Everything in one place

CoRoutines does the job

Assume a standard Blinky demo

```
void setup() {
  pinMode(led13, OUTPUT);           // initialize the digital pin as an output.
}
void loop() {
  digitalWrite(led13, HIGH);        // turn the LED on (HIGH is the voltage level)
  delay(1000);                      // wait for a second
  digitalWrite(led13, LOW);         // turn the LED off by making the voltage LOW
  delay(1000);                      // wait for a second
}
```

Now this one will light led13 for 1 second, and keep it shut for 1 second.

Now assume just another Blinky demo, on led12, blinking somewhat faster

```
void setup() {
  pinMode(led12, OUTPUT);           // initialize the digital pin as an output.
}
void loop() {
  digitalWrite(led12, HIGH);        // turn the LED on (HIGH is the voltage level)
  delay(100);                       // wait for a second
  digitalWrite(led12, LOW);         // turn the LED off by making the voltage LOW
  delay(200);                       // wait for a second
}
```

But how do we get both demo's to run at the same time, having exactly the same light sequences.

We could rebuild the whole program, like using another programming approach like

- Keeping milliseconds variables handy and check every 1 mSec to see if now is a good time to flip the power.
- Use state variables
- Use a lot of "flags"
- Use CoRoutines

This CoRoutines presented here is my best efford to make Arduino CoRoutine as easy to use as an Arduino program without CoRoutines.

The problem above can be programmed as elegant as this

```
void setup() {
  hsTASK_SETUP (Blinky1, 100);      //One CoRoutine, with its own stack of 100 bytes to use
  hsTASK_SETUP (Blinky2, 100);      //One CoRoutine, with its own stack of 100 bytes to use
}
```

```
hsTASK_INIT(Blinky1) {
  if (Init) pinMode(led13, OUTPUT); // initialize the digital pin as an output.
  digitalWrite(led13, HIGH);        // turn the LED on (HIGH is the voltage level)
  delay(1000);                      // wait for a second
  digitalWrite(led13, LOW);         // turn the LED off by making the voltage LOW
  delay(1000);                      // wait for a second
}

hsTASK_INIT(Blinky2) {
  if (Init) pinMode(led12, OUTPUT); // initialize the digital pin as an output.
  digitalWrite(led12, HIGH);        // turn the LED on (HIGH is the voltage level)
  delay(100);                       // wait for a second
  digitalWrite(led12, LOW);         // turn the LED off by making the voltage LOW
  delay(200);                       // wait for a second
}

void loop() {
  Serial.Print("Something");        // Not used here, but you need to have one loop()
  delay(10000);
}
```

That's it, that's that.

CoRoutines

The magic is happening in the delay() routines. Normally the Arduino as doing nothing but waiting for say 1000 milliseconds, 1 second just burning off some heat, pure waste of battery power. The magic is that delay() never uses "burn cpu" but instead run some other functions that might be ready to do real work, and if no one is ready then eventually turn off the cpu.

Chapter 2

Simple programming

A lot of coding can be done **without** coroutines (and multi-threading and such).

A lot of coding can be done **with** coroutines and simple use of variables and flags, and I have done many of that kind.

Over time (last 50 years) some coding principles has been developed to help making different parts of a program work together in a safe way.

- Semaphores, used for region protection, and counting too
- Signals, tell when things happens
- Mails, send simple messages (a single number)

Semaphore

A semaphore here is a byte variable, initialized as required.

When initialized to one, the normal use is for region protection, as everybody wanting to use some resource has to decrement then semaphore variable, but if already zero you have to stand in line until the one having the "token" does return it (increment the semaphore to 1)

When initialized to zero (or whatever), it is typical used for counting, like number of items processed, ready for next step. A cookie baker could increment on every cookie produced, and the packer could decrement (when a new cookie shows up), counting to 8 for a full box of cookies.

Signals

A signal in this implementation takes no memory, it is just a shared constant.

Multiple watchers might be interested in when event NOW_OK has happened. When NEW_OK is signaled, everyone waiting is released for instant execution (all, but one at a time as we still only have one cpu, so must act cooperative)

Mails, sending messages

Mails does require a small memory buffer, shared for all mails.

When sending one mail, exactly one receiver gets it, if waiting for it already, or if trying to wait after mail has been posted.

hsCoRoutine, this implementation

This implementation is

- very clean
- very small
- very simple api
- can be used from anywhere, full stack and locals are saved
- (most others and protoThreads too does not save stack)
- signaling can be done from interrupts
- uses very little assembler code, and lets gcc compiler do the register savings for best conformance
- uses about 10 bytes per coroutine, and user defined amount for the stack
- uses

Having said this, the minus is that hsCoRoutines are **not** fair, like it does not guarantee that first in line also gets first served

API

Basics

Semaphores

Signals

Mails

Examples

Cookie baker

Regions

Signals, and interrupt

OS

Structure

SP	Int	
TimeOut	Int	
Flags	Byte	E_RTR=1, E_TOut=2, E_Released=3, E_Sema=4, E_Event=5, E_Mail=6
Obj	Int	Sema#, Event#, Mail#

Semaphore, Wait region, Counting semaphore

```
hsSemaWait(TOut, & SNo): Boolean
 if [SNo] = 0 {
  Flags = E_Sema
  TimeOut = TOut
  Obj = SNo
  Swap
  If Flags != E_Released ClrFlags; return 0 //TOut!
 }
 Critical-dec([SNo])
 ClrFlags;
 return 1

hsSemaSignal(& SNo);
 DI
 [SNo]++
 Signal(SNo,E_Sema) //signal one
 EI (restore)
 Swap

int_hsSemaSignal(SNo);
 DI
 [SNo]++
 Signal(SNo,E_Sema) //signal one
 EI(Restore)
```

Event = Signal all

```
hsEventWait(TOut,ENo): Boolean
  DI
  Flags = E_Event
  TimeOut = TOut
  Obj = ENo
  EI
  Swap
  If Flags != E_Released ClrFlags; return 0 //TOut
  ClrFlags;
  return 1

hsEventSignal(ENo)
  Signal(ENo,E_Event) //signal all
  Swap

int_hsEventSignal(ENo)
  Signal(ENo,E_Event) //signal all
```

Mail, Send int message to named (byte) queue

```
hsMailWait(int TOut, byte MNo, &Result): Boolean
  Flags = E_Mail
  TimeOut = TOut
  Obj = MNo
  while (1) {
    if Result=GetQ(MNo) then ClrFlags; return 1
    Swap
    If Flags != E_Released then ClrFlags;  return 0 //TOut
  }

hsMailSend(MNo, Val);
  AddQ(MNo, Val)
  Signal(MNo,E_Mail) //Signal one
  swap
```

```
int_ hsMailSend(MNo, Val);
  AddQ(Val)
  Signal(MNo,E_Mail) //Signal one
```

hsSwap

```
hsSwap()
Save PrevSP
Check newTaskIndex
DI EI…
While 1 {
  UpdateTimers(), set E_TOut on Timer <= 0
  Find one to run
}


Signal(int AObj, byte AFlag)
For II=0 to TaskMax do
  If (Obj=AObj) and (Flag=AFlag) {
    ClrFlags;
    Flag = E_Released
    If AFlag = E_Sema break; //Signal one only
  }

ClrFlags()
    Flags = 0
    hsDelay = 0
    Obj = 0
```