# JuneBug – BoostC - C18 Tutorials : Cooperative Multitasking (1st Draft, 4rd Edit)

## Table of Contents

Cooperative Multitasking
Daniel Johnson 2008©
An introduction to cooperative multitasking on the PIC18F1320 using the BoostC or C18 compilers.
1
2
3

**I**

# JuneBug – BoostC - C18 Tutorials : Cooperative Multitasking (1st Draft, 4rd Edit)

## ntroduction

A single core/CPU computer is sequential. It is only capable of doing one thing at a time. To do several tasks at the same time multitasking is used.

On a PC users are free to install and run any number of programs preemptive multitasking is a better choice. A PC can not ensure that programs will be well behaved. If one program or driver has problems the entire system can fail.

In the situation where all code is well behaved cooperative multitasking can out preform preemptive multitasking. The cooperative system switches faster and less frequently. A uC running firmware is that sort of system.

This tutorial explores cooperative multitasking with NAOS. “Not An OS”. It requires no software other then your C compiler. NAOS is bare bones bag of techniques and code snippets designed to illustrates Cooperative Multitasking principals.

Complete code for this tutorial can be found in the appendix or at http://www.rocklore.com/3v0. The zip archive contains MPLAB project files for BoostC and C18. The single source is written to work with both compilers.

The code snippets provided within the main body of the tutorial are for illustration purposes and may not compile.

### *What you must know*

To understand this tutorial you need to be familiar with: static variables and switch statements. Timer0 is used, but it is not necessary to fully understand how to set it up. *may add more*

### *What you will need*

A Junebug PIC Laboratory from BlueroomElectronics

or

one of (Microchip ICD2 , PICkit2®) and a breadboard with a PIC18F1320 processor.

Everyone will need a small speaker and a 47uF capacitor.

# JuneBug – BoostC - C18 Tutorials : Cooperative Multitasking (1st Draft, 4rd Edit)

***Definitons***

**Task**

There is not set standard that tells us what a task is.  Each OS or programer determines what a task is and what set of tasks to use.  Tasks range from simple to complex. Tasks may be as simple as blinking an LED or an entire program.

**Sequential Execution**

aaaa

**Paraellel Execution**

aaaa

**Multitasking**

The ability to execute several tasks at the same time, or give the appearance of doing so.

**Cooperative Multitasking**

Cooperative multitasking is a method that allows a single CPU to share its time between two or more tasks.  Each task voluntarily passes control to the next.

**Preemptive (Time Slice) Multitasking**

The ability to execute several tasks at the same time, or give the appearance of doing so.

**Finite State Machine**

aaaa

*More definitions here if needed*

# JuneBug – BoostC - C18 Tutorials : Cooperative Multitasking (1st Draft, 4rd Edit)

```
// Sequential
// with subroutines

void blink(void)
{
  ledBitOn();
  delay_10th(5);
  ledBitOff();
  delay_10th(5);
}
void beep(void);
Void main()
{
  // main loop
  while(1)
  {
    blink();
    beep();
  }
}

Code 1:
```

## Sequential Execution

Code 2 is a typical sequential program. It flashed the LED, then it beeps, flashed the LED, then it beeps, etc.

We will use this code as the starting point as we explore cooperative multitasking.

## Cooperative Multitasking

We named the two tasks taskBlink() and taskBeep(). Each task is like a separate program. They do not take input parameters or return a value.

The main loop checks for tasks which are ready to run and calls them. This allows tasks to run while others are delaying or waiting.

```
// NAOS
// Cooperative Multitasking

void taskBlink(void)
{ ... }
void taskBeep(void)
{ ... }
Void main()
{
  // main loop
  while(1)
  {
    if(BLINK_READY_TO_RUN)
      taskBlink();
    if(BEEP_READY_TO_RUN)
      taskBeep();
  }
}

Code 2:
```

# JuneBug – BoostC - C18 Tutorials : Cooperative Multitasking (1st Draft, 4rd Edit)

## NAOS Tasks

The definitions box provided a definition for task. In short it said a task can be whatever you need it to be. One can not be more specific without knowing the multitasking environment. Now that we have established NAOS as our environment we can provide additional details.

### *Finite State Machines*

To convert a function to a task we need to view it as a Finite State Machine or FSM.

A FSM consists of states connected by arrows, or edges, to indicate how we move between them.

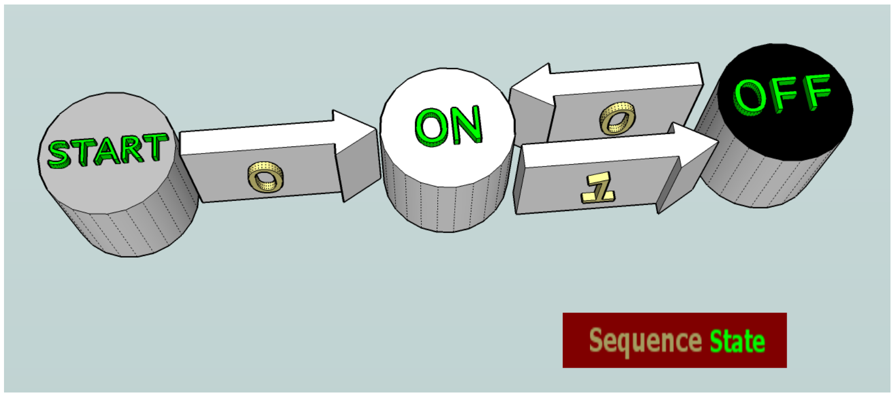

*Illustration 1:*

Illustration 1 shows the FSM for Code1 function blink() which has been repeated in Code 3.

