

CoopOS\_Stack\_MT\_Nano

Generated by Doxygen 1.8.11

## Contents

<b>1</b>	<b>What does this program ?</b>	<b>1</b>
1.1	All Demos as ZIP-files: . . . . .	1
<b>2</b>	<b>Doxyfile</b>	<b>2</b>
<b>3</b>	<b>Introduction</b>	<b>2</b>
<b>4</b>	<b>Overview</b>	<b>3</b>
<b>5</b>	<b>Getting started</b>	<b>6</b>
<b>6</b>	<b>Intertask Communication</b>	<b>10</b>
<b>7</b>	<b>Initializing Tasks</b>	<b>16</b>
<b>8</b>	<b>Tools</b>	<b>17</b>
8.1	Serial Output . . . . .	17
8.2	Pins . . . . .	18
8.3	Show Stack . . . . .	18
8.4	Functions to check stackspace . . . . .	20
8.5	WatchDogTimer . . . . .	21
8.6	Interrupts . . . . .	22
8.7	Debug / Breakpoints . . . . .	24
<b>9</b>	<b>The Program</b>	<b>27</b>
<b>10</b>	<b>Conclusion</b>	<b>30</b>
<b>11</b>	<b>Class Index</b>	<b>31</b>
11.1	Class List . . . . .	31
<b>12</b>	<b>File Index</b>	<b>31</b>
12.1	File List . . . . .	31

<b>13 Class Documentation</b>	<b>32</b>
13.1 mySerial Class Reference . . . . .	32
13.1.1 Detailed Description . . . . .	33
13.1.2 Member Function Documentation . . . . .	33
13.2 task Class Reference . . . . .	37
13.2.1 Detailed Description . . . . .	38
13.2.2 Member Data Documentation . . . . .	38
<b>14 File Documentation</b>	<b>39</b>
14.1 calc.jpg File Reference . . . . .	39
14.2 CoopOS_Stack_MT_Nano.ino File Reference . . . . .	39
14.2.1 Macro Definition Documentation . . . . .	40
14.2.2 Function Documentation . . . . .	42
14.2.3 Variable Documentation . . . . .	44
14.3 Debug.h File Reference . . . . .	45
14.4 Demo.jpg File Reference . . . . .	45
14.5 Demo0.jpg File Reference . . . . .	45
14.6 Demo_0A.jpg File Reference . . . . .	45
14.7 DoPrep.h File Reference . . . . .	45
14.7.1 Macro Definition Documentation . . . . .	45
14.8 Doxyfile.dox File Reference . . . . .	45
14.9 inter1.jpg File Reference . . . . .	45
14.10inter2.jpg File Reference . . . . .	45
14.11MT1.jpg File Reference . . . . .	45
14.12MySer.h File Reference . . . . .	45
14.13MySerial.h File Reference . . . . .	45
14.13.1 Macro Definition Documentation . . . . .	46
14.13.2 Function Documentation . . . . .	46
14.13.3 Variable Documentation . . . . .	47
14.14Pins.cpp File Reference . . . . .	48
14.14.1 Function Documentation . . . . .	48

14.15Pins.h File Reference . . . . .	49
14.15.1 Macro Definition Documentation . . . . .	50
14.15.2 Function Documentation . . . . .	55
14.16ShowSt.h File Reference . . . . .	56
14.17Task.h File Reference . . . . .	56
14.17.1 Typedef Documentation . . . . .	56
14.17.2 Enumeration Type Documentation . . . . .	57
14.17.3 Function Documentation . . . . .	57
14.17.4 Variable Documentation . . . . .	58
14.18TaskSwitch.h File Reference . . . . .	59
14.18.1 Macro Definition Documentation . . . . .	60
14.18.2 Function Documentation . . . . .	61
14.18.3 Variable Documentation . . . . .	65
14.19TaskSwitchDemo.h File Reference . . . . .	65
14.19.1 Macro Definition Documentation . . . . .	67
14.19.2 Typedef Documentation . . . . .	68
14.19.3 Enumeration Type Documentation . . . . .	69
14.19.4 Function Documentation . . . . .	69
14.19.5 Variable Documentation . . . . .	74
14.20Timing.jpg File Reference . . . . .	75

## 1 What does this program ?

(C) 2019 Dipl. Phys. Helmut Weber

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 library 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  $\mu$ s) 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 glimpse of what to do to make it cooperative: [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

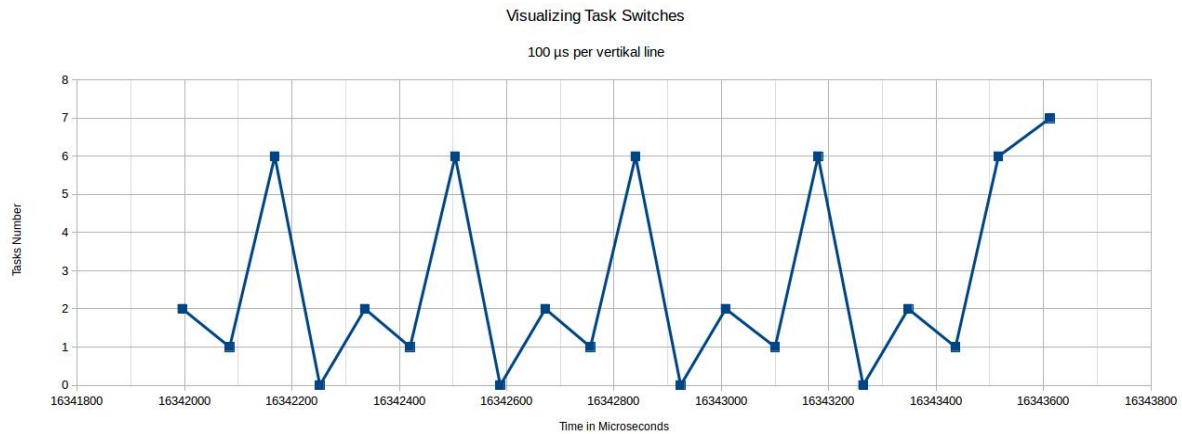
    TaskInit("T1", Task1, 90, 100, 0, READY); // define 2 tasks for multitasking:
    TaskInit("T2", Task2, 90, 100, 0, READY);

    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 preferred 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 multitasking you have to turn a switch in your 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 want 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 running 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

