because they are time consuming.

8.5 WatchDogTimer

One important thing is to test if a task has enough stackspace. But there is another point which is very critical:
How long does a task run without cooerative Yield / Delay ?
It is fine to know that we have for instance 16000 taskswitches per second (as Task 3 tells us) - but what ist the worst case - and how can we detect it?
Here comes the help of the WatchDogTimer! Here Timer-1 is used. The function TaskSwitch increments SwitchCount.
The WDT tests in it's ISR, if SwitchCount was incremented since the last call of WDT.
The WDT interrupts WDT_VALUE $*$ 0.5 µs. If WDT_VALUE is set to 400 the the interrupt occurs every 200 µs.
If the worst case happens and the WDT is enabled (it gives a message after the Init-Message) we get a message like:

ID 7
ERROR WDT: no Yields !

That means: Task #7 (what is our Debugger here) caused the Message. (No wonder - if we press the Debug-Button the program
stops) We can trim down WDT_VALUE until we get the first message from the WDT.

Note

It is a good practice to run a task as the only one (well, together with idle) and try to find the lower time limit

with WDT. Together with SHOWST you get the 2 most important values for a task:

- Stacksize
- longest time without cooperation

Attention!
hallo The time you evaluated with with WDT does nothing tell you, how often this task is called! It just says something about the longest time without cooperation a task may need.

Achtung !
The time you evaluated with with WDT does nothing tell you, how often this task is called! It just says something about the longest time without cooperation a task may need.

Achtung !
The time you evaluated with with WDT does
nothing tell you, how often this
task is called! It just says
something about
the longest time without cooperation a task may need.

sad as s as sad as as

Side Effects:

saoiuoipoipoipoi poipoipoi

And how can I get the time since the last call? Just remember the last call in a variable and micros().

d sad dsasad sad

sdsadsad

Note

It's up to you to use the WDT (16 Bit Timer1) for your own purposes. Just write a new:
ISR(TIMER1_COMPA_vect)
But be warned: do NOT use Serial.print like I did it. Here it is the end of the program - and that is ok!
(see Interrupts)

8.6 Interrupts

Interrupts were a kind of multitasking even in older no-multitasking OS - for instance MSDOS.
And in embedded systems they are very important despite the underlying OS.
Processors become faster and faster and a lot of embedded systems are running Linux now.
But sometimes Linux is not good enough. The reason: Interrupt Latency.
Linux preempt_RT is a way to reduce interrupt latency - but a good prepared Arduino is much faster !!!
Good prepared means:

- Serialized Serial output
If you write a string using the normal Arduino library it can take milliseconds.
MYSER send max. 3 characters in a row and wait for at least 100µs. (I use 500000 baud). That is ok for all my needs.
The output of the character are done in this way:

```
inline void toSer( char c) {
      // wait until Transmitter empty
      while ( !( UCSR0A & (1<<UDRE0)) ); // theoretically max. 20 µs - but fetching the next chara
      // send one byte                   // makes it less
      UDR0 = (uint8_t)c;
}
```

This is as far as I know the fastest way to transmit characters !

- Very short times where interrupts are disabled during the taskswitches
If you change the stackpointer the interrupts must be disabled.
The duration for this in CoopOS_Stack_MT is $<$ 1 µs.

The WDT can be used for other purposes. Then you have to write you own ISR().
Here is an example: (tested with TheProgram ALL options enabled!)

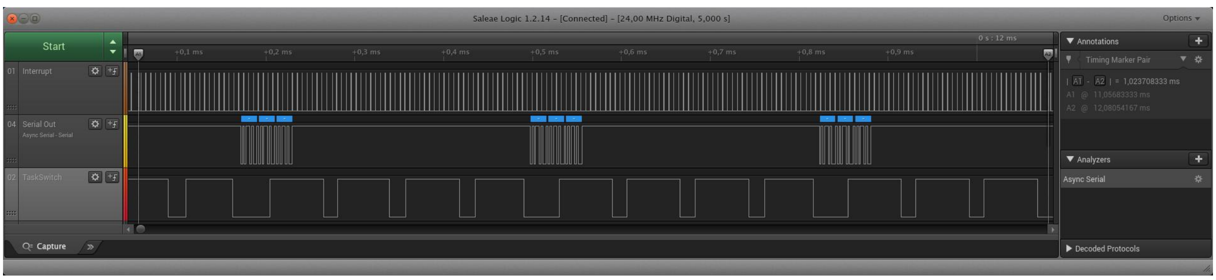
```
ISR(TIMER1_COMPA_vect)
{
  BITSETD4;
  Flag=1;
  BITCLEARD4;
  return;
```

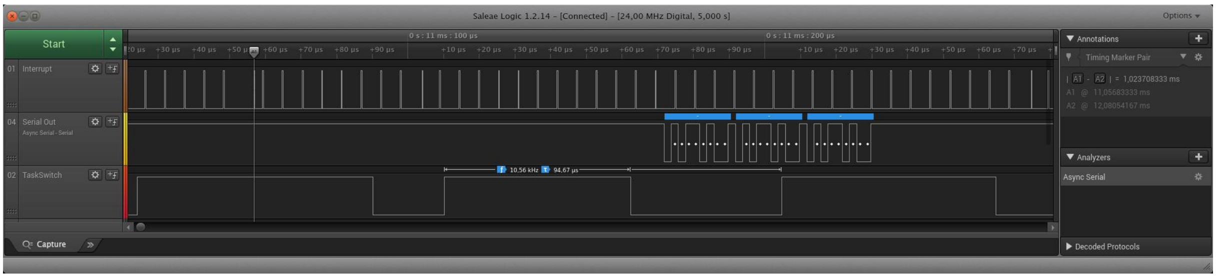
Together with

```
#define WDT
#define WDT_VALUE 10            // = 5 µs
```

we get an **interrupt every 5 µs**. Unbelievable?

Here is the test:





The second image shows a curious latency (left marker). But it's not the responsibility of the serial output nor of the task switches. A closer look shows: this is the timer overflow interrupt of Timer-0 counting milliseconds!

Let us look at the result: (without TRACE_ON)

Note

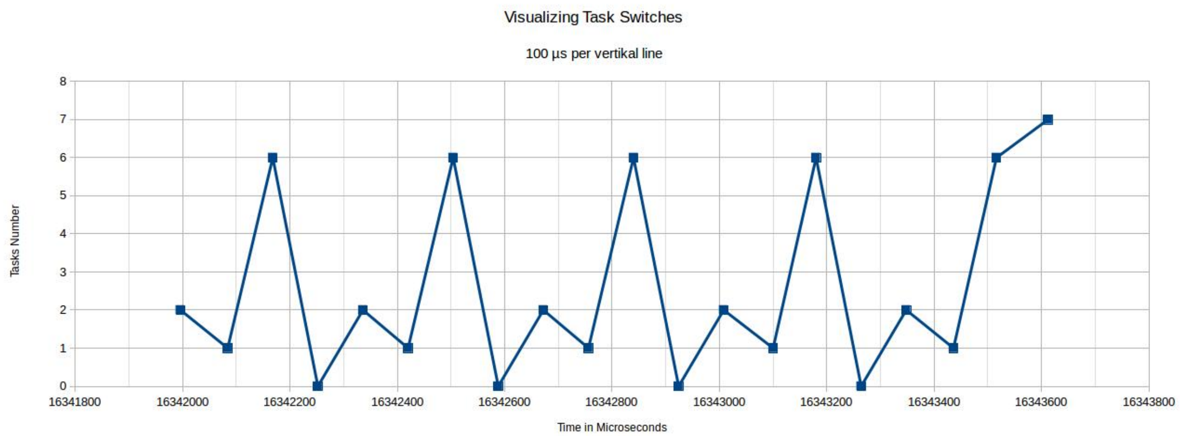
> **We have a lot of serial output - much more than you can read**
> **We are reading incoming bytes (up to 60 in a burst) without loosing charactersa and write a long line for each character**
> **We have incredibil 200.000 (in words: Twohundredthousand) intrerrupts per second setting a flag**
> **We have about 13.000 taskswitches / s**
> **We have 6.500 On/Off switches of the LED done by a signal from Task2 to Task1**
> **We 8 Tasks running.**
> **We have precise timing**
>
> **And: we have done it with a simple Arduino AVR 328p !**

8.7 Debug / Breakpoints

!!! TRACE_ON must be enabled to see the last 20 called tasks!!! One thing we are all missing in the Arduino-IDE are Breakpoints! In a multitasking environment it get more painfully.
I implemented them and you can you can use them in two flavors.
1) Just insert the line BREAKPOINT; in any task. When the BREAKPOINT is reached the program will stop and the time
(in microseconds) for the last 20 task switches are shown: (the secondon number is the number of the task, the third number is the difference in microseconds to the previous taskswitch)
After the name you see the free stackspace of the task:
if TRACE_ON is disabled the Debugger / BREAKPOINTs will work but you get no information about time and called tasks.
That makes sense because the tracing is time consuming and costs up 50% of performance!

```
----------- HERE IS THE DEBUGGER -------------
Latency(µs)    100
Stopped in     MyS
StackPt was    0x733
Last Tasks: µs ID deltaT Name FreeStack
16341996 2 0 T2  31
16342084 1 88 T1  31
16342168 6 84 MyS 33
16342252 0 84 Idle 25
16342336 2 84 T2  31
16342420 1 84 T1  31
16342504 6 84 MyS 33
16342588 0 84 Idle 25
16342672 2 84 T2  31
16342756 1 84 T1  31
16342840 6 84 MyS 33
16342924 0 84 Idle 25
16343008 2 84 T2  31
16343100 1 92 T1  31
16343180 6 80 MyS 33
16343264 0 84 Idle 25
16343348 2 84 T2  31
16343436 1 88 T1  31
16343516 6 80 MyS 33
16343612 7 96 Dbg 28
```

If you copy the output to for instance LibreOffice Calc then you can visualize the timing of taskswitches:



This is a task time window of about 1.6 ms !

Here we can see:

- **Task Switches are done from 80-99 µs (except DBG) => ∼ 10000 Task Switches per second.**
  Without TRACE_ON you can reach 25000 Task Switches /s !.

- **How the blinking is done here.**
  Task 2 switches the LED on and resumes the blocked Task 1. Task 1 switches the LED off and stops itsself.
  The time from Task 1 to Task2 is about 80 µs.

- **Last Task running**
  Task 7 is BREAKPOINT wich is called from Task 4. Task 4 started with 5 seconds delay set in Init(↩ Task4).

- **Task 0 is Idle and it is called some times. This shows, that the program does not use the full capacity of the processor**
  at this moment.

