servoMoba
Public methods & variables

Public methods

checkServo();
setTreshold1(value);
setTreshold2(value);
initPower(idlePowerIsOff, powerEnablePin, powerEnableValue)

Public variables

idlePulseDefault

powerEnableValue tells if the MOSFET / chip that switches the servo power on, expects a high or a low value to enable the 5V for the servo

Private variables

for the start and finish phase
(subset)

servoState

Default values

- idlePulseDefault
- pulseOnBeforeMoving
- pulseOffAfterMoving
- idlePowerIsOff
- powerOnBeforeMoving
- powerOffAfterMoving

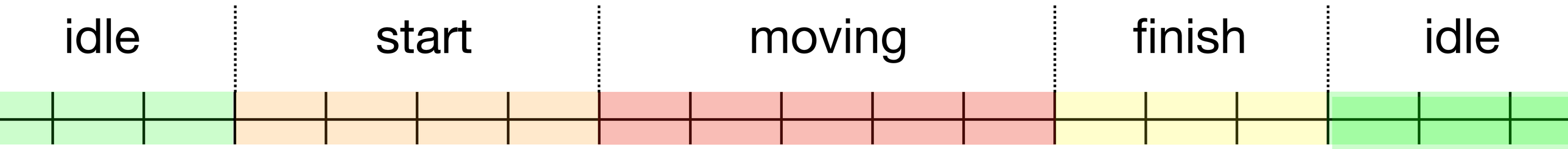
Counters for the start phase

- countServo
- countPulse
- countPower

Counters for the finish phase

- countPulse
- countPower

servoState



if needed:
- *power on*
- *pulse start*

servo moves

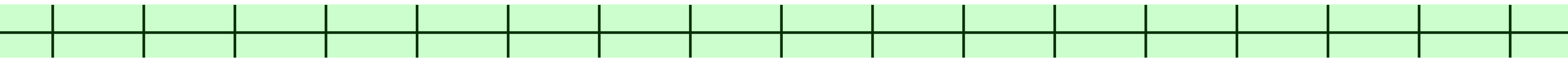
if needed:
- *power off*
- *pulse stop*

```
enum state_t {
  idle,
  start,
  moving,
  finish
} servoState;
```

Note: relais for frog polarisation have no relationship with servo's, and will therefore not be included in the ServoMoba class

Default values for the idle state
servo power and servo pulse (PWM signal)

idle

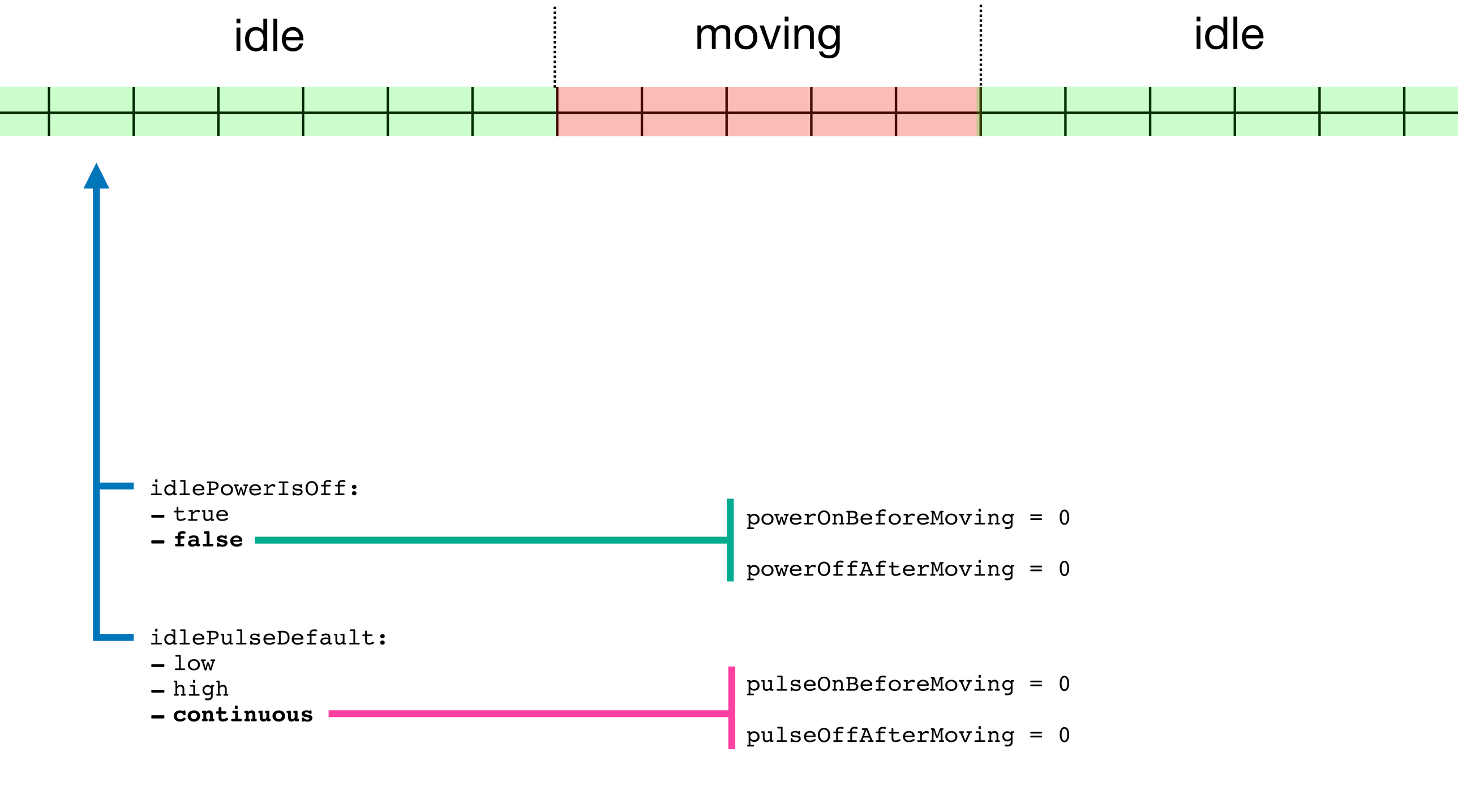


```
idlePowerIsOff:
- true
- false
```