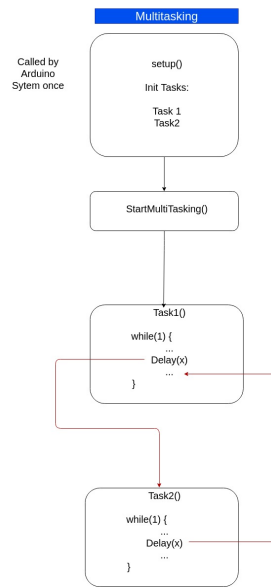
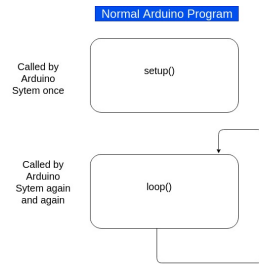
  } // end while
}
```

In the Arduino-IDE you normalay 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

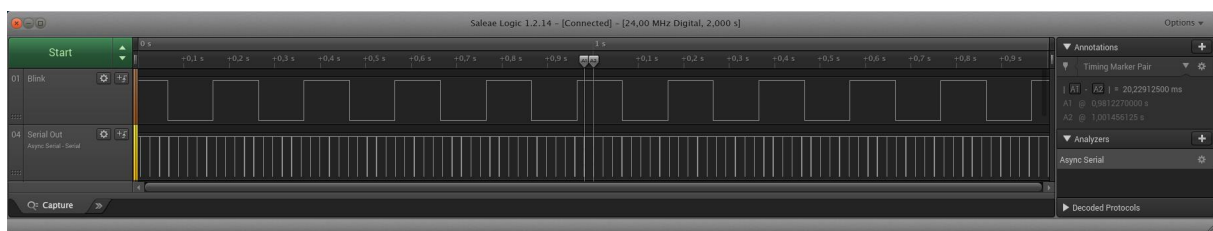
12066660 49 // <<<< Serial Monitor: send characters  
12067260 50  
12067860 51

```

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 results.

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

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 they get switched out. If a task A increments a global variable 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 its work, it got a problem.

Some mechanisms are necessary to solve this problem.>br>

A cooperative task knows, that it will not get disturbed until it relinquishes the control to the scheduler! (Well - another thing are interrupts).

So it is much easier for cooperative systems to communicate through 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.

Then 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 to reach infinity. It has a cycle time 10% less than Task 2. So it will be able to see always the changes Task 2 made.

If PosX reaches 80 Task 3 changes the sign of the increment of Task 2. The the is true if Task 2 reaches 0:

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





Generated by Doxygen



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 itself."

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 and do not save them in registers. Here it is not necessary, but it is a good habit to 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 explanations you may want:

[<< Getting started](#)

[>> Initializing Tasks](#)

## 7 Initializing Tasks

If you look at the source it is not obvious, which functions are just pure functions and which functions should act as tasks.

Therefore it is advisable the name functions which should be tasks with a "task" in its function name.

But that is for readability 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 priority numbers are only used 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 tasklist, which task is READY and has the

highest priority. If two tasks have the same priority AND both tasks are READY then the first of these tasks is executed.

It is a good practice though not mandatory to give all tasks different priorities.

**\_delay:**

In microseconds. If `_delay` is  $> 0$  then this task 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  $>$  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 your 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; BITCLEAR13;` insted of `digitalWrite(13,HIGH); digitalWrite(13,LOW)` and the pulsewidth will reduce

from 3.8  $\mu$ s to 0.125  $\mu$ s - 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  $\mu$ s 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 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 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 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 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 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
```

```
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
```

```
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
```

```
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 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
```

```
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
```

```
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
```

```
...
```

```
...
```

```
.
```

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 stackspace for the running task.

- **stackFree(ID)** Returns the free stackspace 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 cooperative 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 Switch↔ Count.

The WDT tests in it's ISR, if SwitchCount was incremented since the last call of WDT.

The WDT interrupts  $WDT\_VALUE * 0.5 \mu s$ . If WDT\_VALUE is set to 400 the the interrupt occurs every 200  $\mu s$ .

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.}
```

```
{
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.}
```

```
{
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:**

saoiuoipoipoipo poipoipo

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  $< 1 \mu\text{s}$ .

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

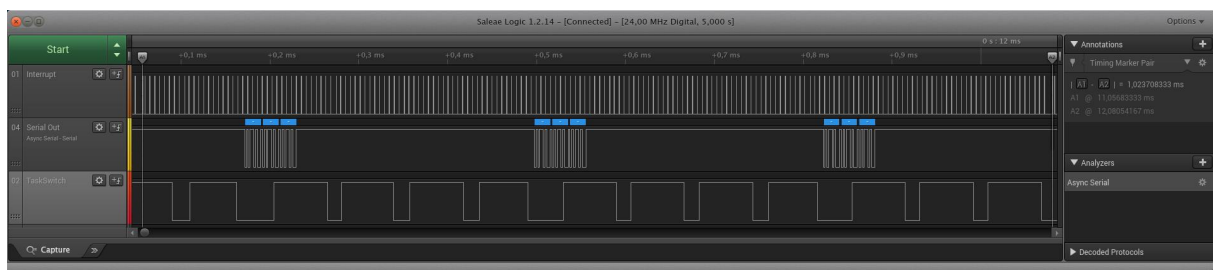
```