But the program has stopped. How would it be, if we can resume at the point where BREAKPOINT stopped the program?
Here comes the 2. flavor of using BREAKPOINT :

There is an Interrupt routine in the program connected to the falling edge of pin D2.
If you mount a switch between D2 an Ground this switch stops/resumes the program. You may press the switch at any time
analyse the output of the debugger and resume again. Here is an example:

```
T4 State  BLOCKED , DEL
StackLen: decimal 80
Stack

////                              Here the program is resumed with the Breakpoint-Button
////                              In the middle of an output: "Stack" is written, ":" comes after
 ----------- HERE IS THE DEBUGGER -------------
Latency(µs)   148
Stopped in    T1
StackPt was   0x771
Last Tasks:
51305568 1 0 T1
51305648 5 80 T5
51305724 6 76 T6
51305816 0 92 Idle
51305928 2 112 T2
51306008 1 80 T1
51306084 5 76 T5
51306188 0 104 Idle
51306296 2 108 T2
51306376 1 80 T1
```

```
51306456 6 80 T6
51306552 0 96 Idle
51306660 2 108 T2
51306740 1 80 T1
51306820 5 80 T5
51306896 0 76 Idle
51307008 2 112 T2
51307088 1 80 T1
51307180 5 92 T5
51307260 7 80 T7

////                               Here the program is resumed with the Breakpoint-Button

:
0x69f:  82 04 18 02 00 c0 08 d8 0a 00 00 d8 10 01 18 64 00 01 00 18 9a 06 98 50 d8 0a 00 00 d8 10
0x67f:  a0 86 00 00 01 03 64 18 49 04 00 60 02 02 02 0b 03 c2 00 55 55 55 55 55 55 55 55 55 55 55
0x65f:  55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55

T5 State  READY , RUN
StackLen: decimal 100
Stack:
0x64f:  82 04 18 02 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 00 00 30 e0 09 d8 0a 00 00
0x62f:  01 18 05 0fb 24 04 03 37 06 98 50 d8 0a 00 00 fc 03 01 18 0a 00 00 00 fa 06 5a 18 49 04 0
0x60f:  08 02 b7 03 ed 00 00 00 00 00 7c 02 30 0a 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
0x5ef:  55 55 55 55
```

We interrupted Task 5 while prinitng the stacks of the tasks, analyzed the output and resume the program.

This is a valuable tool of CoopOS_Stack_MT !<//b>

<< **Intertask Communication**
>> **The Program**

9 The Program

The Program is a working example and could be used as a template for your own programs.
We will going through the whole program to deliver a complete explanation.
Warning: This is NOT usable with "#include TaskSwitchDemo.h" Use "TheProgram.zip" and extract it to your Arduino sketch folder. No special libraries are used.
All "#include" are done from the local directory and you find all files as tabs when you open the sketch.

At the top you find "#include Pins.h". This is for fast digital IO ()s.a)

The next lines define:

- **STACKALLOC**
  This is the total stack reserved for ALL tasks
  You get a message if this amount is to small for your program and the program stops.

- **IDLE_STLEN**
  The lenght of the "Idle" task that must be present at all programs. Idle is call if no other task is READY.

- MAX_TASKS:
  The maximal number of tasks wich can be defined - including "Idle"

- **TRACE_ON**
  This program has a "Debug" module. If enabled "Debug" breaks the program when it comes to a "BREAK↩POINT" in the source
  or when a Button mounted between D2 and Ground is pressed.
  Optionally trace of the last 20 called tasks are shown. That is enabled with "#define TRACE_ON".
  "TRACE_ON" does not make sense without "__DEBUG".
  The performance is reduced to about 50% when TRACE_ON is enabled.
  __DEBUG without TRACE_ON can make sense.

- **__MYSER**
  MySer.h is a module which spread serial output. That can be important when high frequency task switches should
  be performed. Serial.print (long text) can delay the multitask for one or more ms.
  With __MYSER enabled only 2 characters out of a buffer are sent with a delay of 100 µs.
  It is recommended to let it enabled.

- **__DEBUG**
  With __DEBUG BREAKPOINTs in source and BREAK with Button are enable. See TRACE_ON

- **__SHOWST**
  With __SHOWST on the stack of all tasks are displayed every 2 seconds. The stacks of the tasks are prefilled with 0x55
  and the ShowStack makes it easy to see, how much of the reserved stack of a task is used.

The **DoPrep.h** defines the structur for __DEBUG and BREAKPOINT if they are needed and redirects __MYSER (s.a)

LED_on and LED_off are defined using Pins.h to make them faster,
Some Globas are defined and then comes the definition of all
Tasks:

- **Idle**
  Idle must always be present and MUST have function name: Idle !!!
  It is allways the first task which is call if no other task is READY.
  And it is the first task started by **StartMultiTasking()** later on

- **Task1** The first action of Task1 is to stop itself. It is Task2 to resume Task1 again.

- **Task2** Task2 switches the LED on and sets Tasks1 to READY and Delay(200) for 200 microsseconds. As Task1 has the
  highest priority it should run as next - and it does after about 55 µs. Task1 switches the LED off set itself "dormant"
  again. This is to show how fast you can react on a "signal" that Task2 sends to Task1. Good to be seen with a scope.

- **Task3** Task3 shows the values of IdleCount, BlinkCount and SwitchCount every second and resets these values. If ShowStack is active
  in this moment which shows all stacks, Task3 3 will show "missing" instead of the values - because they are not so important.
  But it is possible to wait until ShowStack is ready with the output with this line.
  while (DisplayUsed) Yield(0);
  Then the variable DisplayUsed is used like a "mutex" !

- **Task4** Task4 just prints its stack pointer and tests if there is a character available at the serial line and prints it if available.
  Well, not exact. It call Task4_fun to do that.
  It demonstrates, that you may use al the stuff like stopMe, Yield(0), Delay(x) ... in called functions - not only in Tasks.

- **ShowSt, MySer and Debug**
  are defined here as tasks if they are enable at the top of program. Have in mind that they nedd stackspace. The total allocated stack
  may be reduced if you don't need them.

Now we come to **setup()** where the tasks are inited and started:

- **Serial.begin(500000)**
  Sometimes I have to smile when I see some examples beginning with **Serial.bin(9600)**. That reminds me using my nearly 40 years
  old terminal TelVideo 925 - but that could use 19200 either ;)
  For fast multitasking programming a high speed serial line is very important to save precious time.
  500000 baud works without any problem for me with all microprocessors.

- **Serial.begin(500000)**
  Sometimes I have to smile when I see some examples beginning with **Serial.bin(9600)**. That reminds me using my nearly 40 years

old terminal TelVideo 925 - but that could use 19200 either ;)
For fast multitasking programming a high speed serial line is very important to save precious time.
500000 baud works without any problem for me with all microprocessors.

- **MySerial.setSerial**
  redirects MySerial.print output to the serial line together with the task included by "MySer.h" if enabled.
  Instead of redirecting it to Serial you may redirect it to any Stream you want!

- **StackPrepare()**
  is defined in Task.h and reserve the amount of stackspace defined by STACKALLOC at the top of program.
  As next are the used IO pins are defined and the reserved stackspace is filled with 0x55.

- **TaskInit()**
  is used to inform the system, which functions are used as tasks.
  The format is:

```
////          NAM      FUNC             STCK PRIO  DLY        STATE
  TaskInit("T1 ",  Task1,            90, 104,   0,         READY);
                |        |                |    |       |           |
            NAM is the const char* to a name           |           |
                         |                |    |       |           |
            FUNC is the name of the task function the source|
                                          |    |       |           |
            STCK is the amount of stack used by this task          |
                                               |       |           |
            PRIO is the priority of this task          |           |
                                                       |           |
            DLY is the delay for this task in microseconds         |
                                                                   |
            STATE can be READY or BLOCKED (if DLY > 0)             |
```

  TaskInit() prepares the structur of the TaskBlocks of Tasks[]. The task are NOT RUNNIG after TaskInit() !

- **__SHOWST, __MYSER and _DEBUG** are not added with their TaskInit(), if they are enabled.

Then the free ram left is shown.
Next the interrupt for break button at D2 is defined, if used.

- **StartMultiTasking()**
  All preparation is done now. We are ready to start the construct. With StartMultiTask() at first Idle is started which in turn starts all other tasks.
  StartMultiTasking() will never return! Even loop() is not called!
  Good luck!

  BTW: doxygen is an amazing tool to document programs. But it makes the source some kind of unreadable ;)
  Therfore I think it is the best to use the source provided as ZIP-files, which do not include these doxgen comments, in your Arduino IDE and look at this documentation in a browser in a second window.

10 Conclusion

A lot of programmers don't like cooperative multitasking. Maybe the old Windows version are responsible ;)
But in an embedded system you do not have to start programs from other companies. And if a task runs wrong if may crash
the whole system. That is true for RTOS's on an Arduino either because it has no MMU !

How can we detect the longest timeslice a task will use?
For that we can use a WatchDogTimer. It fires WDT_VALUE∗0.5 µs.
With this tool we can proof, that in "TheProgram" a task never uses more than 100 µs up to the next taskSwitch

- despite heavy serial output.

Another point of criticism is that systems like this most of the time do task switching. That is intended!
Systems like the presented are not intended to calulate pi for thousands of digits - they are good for fast reaction, do "simultaniously" at lot of short task-parts. This is typically for get/set AD/DA-values, running serovs and stepper motors ... etc.

My criticism is: Since the old days processors became thousands and even millions times faster !
But a lot of RTOS's seem to be glued to the 1ms tick timer.br> And great things have been made without RTOS's.
CoopOS demonstrates that a taskswitch at least every 100µs can be done with a small 8 bit microprocessor like an AVR 328p.
If that can be called an RTOS is your decision - but fact is: it is really fast!

And if you look behind the scenes with open eyes you will find cooperative multitasksing / coroutines becomming more
and more modern programming lanuages: C++, Python, Javascript...
The famous node.js is cooperative -
Here an example: The **Televideo 925 Terminal** works with 19200 baud. They invented the Ansii-control Esc-sequences
to move the cursor, do blinking and so on. There are a lot of such commands and we use them until now in Linux command windows and terminal emulation programs like screen.
These masterpieces are n**early 40 years old and where buildt with a 6502 processor with 2k ram.**
The 328p of the Arduino has 2k ram either, more program space, has much more registers and is 16 times faster than the 6502
But I think only a very few programmers worldwide would be able to rebuild a Televideo 925 with an Arduino,
Fact is: You can build such things without an RTOS.
That does not mean, that RTOS's are obsolete. Most of the challanges of modern programming are not possible without them.
But sometimes other solutions are better - especially for small system.
And up to **200000 interrupts - sending signals** - while running a multitasking system is not so bad ;)
As I have told I prefer my **CoopOS**. It does the same cooperative multitasking - but faster, because there is no stack switching. Thats why it could be faster.
But it has some sideeffects you have to learn to deal with. It needs more learning.

I hope TheProgram and the Demos together with the tools and the documentation enable you to make your own programs.

Tasks are the one and only method to break down complex programs to easy understandable modules and helps you building
blocks of yout different tasks.
The full compatibilty with existing libraries helps to start - no special device drivers are needed.
See how much fun it is to demerge existing programs to build simple tasks!

Good luck and have fun with CoopOS_Stack_MT !

11 Class Index

11.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

12 File Index

12.1 File List

Here is a list of all files with brief descriptions:

13 Class Documentation

13.1 mySerial Class Reference

```
#include <MySerial.h>
```

Public Member Functions

- void setSerial (Stream *streamObject)
- write (byte b)
- void toSer (char c)
- void write (char c)
- void println ()
- void print (char *str)
- void println (char *str)
- void print (unsigned int i)
- void println (unsigned int i)
- void print (uint8_t i)
- void print (uint8_t i, uint8_t n)
- void print (unsigned int i, uint8_t n)
- void println (unsigned int i, int n)
- void println (uint8_t i)
- void print (int i)
- void println (int i)
- void print (unsigned long i)
- void println (unsigned long i)
- void print (long i)
- void println (long i)
- void print (float i)
- void println (float i)
- char read ()
- bool available ()
- void flush ()

13.1.1 Detailed Description

Definition at line 118 of file MySerial.h.