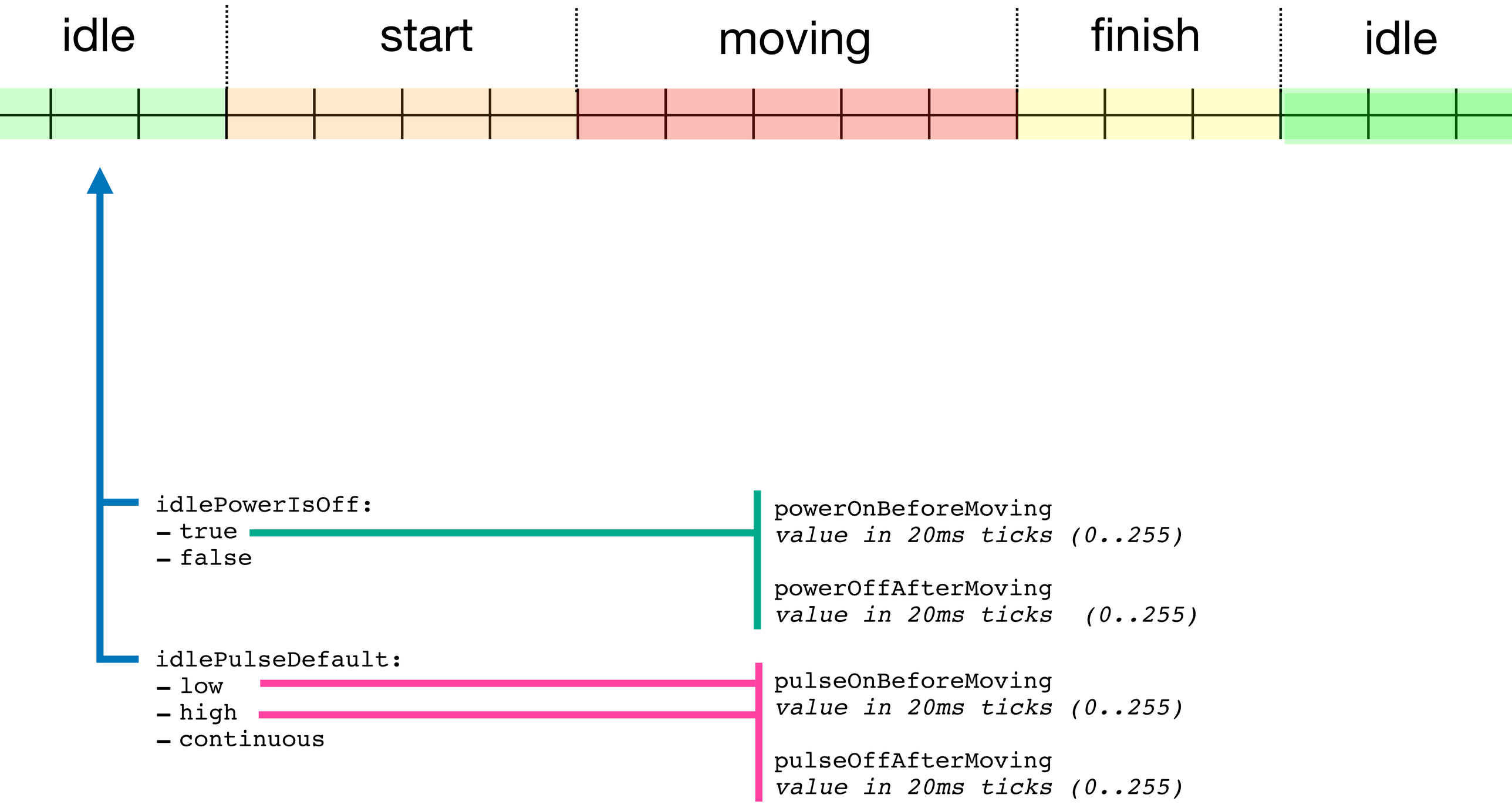
```
idlePulseDefault:
- low
- high
- continuous
```

```
enum idlePulseDefault_t {
  low,
  high,
  continuous
} idlePulseDefault;
```

Default values for the servo power and the servo pulse (PWM signal) while in the idle state
No start nor finish phase



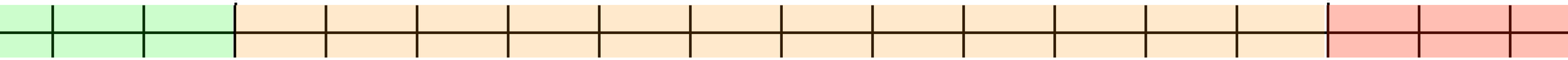
Default values for the servo power and the servo pulse (PWM signal) while in the idle state
With start and finish phase