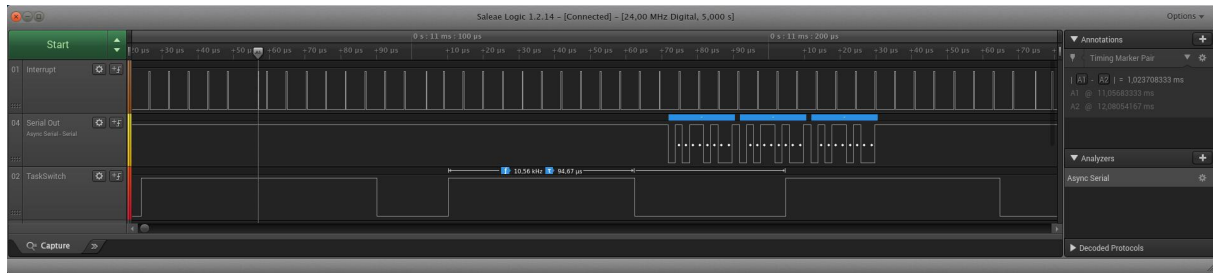
#define WDT
#define WDT_VALUE 10           // = 5 µs

```

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 losing characters and write a long line for each character
- We have incredible 200.000 (in words: Twohundredthousand) interrupts 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 gets more painful.

I implemented them and 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 second 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.

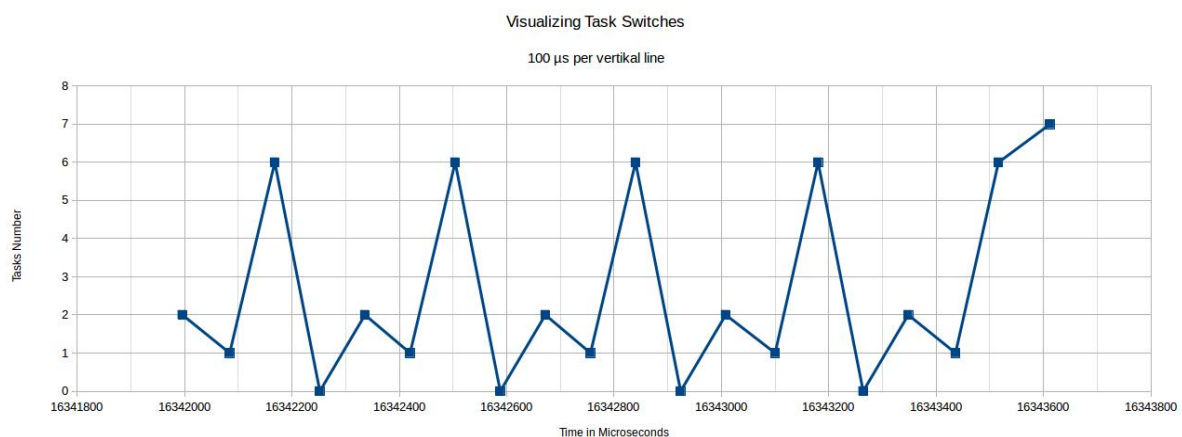
That makes sense because the tracing is time consuming and costs up to 50% of performance!

```

----- 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  $\mu$ s (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 itself.  
The time from Task 1 to Task2 is about 80  $\mu$ s.
- Last Task running  
Task 7 is BREAKPOINT wich is called from Task 4. Task 4 started with 5 seconds delay set in Init( $\leftarrow$  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( $\mu$ s)    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
0x65f: 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

```

```

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 55 55 55 55 55 55 55
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 00
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
0x5ef: 55 55 55 55

```

We interrupted Task 5 while printing 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 go 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 too small for your program and the program stops.

- **IDLE\_STLEN**

The length of the "Idle" task that must be present at all programs. Idle is called if no other task is READY.

- **MAX\_TASKS:**

The maximal number of tasks which can be defined - including "Idle"

- **TRACE\_ON**

This program has a "Debug" module. If enabled "Debug" breaks the program when it comes to a "BREAKPOINT" 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 spreads 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 enabled. 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 structure 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 Globals are defined and then comes the definition of all

**Tasks:**

- **Idle**  
Idle must always be present and MUST have function name: Idle !!!  
It is always 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 microseconds. As Task1 has the highest priority it should run as next - and it does after about 55  $\mu$ s. 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 initied 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 ;)**

**Therefore I think it is the best to use the source provided as ZIP-files, which do not include these doxygen 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 it 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 calculate pi for thousands of digits - they are good for fast reaction, do "simultaneously" at lot of short task-parts. This is typically for get/set AD/DA-values, running servos 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.  
> 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 multitasking / coroutines becoming more and more modern programming languages: 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 **nearly 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 challenges 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 your different tasks.

The full compatibility 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:

<a href="#">mySerial</a>	<a href="#">32</a>
<a href="#">task</a>	<a href="#">37</a>

## 12 File Index

### 12.1 File List

Here is a list of all files with brief descriptions:

<a href="#">calc.jpg</a>	<a href="#">39</a>
<a href="#">CoopOS_Stack_MT_Nano.ino</a>	<a href="#">39</a>
<a href="#">Debug.h</a>	<a href="#">45</a>
<a href="#">Demo.jpg</a>	<a href="#">45</a>
<a href="#">Demo0.jpg</a>	<a href="#">45</a>
<a href="#">Demo_0A.jpg</a>	<a href="#">45</a>
<a href="#">DoPrep.h</a>	<a href="#">45</a>
<a href="#">inter1.jpg</a>	<a href="#">45</a>
<a href="#">inter2.jpg</a>	<a href="#">45</a>
<a href="#">MT1.jpg</a>	<a href="#">45</a>
<a href="#">MySer.h</a>	<a href="#">45</a>
<a href="#">MySerial.h</a>	<a href="#">45</a>
<a href="#">Pins.cpp</a>	<a href="#">48</a>

<a href="#">Pins.h</a>	49
<a href="#">ShowSt.h</a>	56
<a href="#">Task.h</a>	56
<a href="#">TaskSwitch.h</a>	59
<a href="#">TaskSwitchDemo.h</a>	65
<a href="#">Timing.jpg</a>	75

## 13 Class Documentation

### 13.1 mySerial Class Reference

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

#### Public Member Functions

- [setSerial](#) (Stream \*streamObject)
- [write](#) (byte b)
- void [toSer](#) (char c)
- [write](#) (char c)
- [println](#) ()
- [print](#) (char \*str)
- [println](#) (char \*str)
- [print](#) (unsigned int i)
- [println](#) (unsigned int i)
- [print](#) (uint8\_t i)
- [print](#) (uint8\_t i, uint8\_t n)
- [print](#) (unsigned int i, uint8\_t n)
- [println](#) (unsigned int i, int n)
- [println](#) (uint8\_t i)
- [print](#) (int i)
- [println](#) (int i)
- [print](#) (unsigned long i)
- [println](#) (unsigned long i)
- [print](#) (long i)
- [println](#) (long i)
- [print](#) (float i)
- [println](#) (float i)
- char [read](#) ()
- bool [available](#) ()
- void [flush](#) ()

#### 13.1.1 Detailed Description

Definition at line 113 of file MySerial.h.

### 13.1.2 Member Function Documentation

#### 13.1.2.1 `bool mySerial::available ( )` [inline]

Definition at line 247 of file MySerial.h.

```
247         {  
248     return mystream->available();  
249     }
```

#### 13.1.2.2 `void mySerial::flush ( )` [inline]

Definition at line 251 of file MySerial.h.

```
251         {  
252     mystream->flush();  
253     }
```

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

Definition at line 147 of file MySerial.h.