13.1.2 Member Function Documentation

13.1.2.1 bool mySerial::available ( ) [inline]

Definition at line 259 of file MySerial.h.

```
259                       {
260         return mystream->available();
261       }
```

13.1.2.2 void mySerial::flush ( ) [inline]

Definition at line 263 of file MySerial.h.

```
263                   {
264         mystream->flush();
265       }
```

13.1.2.3 void mySerial::print (char * *str*) [inline]

Definition at line 154 of file MySerial.h.

```
154                             {
155       int len;
156       char *pt;
157         pt=str;
158         while(*pt) {
159            write(*pt++);
160            Yield(10);
161         }
162       }
```

13.1.2.4 void mySerial::print (unsigned int *i*) [inline]

Definition at line 171 of file MySerial.h.

```
171                                  {
172       char *pt;
173         pt=_itoa(i);
174         print(pt);
175       }
```

13.1.2.5 void mySerial::print (uint8_t *i*) [inline]

Definition at line 182 of file MySerial.h.

```
182                             {
183       char buf[20];
184         itoa(i, buf, 10);
185         print(buf);
186       }
```

13.1.2.6 void mySerial::print (uint8_t *i*, uint8_t *n*) [inline]

Definition at line 188 of file MySerial.h.

```
188                                         {
189     char buf[20];
190       itoa(i, buf, n);
191       print(buf);
192     }
```

13.1.2.7 void mySerial::print (unsigned int *i*, uint8_t *n*) [inline]

Definition at line 194 of file MySerial.h.

```
194                                              {
195     char buf[20];
196       itoa(i, buf, n);
197       print(buf);
198     }
```

13.1.2.8 void mySerial::print (int *i*) [inline]

Definition at line 210 of file MySerial.h.

```
210                          {
211     char *pt;
212       pt=_itoa(i);
213        print(pt);
214     }
```

13.1.2.9 void mySerial::print (unsigned long *i*) [inline]

Definition at line 222 of file MySerial.h.

```
222                                    {
223     char *pt;
224       pt=_ltoa(i);
225        print(pt);
226     }
```

13.1.2.10 void mySerial::print (long *i*) [inline]

Definition at line 234 of file MySerial.h.

```
234                           {
235     char *pt;
236       pt=_ltoa(i);
237       print(pt);
238     }
```

13.1.2.11 void mySerial::print (float *i*) [inline]

Definition at line 245 of file MySerial.h.

```
245                            {
246       print(_ftoa(i,2));
247     }
```

13.1.2.12 void mySerial::println ( ) [inline]

Definition at line 150 of file MySerial.h.

```
150             {
151       write('\n');
152     }
```

13.1.2.13 void mySerial::println (char * *str*) [inline]

Definition at line 164 of file MySerial.h.

```
164                   {
165       print(str);
166       println();
167     }
```

13.1.2.14 void mySerial::println (unsigned int *i*) [inline]

Definition at line 177 of file MySerial.h.

```
177                      {
178       print(i);
179       println();
180     }
```

13.1.2.15 void mySerial::println (unsigned int *i*, int *n*) [inline]

Definition at line 200 of file MySerial.h.

```
200                           {
201       print(i,n);
202       println();
203     }
```

13.1.2.16 void mySerial::println (uint8_t *i*) [inline]

Definition at line 205 of file MySerial.h.

```
205                   {
206       print(i);
207       println();
208     }
```

13.1.2.17 void mySerial::println (int *i*) [inline]

Definition at line 216 of file MySerial.h.

```
216                 {
217       print(i);
218       println();
219     }
```

13.1.2.18 void mySerial::println (unsigned long *i*) [inline]

Definition at line 228 of file MySerial.h.

```
228                                 {
229       print(i);
230       println();
231     }
```

13.1.2.19 void mySerial::println (long *i*) [inline]

Definition at line 240 of file MySerial.h.

```
240                         {
241       print(i);
242       println();
243     }
```

13.1.2.20 void mySerial::println (float *i*) [inline]

Definition at line 249 of file MySerial.h.

```
249                          {
250       print(i);
251       println();
252     }
```

13.1.2.21 char mySerial::read ( ) [inline]

Definition at line 255 of file MySerial.h.

```
255                 {
256       return mystream->read();
257     }
```

13.1.2.22 void mySerial::setSerial (Stream ∗ *streamObject*) [inline]

Definition at line 124 of file MySerial.h.

```
125     {
126       mystream=streamObject;
127     }
```

13.1.2.23 void mySerial::toSer (char *c*) [inline]

Definition at line 134 of file MySerial.h.

```
134                              {
135       // wait until Transmitter empty
136       while ( !( UCSR0A & (1<<UDRE0)) ) {
137         Yield(0);
138       }
139       // send one byte
140       UDR0 = (uint8_t)c;
141     }
```

13.1.2.24 mySerial::write (byte *b*) [inline]

Definition at line 130 of file MySerial.h.

```
130                 {
131         //mystream->write(b);
132     }
```

13.1.2.25 void mySerial::write (char *c*) [inline]

Definition at line 143 of file MySerial.h.

```
143                     {
144         //mystream->write(c);
145        OutBuf[SerHead++]=c;
146        if (SerHead==SER_BUF_MAX) SerHead=0;
147     }
```

The documentation for this class was generated from the following file:

- MySerial.h

13.2 task Class Reference

`#include <Task.h>`

Public Attributes

- char ∗ name
- FuncPt function
- uint8_t prio
- uint16_t sp_save
- uint16_t task_stack
- uint16_t stackLen
- char new_task
- unsigned long lastCalled
- unsigned long Delay
- State state
- State2 state2

13.2.1 Detailed Description

This is the structure of a TaskBLock

Definition at line 20 of file Task.h.

13.2.2 Member Data Documentation

13.2.2.1 unsigned long task::Delay

Delay in µs

Definition at line 29 of file Task.h.

13.2.2.2 FuncPt task::function

Pointer to function to execute as task

Definition at line 22 of file Task.h.

13.2.2.3 unsigned long task::lastCalled

Last time called in µs

Definition at line 28 of file Task.h.

13.2.2.4 char * task::name

Name of a task as const char*

Definition at line 21 of file Task.h.

13.2.2.5 char task::new_task

Allways true at the beginning except Idle

Definition at line 27 of file Task.h.

13.2.2.6 uint8_t task::prio

Priority of task, only relative values are used

Definition at line 23 of file Task.h.

13.2.2.7 uint16_t task::sp_save

Value of stack pointer saved before switched

Definition at line 24 of file Task.h.

13.2.2.8 uint16_t task::stackLen

Length of reserved stackspace for this task

Definition at line 26 of file Task.h.

13.2.2.9 State task::state

State of task (READY, BLOCKED)

Definition at line 30 of file Task.h.

13.2.2.10 State2 task::state2

Substate of task (sa.)

Definition at line 31 of file Task.h.

13.2.2.11 uint16_t task::task_stack

Pointer to first Stackpointer of this task

Definition at line 25 of file Task.h.

The documentation for this class was generated from the following files:

- Task.h
- TaskSwitchDemo.h

14 File Documentation

14.1 calc.jpg File Reference

14.2 CoopOS_Stack_MT_Nano.ino File Reference

```
#include "Pins.h"
#include "TaskSwitch.h"
#include "DoPrep.h"
#include "ShowSt.h"
#include "MySer.h"
```

Macros

- #define STACKALLOC 710
- #define IDLE_STLEN 80
- #define MAX_TASKS 8
- #define TRACE_ON
- #define WDT

  -------------------------------— CONFIG WDT---------------------------------------— / WDT MUST be enabled before including "TaskSwitch.h / WDT should only be enabled to test the MAX time (worst case) a task needs from / Yiedl(0) / Delay(n) until the next Yiedl(0) / Delay(n) / s. TaskSwitch.h for some examples
- #define WDT_VALUE 199
- #define __MYSER
- #define __SHOWST
- #define LED_On BITSETD13;
- #define LED_Off BITCLEARD13

Functions

- void Idle (void)
- void Task1 (void)
- void Task2 (void)
- void Task3 (void)
- void Task4_fun ()
- void Task4 (void)
- void setup ()
- void loop ()

    /code>

Variables

- volatile char DisplayUsed =0
- unsigned int BlinkCount
- uint8_t DbgHandle

14.2.1 Macro Definition Documentation

14.2.1.1 #define __MYSER

Definition at line 64 of file CoopOS_Stack_MT_Nano.ino.

14.2.1.2 #define __SHOWST

Definition at line 77 of file CoopOS_Stack_MT_Nano.ino.

14.2.1.3 #define IDLE_STLEN 80

Definition at line 25 of file CoopOS_Stack_MT_Nano.ino.

14.2.1.4 #define LED_Off BITCLEARD13

Definition at line 90 of file CoopOS_Stack_MT_Nano.ino.

14.2.1.5 #define LED_On BITSETD13;

Definition at line 89 of file CoopOS_Stack_MT_Nano.ino.

14.2.1.6 #define MAX_TASKS 8

Definition at line 26 of file CoopOS_Stack_MT_Nano.ino.

14.2.1.7 #define STACKALLOC 710

#include "Doxyfile.dox" // extension mapping dox=md does the trick ;) in ...-xx

Definition at line 24 of file CoopOS_Stack_MT_Nano.ino.

14.2.1.8 #define TRACE_ON

Definition at line 35 of file CoopOS_Stack_MT_Nano.ino.

14.2.1.9 #define WDT

-------------------------------— CONFIG WDT--— / WDT MUST be enabled before including "TaskSwitch.h / WDT should only be enabled to test the MAX time (worst case) a task needs from / Yiedl(0) / Delay(n) until the next Yiedl(0) / Delay(n) / s. TaskSwitch.h for some examples

Definition at line 46 of file CoopOS_Stack_MT_Nano.ino.

14.2.1.10 #define WDT_VALUE 199

Definition at line 52 of file CoopOS_Stack_MT_Nano.ino.

14.2.2 Function Documentation

14.2.2.1 void Idle (void)

Definition at line 104 of file CoopOS_Stack_MT_Nano.ino.

```
105 {
106   while (1)
107   {
108     IdleCount++;     // only for test, you may comment out this line
109     Yield (0);
110   }
111 }
```

14.2.2.2 void loop ( )

/code>

Definition at line 324 of file CoopOS_Stack_MT_Nano.ino.