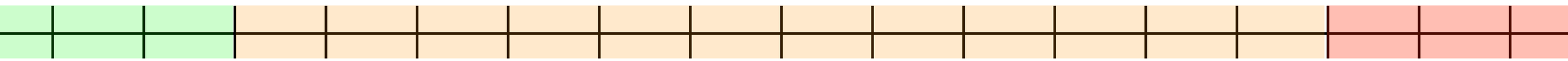
idle

start

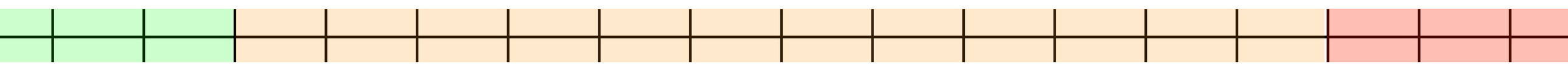
moving



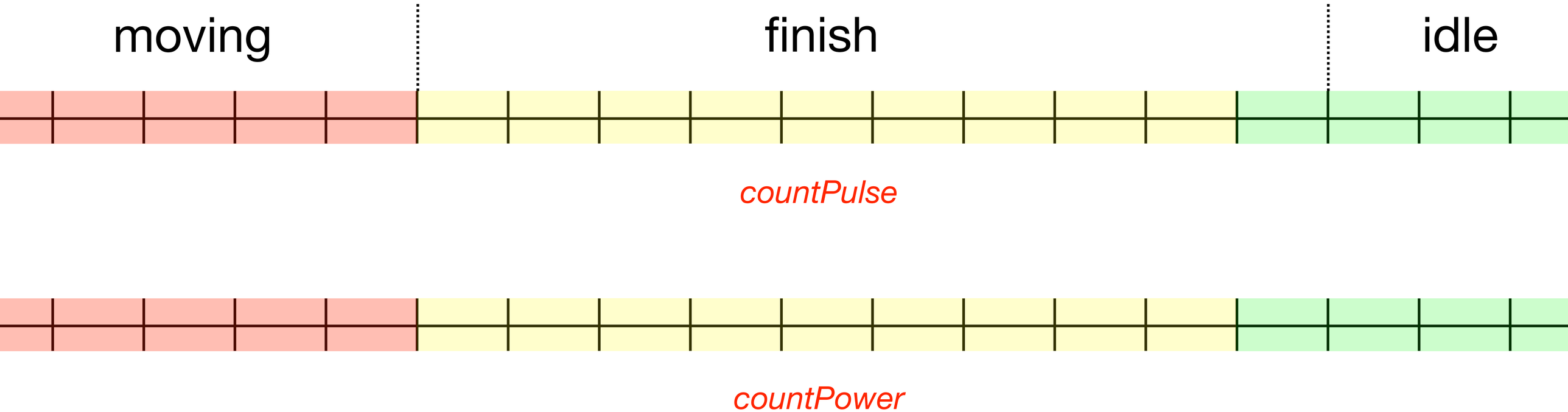
countServo



countPulse



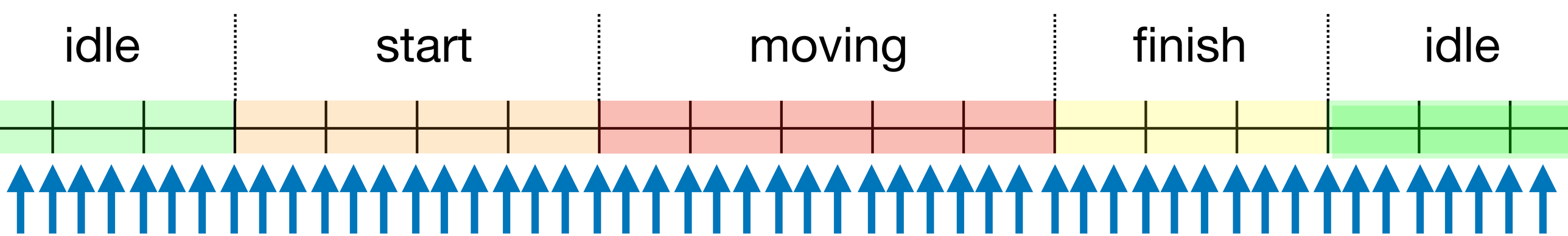
countPower



Note: no countServo is needed

Phases

servoState




```
void checkServo()
  if (acceptsNewValue()) {
    if (PowerOnNextTick) powerOn();
    if (PowerOffNextTick) powerOff();
    switch (servoState) {
      case idle: servoIdle();
      break;
      case start: servoStart();
      break;
      case moving: servoMoving();
      break;
      case finish: servoFinish();
      break;
    };
    waitTillNextPulse();
  };
}
```

Start phase

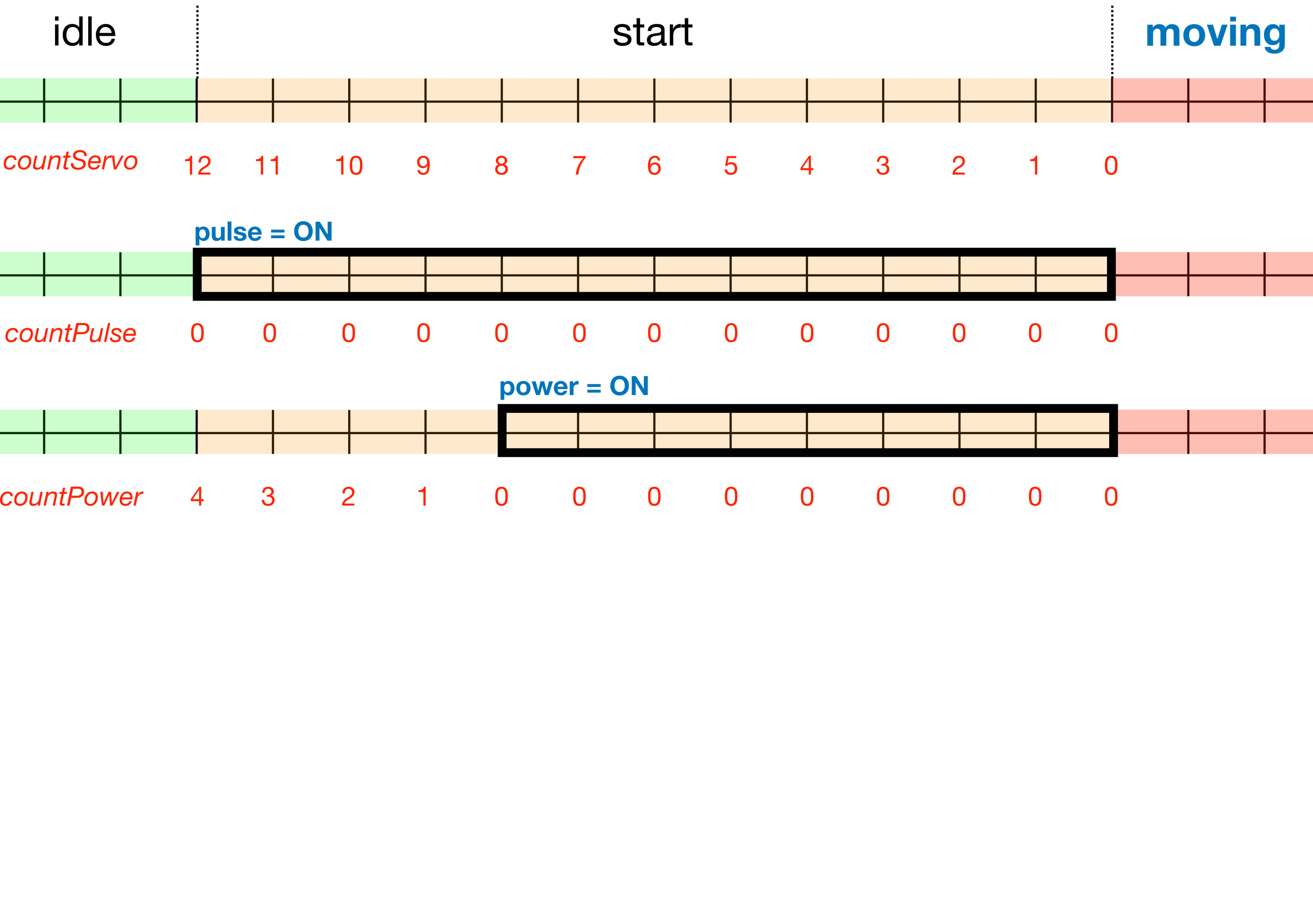
variables:

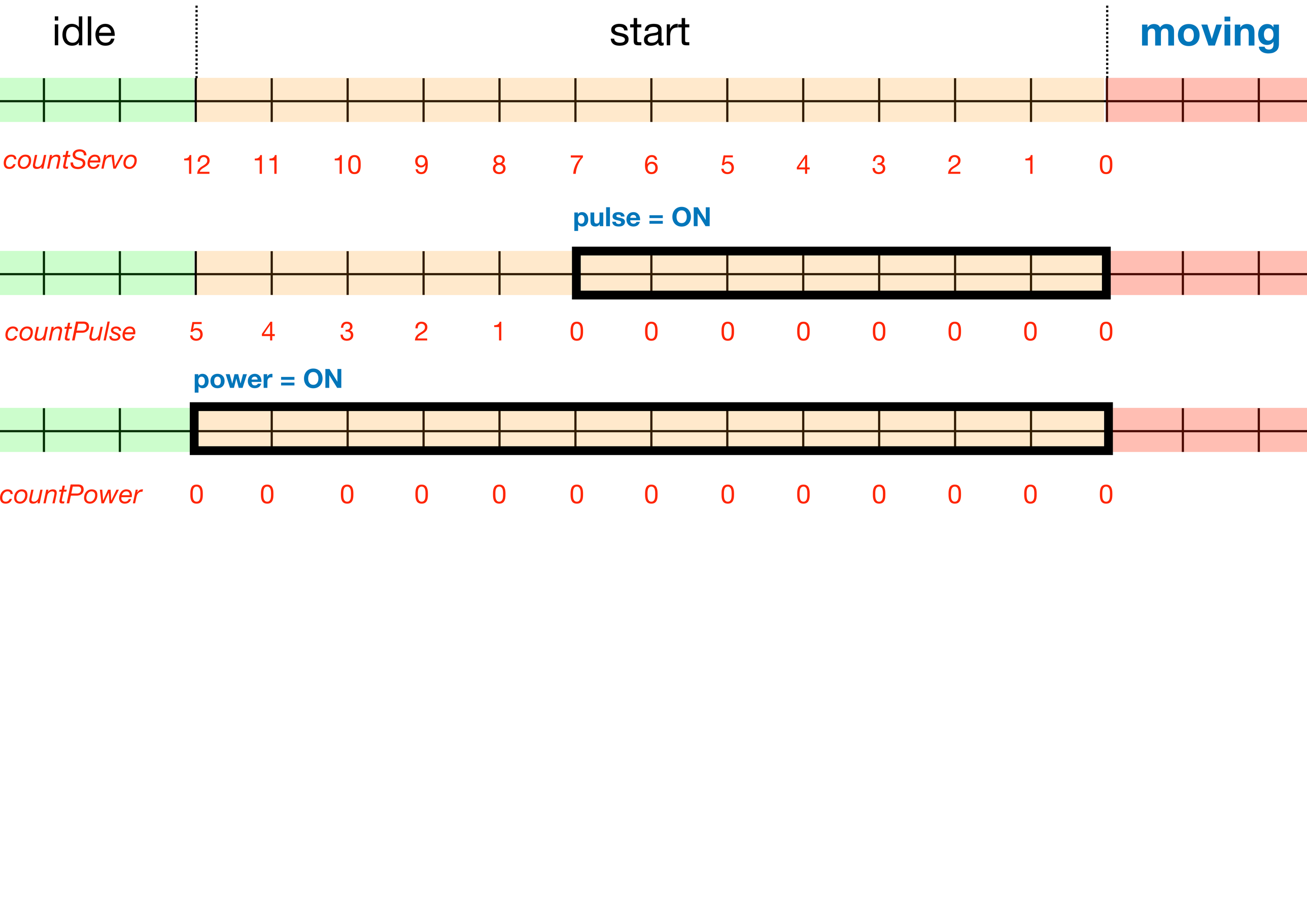
- countServo
- countPulse
- countPower

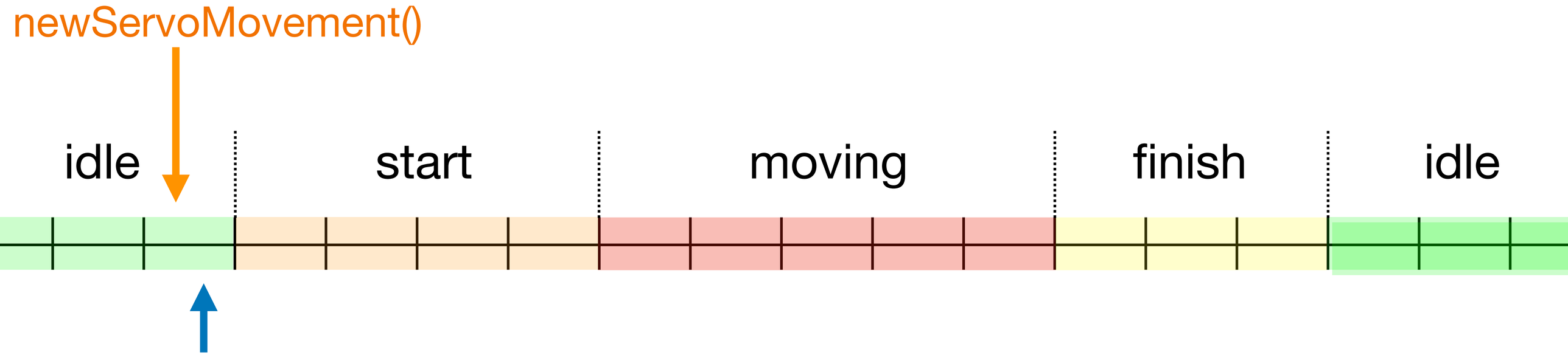
variable initialisation

scenarios:

- powerOnBeforeMoving = pulseOnBeforeMoving = 0
- powerOnBeforeMoving < pulseOnBeforeMoving
- powerOnBeforeMoving >= pulseOnBeforeMoving

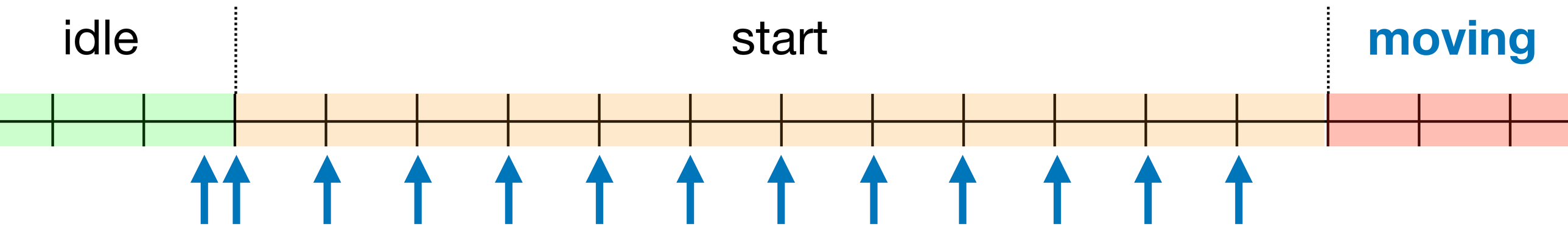

idle
start
moving
countServo
12 11 10 9 8 7 6 5 4 3 2 1 0
pulse = ON
countPulse
0 0 0 0 0 0 0 0 0 0 0 0 0
power = ON
countPower
4 3 2 1 0 0 0 0 0 0 0 0 0


idle
start
moving
countServo
12 11 10 9 8 7 6 5 4 3 2 1 0
pulse = ON
countPulse
5 4 3 2 1 0 0 0 0 0 0 0 0
power = ON
countPower
0 0 0 0 0 0 0 0 0 0 0 0 0



checkServo() => servoStart()

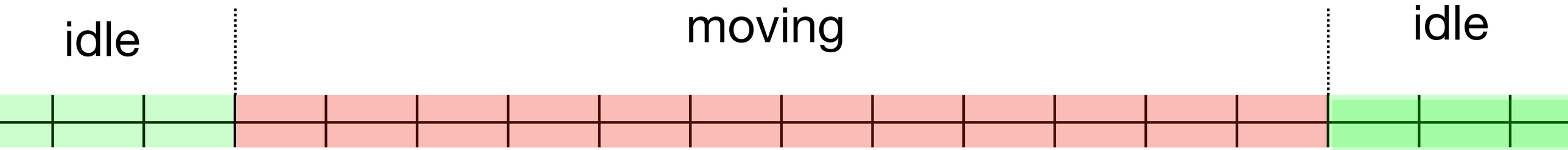
```
void initStart(){
  if (powerOnBeforeMoving >= pulseOnBeforeMoving) {
    countServo = powerOnBeforeMoving;
    countPulse = powerOnBeforeMoving - pulseOnBeforeMoving;
    countPower = 0;
  }
  else {
    countServo = pulseOnBeforeMoving;
    countPulse = 0;
    countPower = pulseOnBeforeMoving - powerOnBeforeMoving;
  };
   servoState = start;
   servoStart();
};
```




```
servoStart() {
  if (countPulse > 0) countPulse--;
    else write(lastPulseWidth);
  if (countPower > 0) countPower--;
    else if (idlePowerIsOff) PowerOnNextTick = true;  // switch power on
  if (countServo > 0) countServo--;
    else {
      servoState = moving;
      servoMoving();
    }
};
```

Note 1: since digitalWrite() is expensive, we check idlePowerIsOff
Note 2: we must wait till the next tick before the servo power can be switched on (or off)

Scenario: idlePowerIsOff & idlePulseDefault == continuous



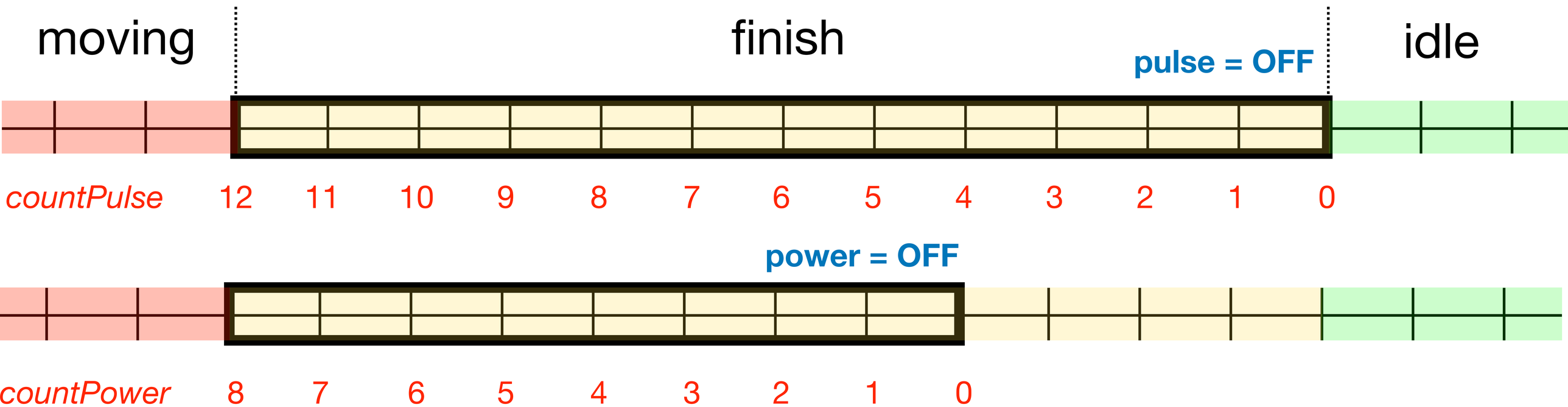

```
Case start:
  if (countPulse > 0) countPulse—;
    else write(lastPulseWidth);
  if (countPower > 0) countPower—;
    else if (idlePowerIsOff) PowerOnNextTick = true;
  if (countServo > 0) countServo--;
    else {
      servoState = moving;
      servoMoving();
    }
```

```
if (powerOnBeforeMoving >= pulseOnBeforeMoving) {
  countPower = 0;
  countPulse = 0;
  countServo = 0;
}
else
  …
}
```