```
147         {  
148     char *pt=str;  
149     while(*pt) write(*pt++);  
150     }
```

#### 13.1.2.4 `mySerial::print ( unsigned int i )` [inline]

Definition at line 159 of file MySerial.h.

```
159         {  
160     char buf[20];  
161     itoa(i, buf, 10);  
162     print(buf);  
163     }
```

#### 13.1.2.5 `mySerial::print ( uint8_t i )` [inline]

Definition at line 170 of file MySerial.h.

```
170         {  
171     char buf[20];  
172     itoa(i, buf, 10);  
173     print(buf);  
174     }
```

#### 13.1.2.6 `mySerial::print ( uint8_t i, uint8_t n )` [inline]

Definition at line 176 of file MySerial.h.

```
176         {  
177     char buf[20];  
178     itoa(i, buf, n);  
179     print(buf);  
180     }
```

#### 13.1.2.7 mySerial::print ( unsigned int *i*, uint8\_t *n* ) [inline]

Definition at line 182 of file MySerial.h.

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

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

Definition at line 198 of file MySerial.h.

```
198     {
199     char buf[20];
200     itoa(i, buf, 10);
201     print(buf);
202 }
```

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

Definition at line 210 of file MySerial.h.

```
210                                     {
211     char buf[20];
212     ltoa(i, buf, 10);
213     print(buf);
214 }
```

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

Definition at line 222 of file MySerial.h.

```
222     {
223     char buf[20];
224     ltoa(i, buf, 10);
225     print(buf);
226 }
```

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

Definition at line 233 of file MySerial.h.

```
233     {
234     print(ftoa(i, 2));
235 }
```

#### 13.1.2.12 mySerial::println ( ) [inline]

Definition at line 143 of file MySerial.h.

```
143     {
144     write('\n');
145 }
```

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

Definition at line 152 of file MySerial.h.

```
152             {  
153     print(str);  
154     println();  
155 }
```

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

Definition at line 165 of file MySerial.h.

```
165             {  
166     print(i);  
167     println();  
168 }
```

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

Definition at line 188 of file MySerial.h.

```
188             {  
189     print(i,n);  
190     println();  
191 }
```

**13.1.2.16 mySerial::println ( uint8\_t *i* ) [inline]**

Definition at line 193 of file MySerial.h.

```
193             {  
194     print(i);  
195     println();  
196 }
```

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

Definition at line 204 of file MySerial.h.

```
204             {  
205     print(i);  
206     println();  
207 }
```

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

Definition at line 216 of file MySerial.h.

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

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

Definition at line 228 of file MySerial.h.

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

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

Definition at line 237 of file MySerial.h.

```
237         {
238     print(i);
239     println();
240 }
```

**13.1.2.21 char mySerial::read ( ) [inline]**

Definition at line 243 of file MySerial.h.

```
243         {
244     return mystream->read();
245 }
```

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

Definition at line 119 of file MySerial.h.

```
120     {
121     mystream=streamObject;
122 }
```

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

Definition at line 129 of file MySerial.h.

```
129         {
130     // wait until Transmitter empty
131     while ( !( UCSRA & (1<<UDRE0)) );
132     // send one byte
133     UDR0 = (uint8_t)c;
134 }
```

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

Definition at line 125 of file MySerial.h.

```
125         {
126     //mystream->write(b);
127 }
```



### 13.1.2.25 mySerial::write ( char c ) [inline]

Definition at line 136 of file MySerial.h.

```

136         {
137             //mystream->write(c);
138             OutBuf[SerHead++]=c;
139             if (SerHead==SER_BUF_MAX) SerHead=0;
140         }

```

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  $\mu$ s

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  $\mu$ s

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

Always true 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"
#include "Debug.h"
```

#### Macros

- #define [STACKALLOC](#) 730
- #define [IDLE\\_STLEN](#) 80
- #define [MAX\\_TASKS](#) 8
- #define [STACKALLOC](#) 730
- #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](#) 499
- #define [\\_\\_MYSER](#)
- #define [\\_\\_DEBUG](#)
- #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

- char [DisplayUsed](#) =0
- unsigned int [BlinkCount](#)
- uint8\_t [DbgHandle](#)

### 14.2.1 Macro Definition Documentation

#### 14.2.1.1 #define \_\_DEBUG

Definition at line 66 of file CoopOS\_Stack\_MT\_Nano.ino.

#### 14.2.1.2 #define \_\_MYSER

Definition at line 63 of file CoopOS\_Stack\_MT\_Nano.ino.

#### 14.2.1.3 #define \_\_SHOWST

Definition at line 69 of file CoopOS\_Stack\_MT\_Nano.ino.

#### 14.2.1.4 #define IDLE\_STLEN 80

Definition at line 22 of file CoopOS\_Stack\_MT\_Nano.ino.

#### 14.2.1.5 #define LED\_Off BITCLEARD13

Definition at line 82 of file CoopOS\_Stack\_MT\_Nano.ino.

#### 14.2.1.6 #define LED\_On BITSETD13;

Definition at line 81 of file CoopOS\_Stack\_MT\_Nano.ino.

#### 14.2.1.7 #define MAX\_TASKS 8

Definition at line 23 of file CoopOS\_Stack\_MT\_Nano.ino.

#### 14.2.1.8 #define STACKALLOC 730

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

Definition at line 28 of file CoopOS\_Stack\_MT\_Nano.ino.

#### 14.2.1.9 #define STACKALLOC 730

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

Definition at line 28 of file CoopOS\_Stack\_MT\_Nano.ino.

#### 14.2.1.10 #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 / Yield(0) / Delay(n) until the next Yield(0) / Delay(n) / s. TaskSwitch.h for some examples

Definition at line 42 of file CoopOS\_Stack\_MT\_Nano.ino.

#### 14.2.1.11 #define WDT\_VALUE 499

Definition at line 48 of file CoopOS\_Stack\_MT\_Nano.ino.

### 14.2.2 Function Documentation

#### 14.2.2.1 void Idle ( void )

Definition at line 96 of file CoopOS\_Stack\_MT\_Nano.ino.

```

97 {
98   while (1)
99   {
100     IdleCount++;    // only for test, you may comment out this line
101     Yield (0);
102   }
103 }
```

#### 14.2.2.2 void loop ( )

/code>

Definition at line 299 of file CoopOS\_Stack\_MT\_Nano.ino.