14.2.2.3 void setup ( )

Definition at line 251 of file CoopOS_Stack_MT_Nano.ino.

```
251                 {
252   Serial.begin(500000);
253
254 #ifdef __MYSER
255   MySerial.setSerial(&Serial); // Redirect Serial
256 #endif
257
258   StackPrepare();
259
260
261   if (STACK==NULL) {
262     Serial.println(F("ERROR: Not enough room for Stack !!!"));
263     while(1);
264   }
265   Serial.print(F("\n\nStack allocated: ")); Serial.println(STACKALLOC);
266   Serial.print(F("Free Ram now    : ")); Serial.println(freeRam());
267   Serial.println();
268
269   pinMode(4,OUTPUT); // Yield
270   pinMode(5,OUTPUT); // Task3
```

```
  pinMode(6,OUTPUT); // TaskSwitch
  pinMode(7,OUTPUT); // WDT-IRQ
  pinMode(8,OUTPUT); // Task1,2

  pinMode(2,INPUT_PULLUP);

  // fill stack with 0x55 as markers to test stack usage
  uint8_t *pt=STACK;  while (pt>StackLow) { *pt--=0x55;}

  Serial.println(F("----------- Init start -----------"));

  TaskInit("T1 ",  Task1,          70, 105,   0,        READY);
  TaskInit("T2 ",  Task2,          80, 106,   0,        READY);
  TaskInit("T3 ",  Task3,          80, 100,   0,        READY);
  TaskInit("T4 ",  Task4,         100, 102,   5000000,  BLOCKED);  // starts after 5 seconds



#ifdef __SHOWST
  TaskInit("SHS",  ShowStack_Task, 120, 103,   0,        READY);
#endif


#ifdef __MYSER
  TaskInit("MyS", MySer_Task,      80,  103,   0,        READY);
#endif

#ifdef __DEBUG
  DbgHandle=
  TaskInit("Dbg", Dbg_Task,        100,  110,   0, READY);
#endif



  Serial.println(F("----------- Init ready -----------"));
  Serial.print(F("\nFree Ram now   : ")); Serial.println(freeRam());

#ifdef __DEBUG//For Debugger-Button:
  cli(); //For Debugger-Button: Enable FALLING interrupt at D2
  pinMode(2, INPUT_PULLUP);
  attachInterrupt (digitalPinToInterrupt (2), IRQ_Answer, FALLING);  // attach interrupt handler for D2
  sei();
#endif


  StartMultiTasking();
  Serial.println(F("Returned from StartMultiTasking()"));
  Serial.flush();
  while(1);
}
```

14.2.2.4 void Task1 (void)

Definition at line 117 of file CoopOS_Stack_MT_Nano.ino.

```
{
  while (1)
  {
    stopMe();                                      // activated by next Yield/ Delay

    LED_Off;                                       // ~40-50 µs since Task2
    BITCLEARD8;

    BlinkCount++;
    Yield(0);                                      // Task1 has been stopped here
  }
}
```

14.2.2.5 void Task2 (void)

Definition at line 133 of file CoopOS_Stack_MT_Nano.ino.

```
{
  while (1)
  {
    BITSETD8;
    LED_On;
    BlinkCount++;
    Tasks[1].state = READY;                              // Start Task1
    Delay(200);                                          // <-------- NOT 0
  }
}
```

14.2.2.6 void Task3 (void)

Definition at line 147 of file CoopOS_Stack_MT_Nano.ino.

```
{
unsigned char missing;
  while (1)
  {
    BITSETD5;

    while (DisplayUsed) Yield(100);
    if (DisplayUsed==0) {
      DisplayUsed=1;
      MySerial.print("Task 3 :");
      Yield(100);
      if (missing==0) {                                  // Total output: 100 µs
        MySerial.print(" IdleCount/s: ");
        Yield(100);

        MySerial.print(IdleCount);
        Yield(100);
        MySerial.print(", BlinkCount/s: ");
        Yield(100);
        MySerial.print(BlinkCount);
        Yield(100);
        MySerial.print(", SwitchCount/s: ");
        Yield(100);
        MySerial.println(SwitchCount);
        Yield(100);
      }
      else {
        missing=0;
        MySerial.println("missing values");
      }

      DisplayUsed=0;

    }
    else missing=1;
    IdleCount=0;
    BlinkCount=0;
    SwitchCount=0;


    BITCLEARD5;
    Delay(1000000);
  }
}
```

14.2.2.7 void Task4 (void)

Definition at line 220 of file CoopOS_Stack_MT_Nano.ino.

```
{
  while (1)
  {

    Task4_fun();                                              // Test Yield in a called function
//    BITSETD5;
//    if (DisplayUsed==0) { MySerial.print("---------Task 4, SP:"); MySerial.println(SP);}
//    BITCLEARD5;
//    Delay(100000);
  }
}
```

14.2.2.8 void Task4_fun ( )

Definition at line 194 of file CoopOS_Stack_MT_Nano.ino.

```
                {
    //BITSETD5;
    if (DisplayUsed==0) {
      DisplayUsed=1;
      MySerial.print("---------Task 4, SP: 0x");
      Yield(0);
      MySerial.println(SP,16);
      DisplayUsed=0;
    }
    //BITCLEARD5;
    Delay(100000);

    while (MySerial.available()) {
      char ch = MySerial.read();
      //MySerial.flush();
      Serial.print(F("\n---------------------- Serial input: "));  Serial.println(ch);

#ifdef __DEBUG
      BREAKPOINT;
#endif
    }

}
```

14.2.3 Variable Documentation

14.2.3.1 unsigned int BlinkCount

Incremented by task 1

Definition at line 95 of file CoopOS_Stack_MT_Nano.ino.

14.2.3.2 uint8_t DbgHandle

Handle for Debug = Waked up by interrupt

Definition at line 96 of file CoopOS_Stack_MT_Nano.ino.

14.2.3.3 volatile char DisplayUsed =0

Used as mutex for the output

Definition at line 94 of file CoopOS_Stack_MT_Nano.ino.

14.3 Debug.h File Reference

14.4 Demo.jpg File Reference

14.5 Demo0.jpg File Reference

14.6 Demo_0A.jpg File Reference

14.7 DoPrep.h File Reference

```
#include "MySerial.h"
```

Macros

- #define MySerial Serial

14.7.1 Macro Definition Documentation

14.7.1.1 #define MySerial Serial

Definition at line 2 of file DoPrep.h.

14.8 Doxyfile.dox File Reference

14.9 inter1.jpg File Reference

14.10 inter2.jpg File Reference

14.11 MT1.jpg File Reference

14.12 MySer.h File Reference

14.13 MySerial.h File Reference

Classes

- class mySerial

Macros

- #define SER_BUF_MAX 200

Functions

- void Yield (unsigned long mics)

    Yield() is the Scheduler.
- char ∗ _itoa (unsigned int l)

    "_itoa"
    This is the selfmade conversion from unsigned int to ascii-string
    digits are the number of digits behind the"."
- char ∗ _ltoa (unsigned long l)

    "ltoa"
    This is the selfmade conversion from unsigned long to ascii-string
    digits are the number of digits behind the"."
- char ∗ _ftoa (double f, int digits)

    "ftoa"
    This is the selfmade conversion from float to ascii-string
    digits are the number of digits behind the"."

Variables

- volatile int SerHead
- volatile int SerTail
- char OutBuf [SER_BUF_MAX]
- mySerial MySerial

14.13.1 Macro Definition Documentation

14.13.1.1 #define SER_BUF_MAX 200

Definition at line 6 of file MySerial.h.

14.13.2 Function Documentation

14.13.2.1 char∗ _ftoa (double *f,* int *digits*)

"ftoa"
This is the selfmade conversion from float to ascii-string
digits are the number of digits behind the"."

Definition at line 80 of file MySerial.h.

```
80                                    {
81 static char b[31];
82 static char const digit[] = "0123456789";
83 char* p = b;
84 uint32_t i;
85
86 int d,j;
87   d=digits;
88   while (d) {
89     f*=10.0;
90     d--;
91   }
92
93   i=(uint32_t)f;
94
95   p=b+28;
96   j=0;
97   *p = 0;
98   *(p+1)=0;
99
100   do { //Move back, inserting digits as u go
101     Yield(0);
102       if (j == digits) { p--; *p=','; }
103       p--;
104       *p = digit[i % 10ll];
105       i = i/10ll;
106       j++;
107   } while(i);
108
109   return p; // return result as a pointer to string
110 }
```

14.13.2.2 char∗ _itoa (unsigned int *l*)

"_itoa"
This is the selfmade conversion from unsigned int to ascii-string
digits are the number of digits behind the"."

Definition at line 18 of file MySerial.h.

```
18                              {
19 static char b[31];
20 static char const digit[] = "0123456789";
21 char* p = b;
22 uint32_t i;
23
24
25   //i=(uint32_t)f;
26
27   p=b+28;
28   *p = 0;
29   *(p+1)=0;
30
31   do { //Move back, inserting from last digit
32       Yield(20);
33       p--;
34       *p = digit[l % 10];
35       //p--;                           // just to proof we are using this _itoa
36       //*p = ' ';
37       l = l/10;
38   } while(l);
39
40   return p; // return result as a pointer to string
41 }
```

14.13.2.3 char∗ _ltoa (unsigned long *l*)

"ltoa"
This is the selfmade conversion from unsigned long to ascii-string
digits are the number of digits behind the"."

Definition at line 50 of file MySerial.h.

```
50                               {
51 static char b[31];
52 static char const digit[] = "0123456789";
53 char* p = b;
54 uint32_t i;
55   p=b+28;
56   *p = 0;
57   *(p+1)=0;
58
59   do { //Move back, inserting digits as u go
60     Yield(20);
61       p--;
62       *p = digit[l % 10l];
63       //p--;                           // just to proof we are using this _ltoa
64       //*p = 'l';
65       l = l/10l;
66   } while(l);
67
68
69   return p; // return result as a pointer to string
70 }
```

14.13.2.4 void Yield (unsigned long *mics*)

Yield() is the Scheduler.

Yield() is called by Yield(0) and Delay(micros) Yield decides which task should run next and starts that task via TaskSwitch(old, new)

Yield() is called by Yield(0) and Delay(micros) Yield decides which task should run next and start that task via TaskSwitch(old, new) MySerial.println(i);

MySerial.print("Task 0 : ");

MySerial.println(i);

MySerial.print("------------------------to Ready: "); slow=1; MySerial.println(i);

MySerial.print("Delay: "); MySerial.println(mics);

Definition at line 260 of file TaskSwitch.h.

```
260                                {
261 unsigned long m=micros();
262 struct task *tp, *tp2;
263   if (YActive) return;
264
265 YieldLoop:
266   YActive=true;
267
268   SwitchCount++;
269   BITSETD6;
270   tp=&Tasks[current_task];
271   tp->lastCalled=m;
272
273   // Delay(x) ?
274   if (tp->state==READY) {
275     if (mics>0) {
276       tp->Delay=mics;
277       //tp->stopped=1;
278       tp->state = BLOCKED;
279       tp->state2 = DEL;
280     }
281   }
282
283   // Search for the next task to run
284   uint8_t prio=0;
285   uint8_t oldTask=current_task;
286
287   oldTask=current_task;                            // save the old task for TaskSwitch(old, new)
288   uint8_t HiPrio=0, HiNum=0;
289   extern uint8_t number_of_tasks;
290   uint8_t i;
291
292
293   for (i=1; i < number_of_tasks; i++) {
294
295     tp2=&Tasks[i];
296     if (tp2->state==BLOCKED) {                     // test: BLOCKED -> READY? Delay is over?
298       if (tp2->Delay) {
299         if ((m-tp2->lastCalled)>=tp2->Delay) {       // is a new task ready ?
300           tp2->Delay=0;
301           tp2->state=READY;
302           tp2->state2=RDY;
303         }
304       }
305     }
306
307     // Search Task with highest priority:
308     if (tp2->state== READY) {
309        if (tp2->prio > HiPrio) {
310          HiPrio=tp2->prio;
311          HiNum=i;
312        }
313     }
314
315   }
316
```

```
current_task=HiNum;                      // This is the new task to run
Tasks[current_task].state=READY;
Tasks[current_task].state2=RUN;
YActive=false;

BITSETD4;
TaskSwitch(oldTask, current_task);
BITCLEARD4;

}
```

14.13.3 Variable Documentation

14.13.3.1 mySerial MySerial

Definition at line 278 of file MySerial.h.

14.13.3.2 char OutBuf[SER_BUF_MAX]

Definition at line 8 of file MySerial.h.

14.13.3.3 volatile int SerHead

Definition at line 7 of file MySerial.h.

14.13.3.4 volatile int SerTail

Definition at line 7 of file MySerial.h.