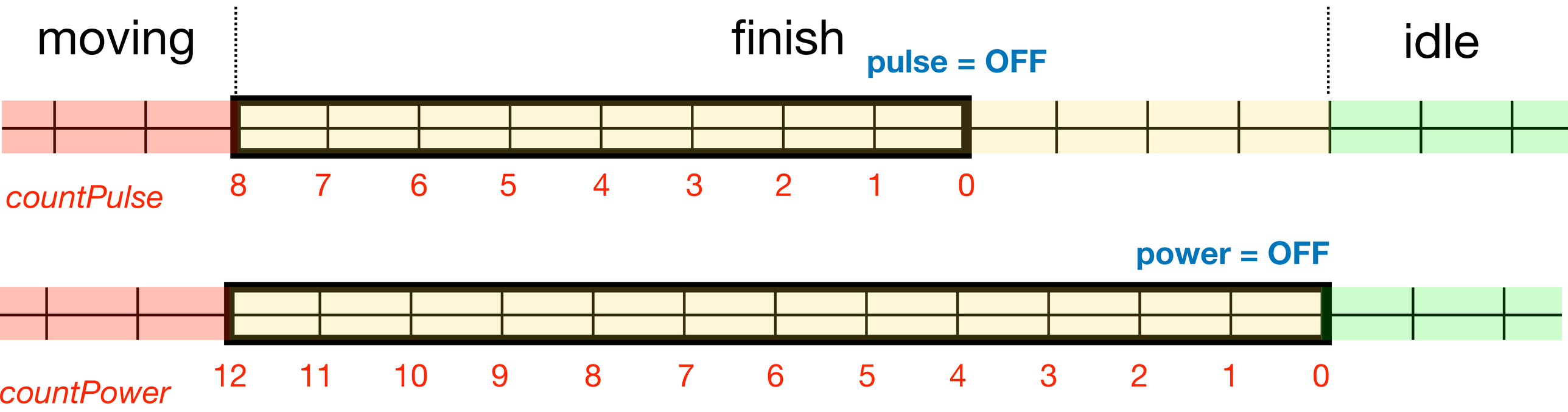
```
idlePowerIsOff:
– true
– false ──┬── powerOnBeforeMoving = 0
          └── powerOffAfterMoving = 0

idlePulseDefault:
– low
– high
– continuous ──┬── pulseOnBeforeMoving = 0
               └── pulseOffAfterMoving = 0
```

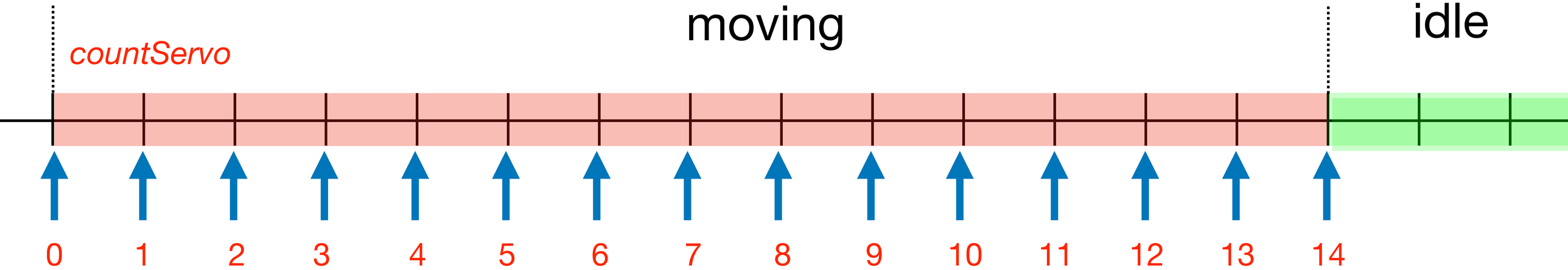
Finish phase




```
servoFinish() {
  bool move2idle = ((countPulse == 0) && (countPower == 0));
  if (countPulse > 0) countPulse--;
    else {
      if (idlePulseDefault == low) constantOutput(0);
      if (idlePulseDefault == high) constantOutput(1);
    }
  if (countPower > 0) countPower--;
    else {
      if (idlePowerIsOff) PowerOffNextTick = true;
    }
  if (move2idle) {
    servoState = idle;
    servoIdle();
  }
}
```


moving
finish
idle
pulse = OFF
8 7 6 5 4 3 2 1 0
countPulse
power = OFF
12 11 10 9 8 7 6 5 4 3 2 1 0
countPower