#### 14.2.2.3 void setup ( )

Definition at line 229 of file CoopOS\_Stack\_MT\_Nano.ino.

```

229   {
230     Serial.begin(500000);
231
232     #ifdef __MYSER
233     MySerial.setSerial(&Serial); // Redirect Serial
234     #endif
235
236     StackPrepare();
237
238
239     if (STACK==NULL) {
240       Serial.println(F("ERROR: Not enough room for Stack !!!"));
241       while(1);
242     }
243     Serial.print(F("\n\nStack allocated: ")); Serial.println(STACKALLOC);
244     Serial.print(F("Free Ram now   : ")); Serial.println(freeRam());
245     Serial.println();
246
247     pinMode(4,OUTPUT);
248     pinMode(5,OUTPUT);
249     pinMode(6,OUTPUT); // TaskSwitch
250     pinMode(2, INPUT_PULLUP);
251
252     // fill stack with 0x55 as markers to test stack usage
253     uint8_t *pt=STACK; while (pt>StackLow) { *pt--=0x55;}
254
255     Serial.println(F("----- Init start -----"));
```

```

256
258 TaskInit("T1 ", Task1,          90, 105, 0,    READY);
259 TaskInit("T2 ", Task2,          90, 106, 0,    READY);
260 TaskInit("T3 ", Task3,          80, 100, 0,    READY);
261 TaskInit("T4 ", Task4,          90, 102, 5000000, BLOCKED); // starts after 5
    seconds
262
263
264
265 #ifdef __SHOWST
266 TaskInit("SHS", ShowStack_Task, 110, 103, 0,    READY);
267 #endif
268
269
270 #ifdef __MYSER
271 TaskInit("MyS", MySer_Task,      80, 103, 0,    READY);
272 #endif
273
274 #ifdef __DEBUG
275 DbgHandle=
276 TaskInit("Dbg", Dbg_Task,       110, 102, 0,    READY);
277 #endif
278
279
280
281 Serial.println(F("----- Init ready -----"));
282 Serial.print(F("\nFree Ram now : ")); Serial.println(freeRam());
283
284 #ifdef __DEBUG//For Debugger-Button:
285 cli(); //For Debugger-Button: Enable FALLING interrupt at D2
286 pinMode(2, INPUT_PULLUP);
287 attachInterrupt (digitalPinToInterrupt (2), IRQ_Answer, FALLING); // attach interrupt handler for D2
288 sei();
289 #endif
290
291
292 StartMultiTasking();
293 Serial.println(F("Returned from StartMultiTasking()"));
294 Serial.flush();
295 while(1);
296 }

```

#### 14.2.2.4 void Task1 ( void )

Definition at line 109 of file CoopOS\_Stack\_MT\_Nano.ino.

```

110 {
111 while (1)
112 {
113     stopMe(); // activated by next Yield/ Delay
114     LED_Off; // 45 µs since Task2
115     BlinkCount++;
116     Yield(0); // Task1 has been stopped here
117 }
118 }

```

#### 14.2.2.5 void Task2 ( void )

Definition at line 122 of file CoopOS\_Stack\_MT\_Nano.ino.

```

123 {
124 while (1)
125 {
126     LED_On;
127     BlinkCount++;
128     Tasks[1].state = READY; // Start Task1
129     Delay(200); // <----- NOT 0
130 }
131 }

```

#### 14.2.2.6 void Task3 ( void )

Definition at line 135 of file CoopOS\_Stack\_MT\_Nano.ino.

```

136 {
137   unsigned char missing;
138   while (1)
139   {
140     BITSETD5;
141     if (DisplayUsed==0) {
142       DisplayUsed=1;
143       MySerial.print("Task 3 :");
144       if (missing==0) { // Total output: 100 µs
145         MySerial.print(" IdleCount/s: ");
146         Yield(100);
147
148         MySerial.print(IdleCount);
149         Yield(100);
150         MySerial.print(", BlinkCount/s: ");
151         Yield(100);
152         MySerial.print(BlinkCount);
153         Yield(100);
154         MySerial.print(", SwitchCount/s: ");
155         Yield(100);
156         MySerial.println(SwitchCount);
157         Yield(100);
158       }
159       else {
160         missing=0;
161         MySerial.println("missing values");
162       }
163       DisplayUsed=0;
164     }
165     else missing=1;
166     IdleCount=0;
167     BlinkCount=0;
168     SwitchCount=0;
169
170
171     BITCLEARD5;
172     Delay(1000000);
173   }
174 }
175 }

```

#### 14.2.2.7 void Task4 ( void )

Definition at line 198 of file CoopOS\_Stack\_MT\_Nano.ino.

```

199 {
200   while (1)
201   {
202
203     Task4_fun(); // Test Yield in a called function
204     // BITSETD5;
205     // if (DisplayUsed==0) { MySerial.print("-----Task 4, SP:"); MySerial.println(SP);}
206     // BITCLEARD5;
207     // Delay(100000);
208   }
209 }

```

#### 14.2.2.8 void Task4\_fun ( )

Definition at line 178 of file CoopOS\_Stack\_MT\_Nano.ino.

```

178   {
179     BITSETD5;
180     if (DisplayUsed==0) { MySerial.print("-----Task 4, SP: 0x");
181     MySerial.println(SP,16);}
182     BITCLEARD5;
183     Delay(100000);
184
185     while (MySerial.available()) {
186       char ch = MySerial.read();
187       //MySerial.flush();
188       Serial.print("\n----- Serial input: "); Serial.println(ch);
189
190     }
191   }
192   #ifdef __DEBUG
193   }
194   #endif
195 }

```

### 14.2.3 Variable Documentation

#### 14.2.3.1 unsigned int BlinkCount

Incremented by task 1

Definition at line 87 of file CoopOS\_Stack\_MT\_Nano.ino.

#### 14.2.3.2 uint8\_t DbgHandle

Handle for Debug = Waked up by interrupt

Definition at line 88 of file CoopOS\_Stack\_MT\_Nano.ino.

#### 14.2.3.3 char DisplayUsed =0

Used as mutex for the output

Definition at line 86 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](#) 150

### Functions

- 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](#) 150

Definition at line 6 of file MySerial.h.