14.14 Pins.cpp File Reference

```
#include <Arduino.h>
#include <inttypes.h>
```

Functions

- uint8_t ∗ MyPinToPort (int pin)
- uint8_t MyPinToBitMask (int pin)

14.14.1 Function Documentation

14.14.1.1 uint8_t MyPinToBitMask (int *pin*)

Definition at line 11 of file Pins.cpp.

```
                                {
  if (pin<8) return (1<<pin);
  if ((pin>=8) & (pin<=13)) return (1<<(pin-8));
}
```

14.14.1.2 uint8_t* MyPinToPort (int *pin*)

Definition at line 6 of file Pins.cpp.

```
6                              {
7   if (pin<8) return &PIND;
8   if ((pin>=8) & (pin<=13)) return &PINB;
9 }
```

14.15 Pins.h File Reference

Macros

- #define BIT_SET(a, b) ((a) |= (1ULL<<(b)))
- #define BIT_CLEAR(a, b) ((a) &= ~(1ULL<<(b)))
- #define BIT_TOGGLE(a, b) ((a) ^= (1ULL<<(b)))
- #define BIT_CHECK(a, b) (!!((a) & (1ULL<<(b))))
- #define BIT_CHECKH(a, b) ((a&(1<<b)) != 0)
- #define BIT_CHECKL(a, b) ((a&(1<<b)) == 0)
- #define BITSETD0 BIT_SET(PORTD,0)
- #define BITSETD1 BIT_SET(PORTD,1)
- #define BITSETD2 BIT_SET(PORTD,2)
- #define BITSETD3 BIT_SET(PORTD,3)
- #define BITSETD4 BIT_SET(PORTD,4)
- #define BITSETD5 BIT_SET(PORTD,5)
- #define BITSETD6 BIT_SET(PORTD,6)
- #define BITSETD7 BIT_SET(PORTD,7)
- #define BITSETD8 BIT_SET(PORTB,0)
- #define BITSETD9 BIT_SET(PORTB,1)
- #define BITSETD10 BIT_SET(PORTB,2)
- #define BITSETD11 BIT_SET(PORTB,3)
- #define BITSETD12 BIT_SET(PORTB,4)
- #define BITSETD13 BIT_SET(PORTB,5)
- #define BITCLEARD0 BIT_CLEAR(PORTD,0)
- #define BITCLEARD1 BIT_CLEAR(PORTD,1)
- #define BITCLEARD2 BIT_CLEAR(PORTD,2)
- #define BITCLEARD3 BIT_CLEAR(PORTD,3)
- #define BITCLEARD4 BIT_CLEAR(PORTD,4)
- #define BITCLEARD5 BIT_CLEAR(PORTD,5)
- #define BITCLEARD6 BIT_CLEAR(PORTD,6)
- #define BITCLEARD7 BIT_CLEAR(PORTD,7)
- #define BITCLEARD8 BIT_CLEAR(PORTB,0)
- #define BITCLEARD9 BIT_CLEAR(PORTB,1)
- #define BITCLEARD10 BIT_CLEAR(PORTB,2)
- #define BITCLEARD11 BIT_CLEAR(PORTB,3)
- #define BITCLEARD12 BIT_CLEAR(PORTB,4)
- #define BITCLEARD13 BIT_CLEAR(PORTB,5)
- #define BITTOGGLED0 BIT_TOGGLE(PORTD,0)
- #define BITTOGGLED1 BIT_TOGGLE(PORTD,1)
- #define BITTOGGLED2 BIT_TOGGLE(PORTD,2)
- #define BITTOGGLED3 BIT_TOGGLE(PORTD,3)
- #define BITTOGGLED4 BIT_TOGGLE(PORTD,4)
- #define BITTOGGLED5 BIT_TOGGLE(PORTD,5)
- #define BITTOGGLED6 BIT_TOGGLE(PORTD,6)

- #define BITTOGGLED7 BIT_TOGGLE(PORTD,7)
- #define BITTOGGLED8 BIT_TOGGLE(PORTB,0)
- #define BITTOGGLED9 BIT_TOGGLE(PORTB,1)
- #define BITTOGGLED10 BIT_TOGGLE(PORTB,2)
- #define BITTOGGLED11 BIT_TOGGLE(PORTB,3)
- #define BITTOGGLED12 BIT_TOGGLE(PORTB,4)
- #define BITTOGGLED13 BIT_TOGGLE(PORTB,5)
- #define BITCHECKD0 BIT_CHECK(PIND,0)
- #define BITCHECKD1 BIT_CHECK(PIND,1)
- #define BITCHECKD2 BIT_CHECK(PIND,2)
- #define BITCHECKD3 BIT_CHECK(PIND,3)
- #define BITCHECKD4 BIT_CHECK(PIND,4)
- #define BITCHECKD5 BIT_CHECK(PIND,5)
- #define BITCHECKD6 BIT_CHECK(PIND,6)
- #define BITCHECKD7 BIT_CHECK(PIND,7)
- #define BITCHECKD8 BIT_CHECK(PORTB,0)
- #define BITCHECKD9 BIT_CHECK(PORTB,1)
- #define BITCHECKD10 BIT_CHECK(PORTB,2)
- #define BITCHECKD11 BIT_CHECK(PORTB,3)
- #define BITCHECKD12 BIT_CHECK(PORTB,4)
- #define BITCHECKD13 BIT_CHECK(PORTB,5)

Functions

- uint8_t * MyPinToPort (int pin)
- uint8_t MyPinToBitMask (int pin)

14.15.1 Macro Definition Documentation

14.15.1.1 #define BIT_CHECK(*a*, *b*) (!!((a) & (1ULL<<(b))))

Definition at line 9 of file Pins.h.

14.15.1.2 #define BIT_CHECKH(*a*, *b*) ((a&(1<<b)) != 0)

Definition at line 10 of file Pins.h.

14.15.1.3 #define BIT_CHECKL(*a*, *b*) ((a&(1<<b)) == 0)

Definition at line 11 of file Pins.h.

14.15.1.4 #define BIT_CLEAR(*a*, *b*) ((a) &= ~(1ULL<<(b)))

Definition at line 6 of file Pins.h.

14.15.1.5 #define BIT_SET(*a*, *b*) ((a) |= (1ULL<<(b)))

Definition at line 5 of file Pins.h.

14.15.1.6 #define BIT_TOGGLE(a, b) ((a) ^= (1ULL<<(b)))

Definition at line 7 of file Pins.h.

14.15.1.7 #define BITCHECKD0 BIT_CHECK(PIND,0)

Definition at line 93 of file Pins.h.

14.15.1.8 #define BITCHECKD1 BIT_CHECK(PIND,1)

Definition at line 94 of file Pins.h.

14.15.1.9 #define BITCHECKD10 BIT_CHECK(PORTB,2)

Definition at line 104 of file Pins.h.

14.15.1.10 #define BITCHECKD11 BIT_CHECK(PORTB,3)

Definition at line 105 of file Pins.h.

14.15.1.11 #define BITCHECKD12 BIT_CHECK(PORTB,4)

Definition at line 106 of file Pins.h.

14.15.1.12 #define BITCHECKD13 BIT_CHECK(PORTB,5)

Definition at line 107 of file Pins.h.

14.15.1.13 #define BITCHECKD2 BIT_CHECK(PIND,2)

Definition at line 95 of file Pins.h.

14.15.1.14 #define BITCHECKD3 BIT_CHECK(PIND,3)

Definition at line 96 of file Pins.h.

14.15.1.15 #define BITCHECKD4 BIT_CHECK(PIND,4)

Definition at line 97 of file Pins.h.

14.15.1.16 #define BITCHECKD5 BIT_CHECK(PIND,5)

Definition at line 98 of file Pins.h.

14.15.1.17 #define BITCHECKD6 BIT_CHECK(PIND,6)

Definition at line 99 of file Pins.h.

14.15.1.18 #define BITCHECKD7 BIT_CHECK(PIND,7)

Definition at line 100 of file Pins.h.

14.15.1.19 #define BITCHECKD8 BIT_CHECK(PORTB,0)

Definition at line 102 of file Pins.h.

14.15.1.20 #define BITCHECKD9 BIT_CHECK(PORTB,1)

Definition at line 103 of file Pins.h.

14.15.1.21 #define BITCLEARD0 BIT_CLEAR(PORTD,0)

Definition at line 58 of file Pins.h.

14.15.1.22 #define BITCLEARD1 BIT_CLEAR(PORTD,1)

Definition at line 59 of file Pins.h.

14.15.1.23 #define BITCLEARD10 BIT_CLEAR(PORTB,2)

Definition at line 69 of file Pins.h.

14.15.1.24 #define BITCLEARD11 BIT_CLEAR(PORTB,3)

Definition at line 70 of file Pins.h.

14.15.1.25 #define BITCLEARD12 BIT_CLEAR(PORTB,4)

Definition at line 71 of file Pins.h.

14.15.1.26 #define BITCLEARD13 BIT_CLEAR(PORTB,5)

Definition at line 72 of file Pins.h.

14.15.1.27 #define BITCLEARD2 BIT_CLEAR(PORTD,2)

Definition at line 60 of file Pins.h.

14.15.1.28 #define BITCLEARD3 BIT_CLEAR(PORTD,3)

Definition at line 61 of file Pins.h.

14.15.1.29 #define BITCLEARD4 BIT_CLEAR(PORTD,4)

Definition at line 62 of file Pins.h.

14.15.1.30 #define BITCLEARD5 BIT_CLEAR(PORTD,5)

Definition at line 63 of file Pins.h.

14.15.1.31 #define BITCLEARD6 BIT_CLEAR(PORTD,6)

Definition at line 64 of file Pins.h.

14.15.1.32 #define BITCLEARD7 BIT_CLEAR(PORTD,7)

Definition at line 65 of file Pins.h.

14.15.1.33 #define BITCLEARD8 BIT_CLEAR(PORTB,0)

Definition at line 67 of file Pins.h.

14.15.1.34 #define BITCLEARD9 BIT_CLEAR(PORTB,1)

Definition at line 68 of file Pins.h.

14.15.1.35 #define BITSETD0 BIT_SET(PORTD,0)

Definition at line 42 of file Pins.h.

14.15.1.36 #define BITSETD1 BIT_SET(PORTD,1)

Definition at line 43 of file Pins.h.

14.15.1.37 #define BITSETD10 BIT_SET(PORTB,2)

Definition at line 53 of file Pins.h.

14.15.1.38 #define BITSETD11 BIT_SET(PORTB,3)

Definition at line 54 of file Pins.h.

14.15.1.39 #define BITSETD12 BIT_SET(PORTB,4)

Definition at line 55 of file Pins.h.

14.15.1.40 #define BITSETD13 BIT_SET(PORTB,5)

Definition at line 56 of file Pins.h.

14.15.1.41 #define BITSETD2 BIT_SET(PORTD,2)

Definition at line 44 of file Pins.h.

14.15.1.42 #define BITSETD3 BIT_SET(PORTD,3)

Definition at line 45 of file Pins.h.

14.15.1.43 #define BITSETD4 BIT_SET(PORTD,4)

Definition at line 46 of file Pins.h.

14.15.1.44 #define BITSETD5 BIT_SET(PORTD,5)

Definition at line 47 of file Pins.h.

14.15.1.45 #define BITSETD6 BIT_SET(PORTD,6)

Definition at line 48 of file Pins.h.

14.15.1.46 #define BITSETD7 BIT_SET(PORTD,7)

Definition at line 49 of file Pins.h.

14.15.1.47 #define BITSETD8 BIT_SET(PORTB,0)

Definition at line 51 of file Pins.h.

14.15.1.48 #define BITSETD9 BIT_SET(PORTB,1)

Definition at line 52 of file Pins.h.

14.15.1.49 #define BITTOGGLED0 BIT_TOGGLE(PORTD,0)

Definition at line 75 of file Pins.h.

14.15.1.50 #define BITTOGGLED1 BIT_TOGGLE(PORTD,1)

Definition at line 76 of file Pins.h.