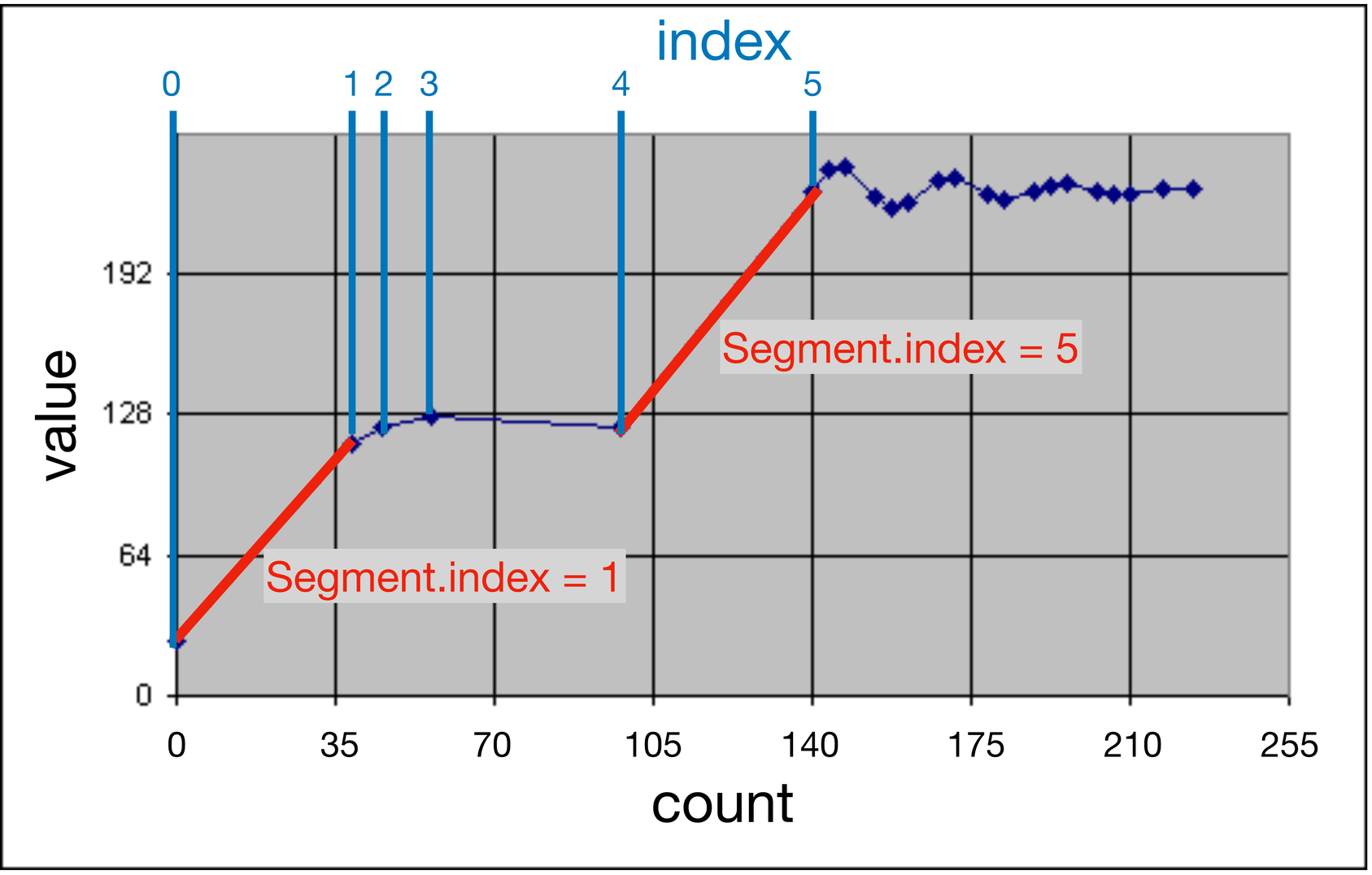
Moving phase



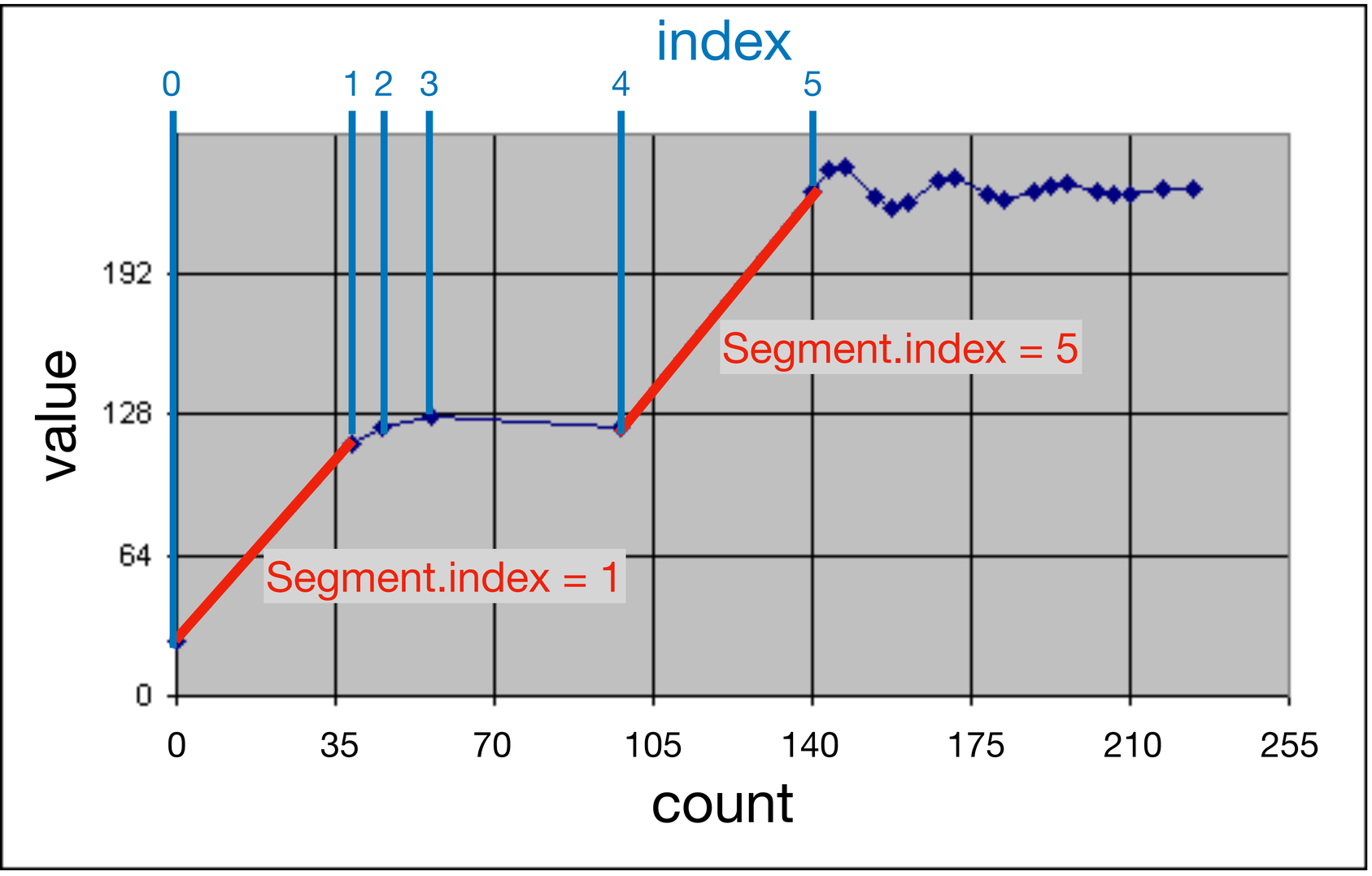

```
servoMoving() {
  …
  if (countServo == 0) state = idle;
}
```