### 14.13.2 Function Documentation

#### 14.13.2.1 char\* \_itoa ( unsigned int i )

"\_itoa"

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

Definition at line 17 of file MySerial.h.

```
17                                     {
18 static char b[31];
19 static char const digit[] = "0123456789";
20 char* p = b;
21 uint32_t i;
22
23
24 //i=(uint32_t)f;
25
26 p=b+28;
27 *p = 0;
28 *(p+1)=0;
29
30 do { //Move back, inserting digits as u go
31     p--;
32     *p = digit[i % 10];
33     i = i/10;
34 } while(i);
35
36 return p; // return result as a pointer to string
37 }
```

#### 14.13.2.2 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 76 of file MySerial.h.

```
76                                     {
77 static char b[31];
78 static char const digit[] = "0123456789";
79 char* p = b;
80 uint32_t i;
81
82 int d,j;
83 d=digits;
84 while (d) {
85     f*=10.0;
86     d--;
87 }
88
89 i=(uint32_t)f;
90
91 p=b+28;
92 j=0;
93 *p = 0;
94 *(p+1)=0;
95
96 do { //Move back, inserting digits as u go
97     if (j == digits) { p--; *p='.'; }
98     p--;
99     *p = digit[i % 10];
100    i = i/10;
101    j++;
102 } while(i);
103
104 return p; // return result as a pointer to string
105 }
```

### 14.13.2.3 char\* ltoa ( unsigned long / )

"ltoa"

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

Definition at line 46 of file MySerial.h.

```

46                                     {
47 static char b[31];
48 static char const digit[] = "0123456789";
49 char* p = b;
50 uint32_t i;
51
52
53 //i=(uint32_t)f;
54
55 p=b+28;
56 *p = 0;
57 *(p+1)=0;
58
59 do { //Move back, inserting digits as u go
60     p--;
61     *p = digit[l % 10];
62     l = l/10;
63 } while(l);
64
65 return p; // return result as a pointer to string
66 }
```

## 14.13.3 Variable Documentation

### 14.13.3.1 mySerial MySerial

Definition at line 266 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.

```

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

#### 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.

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

## 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 ( )

StackPrepare 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.

```
61         {
62 // Alloc the stack for tasks:
63     STACK=malloc(STACKALLOC+1);
64     //Serial.println((uint16_t)STACK);
65     StackLow=STACK;
66     StackHi=STACK+STACKALLOC;
67     STACK  =StackHi;
68     StackPt=STACK;
69     //Serial.println((uint16_t)STACK);
70
71 }
```

### 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](#) [20]
- unsigned long [oldMicros](#) [20]
- 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 123 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 105 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 101 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 122 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 84 of file TaskSwitch.h.

## 14.18.2 Function Documentation

14.18.2.1 `int freeRam ( )`

Definition at line 443 of file TaskSwitch.h.

```
444 {
445     extern int __heap_start, *__brkval;
446     int v;
447     return (int) &v - (__brkval == 0 ? (int) &__heap_start : (int) __brkval);
448 }
```

14.18.2.2 `void Idle ( )`

Definition at line 96 of file CoopOS\_Stack\_MT\_Nano.ino.

```
97 {
98     while (1)
99     {
100         IdleCount++;    // only for test, you may comment out this line
101         Yield (0);
102     }
103 }
```

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

Definition at line 116 of file TaskSwitch.h.

```
117 { uint8_t *ptr = startp;
118     while (ptr < endp) if (*ptr++ != v) break;
119     return ((uint16_t)ptr-(uint16_t)startp-1);
120 }
```



#### 14.18.2.4 void resume\_task ( uint8\_t t )

Definition at line 149 of file TaskSwitch.h.

```
150 {
151     Tasks[t].state = READY;
152     Tasks[t].state2 = RDY;
153 }
```

#### 14.18.2.5 void stop\_task ( uint8\_t t )

Definition at line 142 of file TaskSwitch.h.

```
143 {
144     Tasks[t].state = BLOCKED;
145     Tasks[t].state2=BLK;
146 }
```

#### 14.18.2.6 void stopMe ( )

Definition at line 134 of file TaskSwitch.h.

```
134 {
135     Tasks[current_task].state=BLOCKED;
136     Tasks[current_task].state2=BLK;
137
138 }
```

#### 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 357 of file TaskSwitch.h.

```
364 {
365     static int task_num=0;
366     //static int StackPt=STACK;
367     extern int __heap_start, *__brkval;
368     if (task_num>=MAX_TASKS) {
369         //MySerial.println(F("INIT: MAX_TASKS OVERFLOW:"));
370         Serial.println("INIT: MAX_TASKS OVERFLOW:");
371         Serial.println(_name);
372         while(1);
373     }
374
375     // auto-create idle task Idle
376     if (task_num==0) {
377         //TaskInit("T0", task0, 0, St_len, false , 0, READY);
378         Tasks[task_num].name = "Idle";
379         Tasks[task_num].function = Idle;
380         Tasks[task_num].prio = 0;
381         Tasks[task_num].stackLen=IDLE_STLEN;
382
383         if (freeRam<100) {
384             //MySerial.println(F("FreeRam < 100 Bytes !!!"));
385             Serial.println(F("FreeRam < 100 Bytes !!!"));
386             while(1);
387         }
388     }
389 }
```

```

387     }
388
389     Tasks[task_num].task_stack = STACK;
390     Tasks[task_num].stackLen = IDLE_STLEN;
391
392     Tasks[task_num].new_task = false;
393     Tasks[task_num].lastCalled = micros();
394     Tasks[task_num].Delay = 0;
395     Tasks[task_num].state = READY;
396     Tasks[task_num].state2 = RDY;
397
398     task_num++;
399     number_of_tasks=task_num;
400     StackPt-=IDLE_STLEN;
401
402     // Serial.println((int)StackLow);
403     // Serial.println((int)IDLE_STLEN);
404     // Serial.println((int)StackPt);
405     Serial.print("Idle"); Serial.print(-IDLE_STLEN); Serial.print(F(": Stack free for next task:
406     ")); Serial.println((int) (StackPt-StackLow));
407 }
408
409
410 Tasks[task_num].name = _name;
411 Tasks[task_num].function = _function;
412 Tasks[task_num].prio = _prio;
413 //Tasks[task_num].task_stack = _task_stack;
414 Tasks[task_num].task_stack = StackPt;
415 StackPt-=_stackLen;
416 Tasks[task_num].stackLen=_stackLen;
417
418 if ( StackPt < StackLow ) {
419     //MySerial.println(F("ERROR: STACK too small !"));
420     Serial.println(F("ERROR: STACK too small !"));
421     while(1);
422 }
423
424 //Serial.println((int)_stackLen);
425 //Serial.println((int)StackPt);
426 Serial.print(_name); Serial.print(-_stackLen); Serial.print(F(": Stack free for next task: ")); Serial.
427 println((int) (StackPt-StackLow));
428
429 Tasks[task_num].new_task = true;
430 Tasks[task_num].lastCalled = micros();
431 Tasks[task_num].Delay = _delay;
432 if (_delay != 0) _state=BLOCKED;
433 Tasks[task_num].state = _state;
434
435 task_num++;
436 number_of_tasks=task_num;
437 return task_num-1;
438
439 }

```

#### 14.18.2.8 void TaskSwitch ( uint8\_t *old*, uint8\_t *newer* ) [inline]

Definition at line 173 of file TaskSwitch.h.