14.15.1.51 #define BITTOGGLED10 BIT_TOGGLE(PORTB,2)

Definition at line 86 of file Pins.h.

14.15.1.52 #define BITTOGGLED11 BIT_TOGGLE(PORTB,3)

Definition at line 87 of file Pins.h.

14.15.1.53 #define BITTOGGLED12 BIT_TOGGLE(PORTB,4)

Definition at line 88 of file Pins.h.

14.15.1.54 #define BITTOGGLED13 BIT_TOGGLE(PORTB,5)

Definition at line 89 of file Pins.h.

14.15.1.55 #define BITTOGGLED2 BIT_TOGGLE(PORTD,2)

Definition at line 77 of file Pins.h.

14.15.1.56 #define BITTOGGLED3 BIT_TOGGLE(PORTD,3)

Definition at line 78 of file Pins.h.

14.15.1.57 #define BITTOGGLED4 BIT_TOGGLE(PORTD,4)

Definition at line 79 of file Pins.h.

14.15.1.58 #define BITTOGGLED5 BIT_TOGGLE(PORTD,5)

Definition at line 80 of file Pins.h.

14.15.1.59 #define BITTOGGLED6 BIT_TOGGLE(PORTD,6)

Definition at line 81 of file Pins.h.

14.15.1.60 #define BITTOGGLED7 BIT_TOGGLE(PORTD,7)

Definition at line 82 of file Pins.h.

14.15.1.61 #define BITTOGGLED8 BIT_TOGGLE(PORTB,0)

Definition at line 84 of file Pins.h.

14.15.1.62 #define BITTOGGLED9 BIT_TOGGLE(PORTB,1)

Definition at line 85 of file Pins.h.

14.15.2 Function Documentation

14.15.2.1 uint8_t MyPinToBitMask (int *pin*)

Definition at line 11 of file Pins.cpp.

```
11                                   {
12   if (pin<8) return (1<<pin);
13   if ((pin>=8) & (pin<=13)) return (1<<(pin-8));
14 }
```

14.15.2.2 uint8_t∗ MyPinToPort (int *pin*)

Definition at line 6 of file Pins.cpp.

```
{
  if (pin<8) return &PIND;
  if ((pin>=8) & (pin<=13)) return &PINB;
}
```

14.16 ShowSt.h File Reference

14.17 Task.h File Reference

Classes

- class task

Typedefs

- typedef void(∗ FuncPt) (void)

Enumerations

- enum State { BLOCKED, READY, BLOCKED, READY }
- enum State2 {
  NON, RDY, RUN, DEL,
  BLK, NON, RDY, RUN,
  DEL, BLK }

Functions

- void StackPrepare ()

  StackPrepepare Alloc the new stack for all tasks.

Variables

- char ∗ State2Txt [ ] = { "NON", "RDY", "RUN", "DEL", "BLK" }
- uint8_t number_of_tasks
- unsigned int FirstSP
- unsigned int StackLow
- unsigned int StackHi
- char ∗ STACK
- volatile struct task Tasks [MAX_TASKS]
- volatile uint8_t current_task =0
- unsigned int IdleCount
- volatile unsigned int SwitchCount

14.17.1 Typedef Documentation

14.17.1.1 typedef void(∗ FuncPt) (void)

Definition at line 9 of file Task.h.

14.17.2 Enumeration Type Documentation

14.17.2.1 enum State

Enumerator

- ***BLOCKED***
- ***READY***
- ***BLOCKED***
- ***READY***

Definition at line 11 of file Task.h.

```
11 { BLOCKED, READY};
```

14.17.2.2 enum State2

Enumerator

- ***NON***
- ***RDY***
- ***RUN***
- ***DEL***
- ***BLK***
- ***NON***
- ***RDY***
- ***RUN***
- ***DEL***
- ***BLK***

Definition at line 12 of file Task.h.

```
12 { NON, RDY, RUN, DEL, BLK };
```

14.17.3 Function Documentation

14.17.3.1 void StackPrepare ( )

StackPrepepare Alloc the new stack for all tasks.

Malloc is used to get the amount of STACKALLOC bytes from the heap as a new stack for all tasks
This is the only malloc in the program.
The reserved block is used by TaskInit() to distribute parts of this block to the single tasks
TaskInit() stops if there is not enough space for the stacks of the tasks.

Definition at line 61 of file Task.h.

```
{
// Alloc the stack for tasks:
  STACK=malloc(STACKALLOC+1);
  //Serial.println((uint16_t)STACK);
  StackLow=STACK;
  StackHi=STACK+STACKALLOC;
  STACK  =StackHi;
  StackPt=STACK;
  //Serial.println((uint16_t)STACK);

}
```

14.17.4 Variable Documentation

14.17.4.1 volatile uint8_t current_task =0

Actual Task running

Definition at line 44 of file Task.h.

14.17.4.2 unsigned int FirstSP

StackPtr for all tasks at start of program

Definition at line 37 of file Task.h.

14.17.4.3 unsigned int IdleCount

Definition at line 45 of file Task.h.

14.17.4.4 uint8_t number_of_tasks

set and counted by Init()

Definition at line 36 of file Task.h.

14.17.4.5 char* STACK

Definition at line 42 of file Task.h.

14.17.4.6 unsigned int StackHi

High Boundery of all taskstacks

Definition at line 39 of file Task.h.

14.17.4.7 unsigned int StackLow

Low Boundery of all taskstacks

Definition at line 38 of file Task.h.

14.17.4.8 char* State2Txt[ ] = { "NON", "RDY", "RUN", "DEL", "BLK" }

full description of state

Definition at line 13 of file Task.h.

14.17.4.9 volatile unsigned int SwitchCount

incremented by Yield/Delay calls

Definition at line 46 of file Task.h.

14.17.4.10 volatile struct task Tasks[MAX_TASKS]

Table of all initialized task

Definition at line 43 of file Task.h.

14.18 TaskSwitch.h File Reference

```
#include "Task.h"
```

Macros

- #define Delay(n) Yield(n)
- #define StartMultiTasking()

  StartMultiTasking.
- #define pushall()
- #define popall()
- #define stackFree(task) memSearch(Tasks[task].task_stack-Tasks[task].stackLen+1,Tasks[task].sp_↩ save,0x55)
- #define myStackFree() stackFree(current_task)

Functions

- uint16_t memSearch (uint8_t ∗startp, uint8_t ∗endp, uint8_t v)
- void stopMe ()
- void stop_task (uint8_t t)
- void resume_task (uint8_t t)
- void TaskSwitch (uint8_t old, uint8_t newer)
- void Yield (unsigned long mics)

  Yield() is the Scheduler.
- void Idle ()
- int freeRam ()
- uint8_t TaskInit (char ∗_name, FuncPt _function, int16_t _stackLen, uint8_t _prio, unsigned long _delay, State _state)

  TaskInit() fills the Table of Tasks to start.

Variables

- uint8_t Flag = 0
- bool YActive =false
- uint8_t oldTasks [10]
- uint32_t oldMicros [10]
- uint8_t Sreg

  TaskSwitch.

14.18.1 Macro Definition Documentation

14.18.1.1 #define Delay(*n*) Yield(n)

Definition at line 5 of file TaskSwitch.h.

14.18.1.2 #define myStackFree( ) stackFree(current_task)

Definition at line 125 of file TaskSwitch.h.

14.18.1.3 #define popall( )

Value:

```
asm volatile ("pop r29 \n\t pop r28 \n\t pop r17 \n\t pop r16 \n\t pop r15 \n\t pop r14 \n\t pop r13 \n\t
       pop r12 \n\t pop r11 \n\t" \
              "pop r10 \n\t pop r9  \n\t pop r8  \n\t pop r7  \n\t pop r6  \n\t pop r5  \n\t pop r4  \n\t
       pop r3  \n\t pop r2");
```

Definition at line 107 of file TaskSwitch.h.

14.18.1.4 #define pushall( )

Value:

```
asm volatile ("push r2  \n\t push r3  \n\t push r4  \n\t push r5  \n\t push r6  \n\t push r7  \n\t push r8
       \n\t push r9  \n\t push r10 \n\t" \
              "push r11 \n\t push r12 \n\t push r13 \n\t push r14 \n\t push r15 \n\t push r16 \n\t push
       r17 \n\t push r28 \n\t push r29");
```

Definition at line 103 of file TaskSwitch.h.

14.18.1.5 #define stackFree(task) memSearch(Tasks[task].task_stack-Tasks[task].stackLen+1,Tasks[task].sp_save,0x55)

Definition at line 124 of file TaskSwitch.h.

14.18.1.6 #define StartMultiTasking( )

Value:

```
SP=STACK;\
  Tasks[0].function();
```

StartMultiTasking.

All preparations are done now and the multitasking begins

Definition at line 86 of file TaskSwitch.h.

14.18.2 Function Documentation

14.18.2.1 int freeRam ( )

Definition at line 446 of file TaskSwitch.h.

```
{
  extern int __heap_start, *__brkval;
  int v;
  return (int) &v - (__brkval == 0 ? (int) &__heap_start : (int) __brkval);
}
```

14.18.2.2 void Idle ( )

Definition at line 104 of file CoopOS_Stack_MT_Nano.ino.

```
{
  while (1)
  {
    IdleCount++;     // only for test, you may comment out this line
    Yield (0);
  }
}
```

14.18.2.3 uint16_t memSearch (uint8_t * *startp*, uint8_t * *endp*, uint8_t *v*)

Definition at line 118 of file TaskSwitch.h.

```
{ uint8_t *ptr = startp;
  while (ptr < endp) if (*ptr++ != v) break;
  return ((uint16_t)ptr-(uint16_t)startp-1);
}
```

14.18.2.4 void resume_task (uint8_t *t*)

Definition at line 151 of file TaskSwitch.h.

```
{
  Tasks[t].state = READY;
  Tasks[t].state2 = RDY;
}
```

14.18.2.5 void stop_task (uint8_t *t*)

Definition at line 144 of file TaskSwitch.h.

```
{
  Tasks[t].state = BLOCKED;
  Tasks[t].state2=BLK;
}
```

14.18.2.6 void stopMe ( )

Definition at line 136 of file TaskSwitch.h.

```
136              {
137   Tasks[current_task].state=BLOCKED;
138   Tasks[current_task].state2=BLK;
139
140 }
```

14.18.2.7 uint8_t TaskInit (char ∗ *_name*, FuncPt *_function*, int16_t *_stackLen*, uint8_t *_prio*, unsigned long *_delay*, State *_state*)

TaskInit() fills the Table of Tasks to start.