Wat is handiger????

```
servoMoving() {
  …
  if (countServo == last) state = idle;
}
```



index	count	value
0	0	25
1	38	105
2	45	120
3	60	127
4	98	122
5	140	230
...	...	...
22	0	0
x	x	x



servoCurve

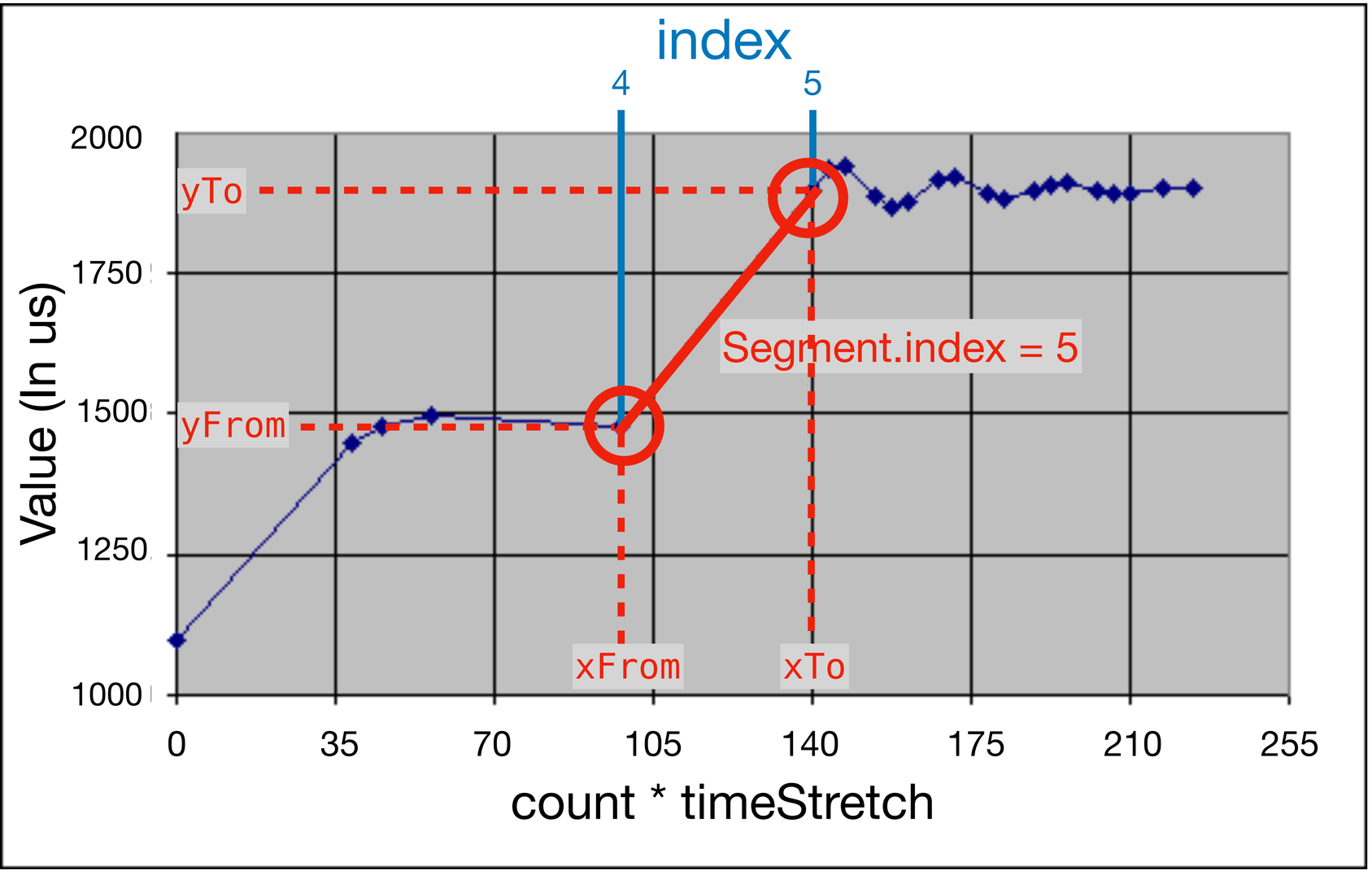
index	count	value
0	0	25
1	38	105
2	45	120
3	60	127
4	98	122
5	140	230
...	...	...
22	0	0
x	x	x

```
typedef struct {
  uint8_t count;
  uint8_t value;
} curvePoint_t;
```

```
#define SIZE_SERVO_CURVE      24
curvePoint_t servoCurve[SIZE_SERVO_CURVE];

uint16_t treshold1;       // in us
uint16_t treshold2;       // in us
uint16_t tresholdDelta = treshold2 - treshold1;

uint16_t valueTo_us(uint8_t value) {
  return (value * (long)(treshold2 - treshold1) / 255 + treshold1);
};
```




```
typedef struct {
  uint8_t count;
  uint8_t value;
} curvePoint_t;

typedef struct {
  uint8_t index;
  uint16_t xFrom;
  uint16_t xTo;
  uint16_t xDelta;
  uint16_t yFrom;
  uint16_t yTo;
  uint16_t yDelta;
} segment_t;

segment_t segment;
```

```
segment.index = 5;
segment.xFrom = servoCurve[index-1].count * timeStretch;
segment.xTo = servoCurve[index].count * timeStretch;
segment.xDelta = segment.xTo - segment.xFrom;
segment.yFrom = valueTo_us(servoCurve[index-1].value);
segment.yTo = valueTo_us(servoCurve[index].value);
segment.yDelta = segment.yTo - segment.yFrom;
```

Calculate the Y position, for a given X