```

174 {
175 struct task *TP;
176
177     pushall(); // save old tasks register
178     Sreg=SREG;
179     cli();
180     Tasks[old].sp_save = SP; // save old tasks stackpointer
181     SREG=Sreg;
182
183 #ifdef TRACE_ON
184     for ( uint8_t i=1; i<20; i++) {
185         oldTasks[i-1]=oldTasks[i];
186         oldMicros[i-1]=oldMicros[i];
187     }
188     oldTasks[19]=newer;
189     oldMicros[19]=micros();
190 #endif
191
192
193     current_task=newer;
194     TP=&Tasks[current_task];

```

```

195
196 if (TP->new_task == true)           // a task marked as NEW should be installed
197 {                                   // it just runs without return other than Yield();
198 // All tasks in Init(...); will come to this point
199 // at first run. They start with their own stack pointer
200 // all registers are filled with scratch but that doesn't matter because they start new !
201
202     TP->new_task = false;
203     Sreg=SREG;                       // we want to save SREG: Interrupts enabled/disable is the
    same for new task
204     cli();
205     SP = TP->task_stack;             // set SP to the new task. It will save it's stack when
    Yielded
206     SREG=Sreg;
207
208     BITCLEARD6;
209                                     // Registers are pushed and old Stackpointer is save and this
    tasks Stackpointer is set
210     TP->function ();                // now run the task amrked as new. It should not return (
    starting a while(1)-loop)
211
212     // if there is no while-loop or while is left:
213     // a task has ended!
214     TP->state = BLOCKED;            // all finished task ( jumped out of while(1) ) will
    end here
215     TP->new_task == true;           // if resumed they start again
216                                     // this could make sense: such tasks may be resumed by interrupt
    or other tasks
217                                     // for a "one-shot-action"
218 }
219
220
221 else                                // tasks, which have run and saved their registers are invoked
    here:
222 {
223                                     // Yield had select the this task: highest priority and READY
224     Sreg=SREG;                     // save interrupt enabled state
225     cli();                         // switch to the SP of this task, own stack is saved (s.a.
    pushall )
226     SP = TP->sp_save;               // switch the stack to the saved stackpt
227     SREG=Sreg;                     // restore interrupt enabled state
228     popall();                      // pop saved registers and return from last run => goto
    last Yield-Point of switched out task
229
230     //YActive=false;
231
232     BITCLEARD6;
233     return;
234 }
235
236 }

```

#### 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 257 of file `TaskSwitch.h`.

```

257                                     {
258 unsigned long m=micros();
259 struct task *tp, *tp2;
260 if (YActive) return;
261
262 YieldLoop:
263 YActive=true;
264
265 SwitchCount++;
266 BITSETD6;
267 tp=&Tasks[current_task];
268 tp->lastCalled=m;
269
270 // Delay(x) ?
271 if (tp->state==READY) {
272     if (mics>0) {
273         tp->Delay=mics;
274         //tp->stopped=1;

```

```

275     tp->state = BLOCKED;
276     tp->state2 = DEL;
277 }
278 }
279
280 // Search for the next task to run
281 uint8_t prio=0;
282 uint8_t oldTask=current_task;
283
284 oldTask=current_task; // save the old task for TaskSwitch(old,
new)
285 uint8_t HiPrio=0, HiNum=0;
286 extern uint8_t number_of_tasks;
287 uint8_t i;
288
289
290 for (i=1; i < number_of_tasks; i++) {
291
292     tp2=&Tasks[i];
293     if (tp2->state==BLOCKED) { // test: BLOCKED -> READY? Delay is over?
294         if (tp2->Delay) {
295             if ((m-tp2->lastCalled)>=tp2->Delay) { // is a new task ready ?
296                 tp2->Delay=0;
297                 tp2->state=READY;
298                 tp2->state2=RDY;
299             }
300         }
301     }
302 }
303
304 // Search Task with highest priority:
305 if (tp2->state== READY) {
306     if (tp2->prio > HiPrio) {
307         HiPrio=tp2->prio;
308         HiNum=i;
309     }
310 }
311
312 }
313
314 current_task=HiNum; // This is the new task to run
315 Tasks[current_task].state=READY;
316 Tasks[current_task].state2=RUN;
317 YActive=false;
318
319 BITSETD4;
320 TaskSwitch(oldTask, current_task);
321 BITCLEARD4;
322
323 }

```

### 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 unsigned long oldMicros[20]

Definition at line 161 of file TaskSwitch.h.

#### 14.18.3.3 uint8\_t oldTasks[20]

Definition at line 160 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 171 of file TaskSwitch.h.

#### 14.18.3.5 bool YActive =false

Definition at line 247 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 482 of file TaskSwitchDemo.h.

```

482      { \
483      Serial.println(F("ERROR: Not enough room for Stack !!!")); \
484      while(1); \
485      } \
```

**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`.