// NAM FUNC STCK PRIO DLY STATE TaskInit("T1 ", Task1, 90, 104, 0, READY); | | | | | | NAM is the const char∗ to a name | | | | | | | FUNC is the name of the task function the source| | | | | STCK is the amount of stack used by this task | | | | PRIO is the priority of this task | | | | DLY is the delay for this task in microseconds | | STATE can be READY or BLOCKED (if DLY $>$ 0) |

Definition at line 360 of file TaskSwitch.h.

```
{
static int task_num=0;
//static int StackPt=STACK;
extern int __heap_start, *__brkval;
  if (task_num>=MAX_TASKS) {
    //MySerial.println(F("INIT: MAX_TASKS OVERFLOW:"));
    Serial.println("INIT: MAX_TASKS OVERFLOW:");
    Serial.println(_name);
    while(1);
  }

  // auto-create idle task Idle
  if (task_num==0) {
    //TaskInit("T0", task0, 0, St_len, false , 0, READY);
    Tasks[task_num].name = "Idle";
    Tasks[task_num].function = Idle;
    Tasks[task_num].prio = 0;
    Tasks[task_num].stackLen=IDLE_STLEN;

    if (freeRam<100) {
      //MySerial.println(F("FreeRam < 100 Bytes !!!"));
      Serial.println(F("FreeRam < 100 Bytes !!!"));
      while(1);
```

```
    }

    Tasks[task_num].task_stack = STACK;
    Tasks[task_num].stackLen = IDLE_STLEN;

    Tasks[task_num].new_task = false;
    Tasks[task_num].lastCalled = micros();
    Tasks[task_num].Delay = 0;
    Tasks[task_num].state = READY;
    Tasks[task_num].state2 = RDY;

    task_num++;
    number_of_tasks=task_num;
    StackPt-=IDLE_STLEN;

//    Serial.println((int)StackLow);
//    Serial.println((int)IDLE_STLEN);
//    Serial.println((int)StackPt);
    Serial.print(F("Idle")); Serial.print(-IDLE_STLEN); Serial.print(F(": Stack free for next task: ")); Serial.println((int)(StackPt-StackLow));

  }


  Tasks[task_num].name = _name;
  Tasks[task_num].function = _function;
  Tasks[task_num].prio = _prio;
  //Tasks[task_num].task_stack = _task_stack;
  Tasks[task_num].task_stack = StackPt;
  StackPt-=_stackLen;
  Tasks[task_num].stackLen=_stackLen;

  if ( StackPt < StackLow ) {
    //MySerial.println(F("ERROR: STACK too small !"));
    Serial.println(F("ERROR: STACK too small !"));
    while(1);
  }

  //Serial.println((int)_stackLen);
  //Serial.println((int)StackPt);
  Serial.print(_name);  Serial.print(-_stackLen); Serial.print(F(": Stack free for next task: ")); Serial.println((int)(StackPt-StackLow));


  Tasks[task_num].new_task = true;
  Tasks[task_num].lastCalled = micros();
  Tasks[task_num].Delay = _delay;
  if (_delay != 0) _state=BLOCKED;
  Tasks[task_num].state = _state;

  task_num++;
  number_of_tasks=task_num;
  return task_num-1;

}
```

14.18.2.8 void TaskSwitch (uint8_t *old*, uint8_t *newer*) [inline]

Definition at line 176 of file TaskSwitch.h.

```
{
struct task *TP;

  pushall();                              // save old tasks register
  Sreg=SREG;
  cli();
  Tasks[old].sp_save = SP;                // save old tasks stackpointer


#ifdef TRACE_ON
  for ( uint8_t ii=1; ii<4; ii++) {
    oldTasks[ii-1]=oldTasks[ii];
    oldMicros[ii-1]=oldMicros[ii];
  }
  oldTasks[3]=newer;
  oldMicros[3]=(uint32_t)micros();
#endif
  SREG=Sreg;

  current_task=newer;
  TP=&Tasks[current_task];
```

```
    if (TP->new_task == true)             // a task marked as NEW should be installed
    {                                     // it just runs without return other than Yield();
     // All tasks in Init(...); will come to this point
     // at first run. They start with their own stack pointer
     // all registers are filled with scratch but that doesn't matter because they start new !

      TP->new_task = false;
      Sreg=SREG;                          // we want to save SREG: Interrupts enabled/disable is the same for new task
      cli();
      SP = TP->task_stack;                // set SP to the new task. It will save it's stack when Yielded
      SREG=Sreg;

      BITCLEARD6;
                                          // Registers are pushed and old Stackpointer is save and this tasks Stackpointer is set
      TP->function ();                    // now run the task amrked as new. It should not return ( starting a while(1)-loop)

      // if there is no while-loop or while is left:
      // a task has ended!
      TP->state = BLOCKED;                // all finished task ( jumped out of while(1) ) will end here
      TP->new_task == true;               // if resumed they start again
                                          // this could make sense: such tasks may be resumed by interrupt or other tasks
                                          // for a "one-shot-action"
    }


    else                                  // tasks, which have run and saved their registers are invoked here:
    {
                                          // Yield had select the this task: highest priority and READY
      Sreg=SREG;                          // save interrupt enabled state
      cli();                              // switch to the SP of this task, own stack is saved (s.a. pushall )
      SP = TP->sp_save;                   // switch the stack to the saved stackpt
      SREG=Sreg;                          // restore interrupt enabled state
      popall();                           // pop saved registers and return from last run => goto last Yield-Point of switched out task

      //YActive=false;

      BITCLEARD6;
      return;
    }

}
```

14.18.2.9 void Yield (unsigned long *mics*)

Yield() is the Scheduler.

Yield() is called by Yield(0) and Delay(micros) Yield decides which task should run next and starts that task via TaskSwitch(old, new) MySerial.println(i);

Definition at line 260 of file TaskSwitch.h.

```
260                              {
261 unsigned long m=micros();
262 struct task *tp, *tp2;
263   if (YActive) return;
264
265 YieldLoop:
266   YActive=true;
267
268   SwitchCount++;
269   BITSETD6;
270   tp=&Tasks[current_task];
271   tp->lastCalled=m;
272
273   // Delay(x) ?
274   if (tp->state==READY) {
275     if (mics>0) {
276       tp->Delay=mics;
277       //tp->stopped=1;
```

```
      tp->state = BLOCKED;
      tp->state2 = DEL;
    }
  }

  // Search for the next task to run
  uint8_t prio=0;
  uint8_t oldTask=current_task;

  oldTask=current_task;                        // save the old task for TaskSwitch(old,
   new)
  uint8_t HiPrio=0, HiNum=0;
  extern uint8_t number_of_tasks;
  uint8_t i;


  for (i=1; i < number_of_tasks; i++) {

    tp2=&Tasks[i];
    if (tp2->state==BLOCKED) {                 // test: BLOCKED -> READY? Delay is over?
      if (tp2->Delay) {
        if ((m-tp2->lastCalled)>=tp2->Delay) {       // is a new task ready ?
          tp2->Delay=0;
          tp2->state=READY;
          tp2->state2=RDY;
        }
      }
    }

    // Search Task with highest priority:
    if (tp2->state== READY) {
       if (tp2->prio > HiPrio) {
         HiPrio=tp2->prio;
         HiNum=i;
       }
    }

  }

  current_task=HiNum;                          // This is the new task to run
  Tasks[current_task].state=READY;
  Tasks[current_task].state2=RUN;
  YActive=false;

  BITSETD4;
  TaskSwitch(oldTask, current_task);
  BITCLEARD4;

}
```

14.18.3 Variable Documentation

14.18.3.1 uint8_t Flag = 0

Definition at line 7 of file TaskSwitch.h.

14.18.3.2 uint32_t oldMicros[10]

Definition at line 164 of file TaskSwitch.h.

14.18.3.3 uint8_t oldTasks[10]

Definition at line 162 of file TaskSwitch.h.

14.18.3.4 uint8_t Sreg

TaskSwitch.

TaskSwitch is called by Yield() [Delay()] It saves the state of the running task and switch to newer task

Definition at line 174 of file TaskSwitch.h.

14.18.3.5 bool YActive =false

Definition at line 250 of file TaskSwitch.h.

14.19 TaskSwitchDemo.h File Reference

Classes

- class task

Macros

- #define STACKALLOC 765
- #define IDLE_STLEN 75
- #define MAX_TASKS 8
- #define StartMultiTasking()
- #define pushall()
- #define popall()
- #define Delay(x) Yield(x);
- #define StackInit() STACK=malloc(STACKALLOC+1);\

Typedefs

- typedef void(∗ FuncPt) (void)

Enumerations

- enum State { BLOCKED, READY, BLOCKED, READY }
- enum State2 {
  NON, RDY, RUN, DEL,
  BLK, NON, RDY, RUN,
  DEL, BLK }

Functions

- void stopMe ()
- void stop_task (uint8_t tn)
- void resume_task (uint8_t tn)
- void TaskSwitch (uint8_t old, uint8_t newer)

  TaskSwitch.
- void Yield (unsigned long mics)

  Yield() is the Scheduler.
- void Idle ()
- int freeRam ()
- uint8_t TaskInit (char ∗_name, FuncPt _function, int16_t _stackLen, uint8_t _prio, unsigned long _delay, State _state)

  TaskInit() fills the Table of Tasks to start.
- if (STACK==NULL)
- Serial print (F("\n\nStack allocated: "))
- Serial println (STACKALLOC)
- Serial print (F("Free Ram now : "))
- Serial println (freeRam())
- Serial println ()

Variables

- char ∗ State2Txt [ ] = { "NON", "RDY", "RUN", "DEL", "BLK" }
- char ∗ STACK =STACK+STACKALLOC
- uint8_t number_of_tasks
- volatile uint8_t current_task =0
- unsigned int FirstSP
- volatile struct task Tasks [MAX_TASKS]
- unsigned int StackLow =STACK
- unsigned int StackHi =STACK+STACKALLOC
- char DisplayUsed =0
- unsigned int IdleCount
- unsigned int BlinkCount
- unsigned int SwitchCount
- uint8_t T7Handle

14.19.1 Macro Definition Documentation

14.19.1.1 #define Delay(x) Yield(x);

Definition at line 471 of file TaskSwitchDemo.h.

14.19.1.2 #define IDLE_STLEN 75

Definition at line 4 of file TaskSwitchDemo.h.

14.19.1.3 #define MAX_TASKS 8

Definition at line 5 of file TaskSwitchDemo.h.

14.19.1.4 #define popall( )

Value:

```
asm volatile \
  (            \
    "pop r29 \n\t"      \
    "pop r28 \n\t"      \
    "pop r17 \n\t"      \
    "pop r16 \n\t"      \
    "pop r15 \n\t"      \
    "pop r14 \n\t"      \
    "pop r13 \n\t"      \
    "pop r12 \n\t"      \
    "pop r11 \n\t"      \
    "pop r10 \n\t"      \
    "pop r9  \n\t"      \
    "pop r8  \n\t"      \
    "pop r7  \n\t"      \
    "pop r6  \n\t"      \
    "pop r5  \n\t"      \
    "pop r4  \n\t"      \
    "pop r3  \n\t"      \
    "pop r2  \n\t"      \
  )
```

Definition at line 131 of file TaskSwitchDemo.h.

14.19.1.5 #define pushall( )

Value:

```
asm volatile \
  (           \
    "push r2  \n\t"   \
    "push r3  \n\t"   \
    "push r4  \n\t"   \
    "push r5  \n\t"   \
    "push r6  \n\t"   \
    "push r7  \n\t"   \
    "push r8  \n\t"   \
    "push r9  \n\t"   \
    "push r10 \n\t"   \
    "push r11 \n\t"   \
    "push r12 \n\t"   \
    "push r13 \n\t"   \
    "push r14 \n\t"   \
    "push r15 \n\t"   \
    "push r16 \n\t"   \
    "push r17 \n\t"   \
    "push r28 \n\t"   \
    "push r29 \n\t"   \
  )
```

Definition at line 109 of file TaskSwitchDemo.h.

14.19.1.6 #define STACKALLOC 765

Definition at line 2 of file TaskSwitchDemo.h.

14.19.1.7 #define StackInit( ) STACK=malloc(STACKALLOC+1);\

Definition at line 477 of file TaskSwitchDemo.h.

14.19.1.8 #define StartMultiTasking( )

Value:

```
SP=STACK;\
 Tasks[0].function ();
```

Definition at line 49 of file TaskSwitchDemo.h.

14.19.2 Typedef Documentation

14.19.2.1 typedef void(∗ FuncPt) (void)

Definition at line 9 of file TaskSwitchDemo.h.