```

377 {
378 static int task_num=0;
379 //static int StackPt=STACK;
380 extern int __heap_start, *__brkval;
381 if (task_num>MAX_TASKS) {
382 //MySerial.println(F("INIT: MAX_TASKS OVERFLOW:"));
383 Serial.println("INIT: MAX_TASKS OVERFLOW:");
384 Serial.println(_name);
385 while(1);
386 }
387
388 // auto-create idle task task0
389 if (task_num==0) {
390 //TaskInit("T0", task0, 0, St_len, false , 0, READY);
391 Tasks[task_num].name = "Idle";
392 Tasks[task_num].function = Idle;
393 Tasks[task_num].prio = 0;
394 Tasks[task_num].stackLen=IDLE_STLEN;
395
396 if (freeRam<100) {
397 //MySerial.println(F("FreeRam < 100 Bytes !!!"));
398 Serial.println(F("FreeRam < 100 Bytes !!!"));
399 while(1);
400 }
401
402 Tasks[task_num].task_stack = STACK;
403 Tasks[task_num].stackLen = IDLE_STLEN;
404
405 Tasks[task_num].new_task = false;
406 Tasks[task_num].lastCalled = micros();
407 Tasks[task_num].Delay = 0;
408 Tasks[task_num].state = READY;
409 Tasks[task_num].state2 = RDY;
410
411 task_num++;
412 number_of_tasks=task_num;
413 StackPt-=IDLE_STLEN;
414
415 // Serial.println((int)StackLow);
416 // Serial.println((int)IDLE_STLEN);
417 // Serial.println((int)StackPt);
418 Serial.print("Idle"); Serial.print(-IDLE_STLEN); Serial.print(F(": Stack free for next task:
419 )); Serial.println((int)(StackPt-StackLow));
420 }
421
422
423 Tasks[task_num].name = _name;
424 Tasks[task_num].function = _function;
425 Tasks[task_num].prio = _prio;
426 //Tasks[task_num].task_stack = _task_stack;
427 Tasks[task_num].task_stack = StackPt;
428 StackPt-=_stackLen;
429 Tasks[task_num].stackLen=_stackLen;
430
431 if ( StackPt < StackLow ) {
432 //MySerial.println(F("ERROR: STACK too small !"));
433 Serial.println(F("ERROR: STACK too small !"));
434 while(1);
435 }
436
437 //Serial.println((int)_stackLen);
438 //Serial.println((int)StackPt);
439 Serial.print(_name); Serial.print(-_stackLen); Serial.print(F(": Stack free for next task: ")); Serial.
440 println((int)(StackPt-StackLow));
441
442 Tasks[task_num].new_task = true;
443 Tasks[task_num].lastCalled = micros();
444 Tasks[task_num].Delay = _delay;
445 if (_delay != 0) _state=BLOCKED;

```

```

446   Tasks[task_num].state = _state;
447
448   task_num++;
449   number_of_tasks=task_num;
450   return task_num-1;
451
452 }
```

#### 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.

```

192 {
193   // Save old
194   cli();
195   pushall();
196   Tasks[old].sp_save = SP;
197
198   sei();
199   while (1)
200   {
201
202     current_task=newer;
203
204     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)
205     {
206       // All tasks in Init(...); will come to this point
207       // at first run. All initialisation before while(1) are wil be done now
208       cli();
209       Tasks[current_task].new_task = false;
210       SP = Tasks[current_task].task_stack;              // set SP to the new
task. It will save it's stack when Yielded
211       //BITCLEARD6;
212       sei();
213       Tasks[current_task].function ();                  // run the new task.
It should not return ( starting a while(1)-loop)
214
215       // a task has ended!
216       Tasks[current_task].state = BLOCKED;              // all finished
task ( jumped out of while(1) ) will end here
217       Tasks[current_task].new_task == true;            // if resumed they
start again
218     }
219     else
220     {
221       // this task had been started and has save stack    // Yield had select the next task
222       cli();                                              // switch to the SP of this task, own stack is
saved (s.a. pushall )
223       SP = Tasks[current_task].sp_save;                  // switch the stack to
the save stackpt of next task to execute
224       sei();
225
226       popall();                                          // pop saved registers and return -> goto
last Yield-Point of switched out task
227       //BITCLEARD6;
228       return;
229     }
230   }
231 }
```

#### 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.

```

249                                     {
250 unsigned long m=micros();
251 struct task *tp, *tp2;
252 SwitchCount++;
253 //BITSETD6;
254 tp=&Tasks[current_task];
255 tp->lastCalled=m;
256
257 Start:
258
259 if (current_task==0) {                                     // idle
260     Tasks[0].state2=RDY ;
262     for (int i=1; i < MAX_TASKS; i++) {
263
264         tp2=&Tasks[i];
265         //if (Tasks[i].stopped==1) {
266         if (Tasks[i].state==BLOCKED) {
268             if (Tasks[i].Delay) {
269                 if ((m-tp2->lastCalled)>=Tasks[i].Delay) {
270
271                     //tp2->stopped=0;
272                     tp2->Delay=0;
273                     tp2->state=READY;
274                     tp2->state2=RDY;
275
276                     // while(1);
277                 }
278             }
279         }
280     }
281
282
283
284     //MySerial.println(i);
285
286 }
287 }
288 }
289 else {
290     //Tasks[current_task].lastCalled=micros();
291     if (tp->state==READY) {
292         if (mics>0) {
293             tp->Delay=mics;
294             //tp->stopped=1;
295             tp->state = BLOCKED;
296             tp->state2 = DEL;
297         }
298     }
299 }
300 }
301
302
303
304 // Search for the next task
305 int prio=0;
306 int oldTask=current_task;
307 // do
308 // {
309 //     /* Circle through the tasks, the if() construct is much
310 //     faster than using a mod % operator. */
311 //     if (++current_task == number_of_tasks)
312 //         current_task = 0;
313 //
314 //     /* Skipping inactive tasks. It is not uncommon for the system
315 //     to spend most of it's time chasing its tail going through a
316 //     list of all inactive tasks. This is a good place to add code
317 //     for putting the processor in a low power state. It is also
318 //     a good place to modulate an output pin to make a PWM measurement
319 //     of processor loading. */
320 // } while (Tasks[current_task].state == BLOCKED);
321
322 oldTask=current_task;
323 char HiPrio=0, HiNum=0;
324 extern char number_of_tasks;
325 for (int i=1 /* not Idle */; i < number_of_tasks; i++) {
326     if (Tasks[i].state== READY) {
327         if (Tasks[i].prio > HiPrio) {
328             HiPrio=Tasks[i].prio;
329             HiNum=i;
330         }
331     }
332 }
333 current_task=HiNum;
334
335
336 if (oldTask==current_task) {
337     tp->state=READY;
338     tp->state2=RUN;
339     //BITCLEAR6;
340     // no TaskSwitch needed

```

```

341     return;
342 }
343 else {
344     //current_task=oldTask;
345     //Tasks[oldTask].state=BLOCKED;
346     Tasks[current_task].state=READY;
347     Tasks[current_task].state2=RUN;
348
349     TaskSwitch(oldTask, current_task);
350 }
351 }

```

#### 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**