14.19.3 Enumeration Type Documentation

14.19.3.1 enum State

Enumerator

BLOCKED

READY

BLOCKED

READY

Definition at line 11 of file TaskSwitchDemo.h.

```
11 { BLOCKED, READY};                              // State of task
```

14.19.3.2 enum State2

Enumerator

NON

RDY

RUN

DEL

BLK

NON

RDY

RUN

DEL

BLK

Definition at line 12 of file TaskSwitchDemo.h.

```
12 { NON, RDY, RUN, DEL, BLK };                   // SubState of task
```

14.19.4 Function Documentation

14.19.4.1 int freeRam ( )

Definition at line 457 of file TaskSwitchDemo.h.

```
458 {
459   extern int __heap_start, *__brkval;
460   int v;
461   return (int) &v - (__brkval == 0 ? (int) &__heap_start : (int) __brkval);
462 }
```

14.19.4.2 void Idle (void)

Definition at line 464 of file TaskSwitchDemo.h.

```
464             {
465   while(1) {
466     Yield(0);
467   }
468
469 }
```

14.19.4.3 if (STACK = =NULL)

Definition at line 483 of file TaskSwitchDemo.h.

```
483                    { \
484     Serial.println(F("ERROR: Not enough room for Stack !!!"));\
485     while(1);\
486   }\
```

14.19.4.4 Serial print (F("\n\nStack allocated: "))

14.19.4.5 Serial print (F("Free Ram now : "))

14.19.4.6 Serial println (STACKALLOC)

14.19.4.7 Serial println (freeRam())

14.19.4.8 Serial println ( )

14.19.4.9 void resume_task (uint8_t *tn*)

Definition at line 173 of file TaskSwitchDemo.h.

```
174 {
175   Tasks[tn].state = READY;
176   Tasks[tn].state2 = RDY;
177 }
```

14.19.4.10 void stop_task (uint8_t *tn*)

Definition at line 167 of file TaskSwitchDemo.h.

```
168 {
169   Tasks[tn].state = BLOCKED;
170   Tasks[tn].state2=BLK;
171 }
```

14.19.4.11 void stopMe ( )

Definition at line 161 of file TaskSwitchDemo.h.

```
161               {
162   Tasks[current_task].state=BLOCKED;
163   Tasks[current_task].state2=BLK;
164
165 }
```

14.19.4.12 uint8_t TaskInit (char * _name, FuncPt _function, int16_t _stackLen, uint8_t _prio, unsigned long _delay, State _state)

TaskInit() fills the Table of Tasks to start.

Yield() is called by Yield(0) and Delay(micros) Yield decides which task should run next and start that task via TaskSwitch(old, new)

Definition at line 370 of file TaskSwitchDemo.h.

```
{
static int task_num=0;
//static int StackPt=STACK;
extern int __heap_start, *__brkval;
  if (task_num>=MAX_TASKS) {
    //MySerial.println(F("INIT: MAX_TASKS OVERFLOW:"));
    Serial.println(F("INIT: MAX_TASKS OVERFLOW:"));
    Serial.println(_name);
    while(1);
  }

  // auto-create idle task task0
  if (task_num==0) {
    //TaskInit("T0", task0, 0, St_len, false , 0, READY);
    Tasks[task_num].name = "Idle";
    Tasks[task_num].function = Idle;
    Tasks[task_num].prio = 0;
    Tasks[task_num].stackLen=IDLE_STLEN;

    if (freeRam<100) {
      //MySerial.println(F("FreeRam < 100 Bytes !!!"));
      Serial.println(F("FreeRam < 100 Bytes !!!"));
      while(1);
    }

    Tasks[task_num].task_stack = STACK;
    Tasks[task_num].stackLen = IDLE_STLEN;

    Tasks[task_num].new_task = false;
    Tasks[task_num].lastCalled = micros();
    Tasks[task_num].Delay = 0;
    Tasks[task_num].state = READY;
    Tasks[task_num].state2 = RDY;

    task_num++;
    number_of_tasks=task_num;
    StackPt-=IDLE_STLEN;

//    Serial.println((int)StackLow);
//    Serial.println((int)IDLE_STLEN);
//    Serial.println((int)StackPt);
    Serial.print((F("Idle)"); Serial.print(-IDLE_STLEN); Serial.print(F(": Stack free for next
   task: ")); Serial.println((int)(StackPt-StackLow));

  }


  Tasks[task_num].name = _name;
  Tasks[task_num].function = _function;
  Tasks[task_num].prio = _prio;
  //Tasks[task_num].task_stack = _task_stack;
  Tasks[task_num].task_stack = StackPt;
  StackPt-=_stackLen;
  Tasks[task_num].stackLen=_stackLen;

  if ( StackPt < StackLow ) {
    //MySerial.println(F("ERROR: STACK too small !"));
    Serial.println(F("ERROR: STACK too small !"));
    while(1);
  }

  //Serial.println((int)_stackLen);
  //Serial.println((int)StackPt);
  Serial.print(_name);  Serial.print(-_stackLen); Serial.print(F(": Stack free for next task: ")); Serial.
  println((int)(StackPt-StackLow));


  Tasks[task_num].new_task = true;
  Tasks[task_num].lastCalled = micros();
  Tasks[task_num].Delay = _delay;
  if (_delay != 0) _state=BLOCKED;
```

```
    Tasks[task_num].state = _state;

    task_num++;
    number_of_tasks=task_num;
    return task_num-1;

}
```

14.19.4.13 void TaskSwitch (uint8_t *old*, uint8_t *newer*) [inline]

TaskSwitch.

TaskSwitch is called by Yield() [Delay()] It saves the state of the running task and switch to newer task

Definition at line 191 of file TaskSwitchDemo.h.

```
{
  // Save old
  cli();
  pushall();
  Tasks[old].sp_save = SP;

  sei();
  while (1)
  {

    current_task=newer;

    if (Tasks[current_task].new_task == true)                // a task marked as
   NEW should be installed
    {                                                        // the running task has save it's context (s.a)
     // All tasks in Init(...); will come to this point
     // at first run. All initialisation before while(1) are wil be done now
      cli();
      Tasks[current_task].new_task = false;
      SP = Tasks[current_task].task_stack;                   // set SP to the new
   task. It will save it's stack when Yielded
      //BITCLEARD6;
      sei();
      Tasks[current_task].function ();                       // run the new task.
   It should not return ( starting a while(1)-loop)

      // a task has ended!
      Tasks[current_task].state = BLOCKED;                   // all finished
   task ( jumped out of while(1) ) will end here
      Tasks[current_task].new_task == true;                  // if resumed they
   start again
    }
    else
    {
      // this task had been started and has save stack      // Yield had select the next task
      cli();                                                 // switch to the SP of this task, own stack is
   saved (s.a. pushall )
      SP = Tasks[current_task].sp_save;                      // switch the stack to
   the save stackpt of next task to execute
      sei();

      popall();                                              // pop saved registers and return -> goto
   last Yield-Point of switched out task
      //BITCLEARD6;
      return;
    }
  }
}
```

14.19.4.14 void Yield (unsigned long *mics*)

Yield() is the Scheduler.

Yield() is called by Yield(0) and Delay(micros) Yield decides which task should run next and start that task via TaskSwitch(old, new) MySerial.print("Task 0 : ");

MySerial.println(i);

MySerial.print("------------------------to Ready: "); slow=1; MySerial.println(i);

MySerial.print("Delay: "); MySerial.println(mics);

Definition at line 249 of file TaskSwitchDemo.h.

```
                               {
unsigned long m=micros();
struct task *tp, *tp2;
  SwitchCount++;
  //BITSETD6;
  tp=&Tasks[current_task];
  tp->lastCalled=m;

Start:

  if (current_task==0) {                                       // idle
    Tasks[0].state2=RDY ;
    for (int i=1; i < MAX_TASKS; i++) {

      tp2=&Tasks[i];
      //if (Tasks[i].stopped==1) {
      if (Tasks[i].state==BLOCKED) {
        if (Tasks[i].Delay) {
          if ((m-tp2->lastCalled)>=Tasks[i].Delay) {

            //tp2->stopped=0;
            tp2->Delay=0;
            tp2->state=READY;
            tp2->state2=RDY;

           // while(1);
          }
        }



      //MySerial.println(i);

      }
    }
  }
  else {
    //Tasks[current_task].lastCalled=micros();
    if (tp->state==READY) {
      if (mics>0) {
        tp->Delay=mics;
        //tp->stopped=1;
        tp->state = BLOCKED;
        tp->state2 = DEL;
      }
    }
  }



  // Search for the next task
  int prio=0;
  int oldTask=current_task;
//  do
//  {
//      /*  Circle through the tasks, the if() construct is much
//        faster than using a mod % operator. */
//      if (++current_task == number_of_tasks)
//        current_task = 0;
//
//    /*  Skipping inactive tasks.  It is not uncommon for the system
//      to spend most of it's time chasing its tail going through a
//      list of all inactive tasks.  This is a good place to add code
//      for putting the processor in a low power state.  It is also
//      a good place to modulate an output pin to make a PWM measurement
//      of processor loading. */
//  } while (Tasks[current_task].state == BLOCKED);

  oldTask=current_task;
  char HiPrio=0, HiNum=0;
  extern char number_of_tasks;
  for (int i=1 /* not Idle */; i < number_of_tasks; i++) {
    if (Tasks[i].state== READY) {
       if (Tasks[i].prio > HiPrio) {
         HiPrio=Tasks[i].prio;
         HiNum=i;
       }
    }
  }
  current_task=HiNum;


  if (oldTask==current_task) {
    tp->state=READY;
    tp->state2=RUN;
    //BITCLEARD6;
    // no TaskSwitch needed
```

```
    return;
  }
  else {
    //current_task=oldTask;
    //Tasks[oldTask].state=BLOCKED;
    Tasks[current_task].state=READY;
    Tasks[current_task].state2=RUN;

    TaskSwitch(oldTask, current_task);
  }
}
```

14.19.5 Variable Documentation

14.19.5.1 unsigned int BlinkCount

Definition at line 43 of file TaskSwitchDemo.h.

14.19.5.2 volatile uint8_t current_task =0

Definition at line 34 of file TaskSwitchDemo.h.

14.19.5.3 char DisplayUsed =0

Definition at line 41 of file TaskSwitchDemo.h.

14.19.5.4 unsigned int FirstSP

Definition at line 35 of file TaskSwitchDemo.h.

14.19.5.5 unsigned int IdleCount

Definition at line 42 of file TaskSwitchDemo.h.

14.19.5.6 uint8_t number_of_tasks

Definition at line 33 of file TaskSwitchDemo.h.

14.19.5.7 STACK =STACK+STACKALLOC

Definition at line 32 of file TaskSwitchDemo.h.

14.19.5.8 StackHi =STACK+STACKALLOC

Definition at line 38 of file TaskSwitchDemo.h.

14.19.5.9 StackLow =STACK

Definition at line 37 of file TaskSwitchDemo.h.

14.19.5.10 char* State2Txt[ ] = { "NON", "RDY", "RUN", "DEL", "BLK" }

Definition at line 13 of file TaskSwitchDemo.h.

14.19.5.11 unsigned int SwitchCount

Definition at line 44 of file TaskSwitchDemo.h.

14.19.5.12 uint8_t T7Handle

Definition at line 45 of file TaskSwitchDemo.h.

14.19.5.13 volatile struct task Tasks[MAX_TASKS]

Definition at line 36 of file TaskSwitchDemo.h.

14.20 Timing.jpg File Reference