A task may not contain delay code. When converting a function to a task each delay statement indicates the end of a state.

Every delay in the code result in a edge between two states. The 1st state contains the code prior to the delay, the 2nd state contains the code between the two delays.

Three states describe the workings of Code 3 `blink().`The two obvious states are **ON (**LED illuminated), and **OFF (**LED not illuminated). For clarity we add the **START** state.

```
// Sequential
// with subroutines

void blink(void)
{
  ledBitOn(); // ON
  delay_10th(5);
  ledBitOff(); //OFF
  delay_10th(5);
}
```

Code 3:

**START** has an edge (arrow) to **ON**, which has an edge to **OFF**, which has an edge to **ON**. It is easy to see why the LED blinks.

The **START** state is preformed during program startup. The first time `blink()` is called it will enter the ON state.

```
#define STATE case
void taskBlink(void)
{
  static byte seq=0;
  switch (seq)
  {
    STATE 0: // LED_ON
      ledBitOn();
      seq=1;
      break;
    STATE 1: // LED_OFF
      ledBitOff();
      seq=0;
    }
    // missing timer code
    return;
}
Void main()
{
  while(1) // main loop
  {
    if(BLINK_READY_TO_RUN)
      taskBlink();
    if(BEEP_READY_TO_RUN)
      taskBeep();
    while(waitForTimer);
    waitForTimer = TRUE;
  }
}
Code 4:
```

In Code 4 we have rewritten function `blink()` to work with NAOS. It has been renamed `taskBlink()` to remind us that it is a task. We have defined STATE to mean case for clarity.

"`if(BLINK_READY_TO_RUN)`" in Code 4 calls `taskBlink()`

Each time `taskBlink()` is called the state corresponding to the value of seq is executed. The called state must set the value of seq, which determines which state will be entered next time taskBlink() is called.

`taskBlink()` must be called twice to make the LED blink once.

## It's All About Timing

Did you know that chef's use kitchen timers to multitask? After setting the timer the chef is free to work on another task. When the timer expires the chef resumes the task for which the timer was set.

Code 5 creates and manages three timers for us. To avoid confusion with the PICs internal timers we will call our timers kTimers, as kitchen timer or "kount down timer".

The function `interrupt()` runs each time the free running PIC hardware timer TMR0 rolls over to zero. The speed at which TMR0 increments is governed by the system clock frequency and the TMR0 prescaler setting.

When `interrupt()` executes each `kTimer` with a value greater the zero is decreased by 1. We will discuss `TASK_BLOCKED` in another section.

```
#define uInt unsigned int
#define byte unsigned int

#define TASK_BLOCKED    \
                0xFFFF
#define KMAX 3
#define BLINK 0

uInt kTimer[KMAX];
byte waitForTimer;

// allocate KMAX kTimers
uInt kTimer[KMAX];

// Update kTimers
void interrupt(void)
{
  byte k;
  // clear the IF
  intcon_.TMR0IF = 0;
  for (k=0;k<KMAX;k++)
  {
    // hold at 0
    if((kTimer[k]) &&
       (kTimer[k]!=
         TASK_BLOCKED))
    {
      kTimer[k]--;
    }
  }
  waitForTimer = FALSE;
}
Code 5:
```

## Multitasking = Tasks + Timing

```
void taskBlink(void)
{
  static byte seq=0;
  switch (seq)
  {
    state 0:
      ledBitOn();
      seq=1;
      break;
    state 1:
      ledBitOff();
      seq=0;
  }
  kTimer[BLINK]=2000;
  return;
}
void main()
{
  while(1) // main loop
  {
    if (!kTimer[BLINK])
      taskBlink();
    while(waitForTimer);
    waitForTimer = TRUE;
  }
}
Code 6:
```

`In Code 6 we upgrade taskBlink()` to work with kTimers. We replace

`if(BLINK_READY_TO_RUN)`

with functional code.

`If (!kTimer[BLINK])`

In task `taskBlink()` we set

`kTimer[BLINK]=2000`

A count of 2000 is equivalent to a ½ second delay. The LED blinks once per second. See the box kTimer's Math for details.

We do not use delays in tasks. But there is one at the end of main loop. To ensure that timing is uniform we wait here for the kTimers to change.

We now have everything need to run a single task in a multitasking way. In the next section we will add a 2nd task.

### *kTimer's Math*

Junebug's internal clock is set to 8,000,000.

The input to TMR0 is ¼ clock speed or 2,000,000.

With a prescaler of 2 TMR0 increments 1,000,000 times a second.

In the 8 bit mode TMR0 overflows 1,000,000/256 or 3,906 times per second.

Each time TMR0 rolls over kTimer[BLINK] decrements.

Each time we visit `taskBlink()` kTimer[BLINK] is set to 2000. We do not return to `taskBlink()` until kTimer[BLINK] has counted down to 0. Starting from 2000 it takes 0.512 seconds. 2000*(1/3906)

We call the `taskBlink()` twice per blink which results in a period of 1 .024 seconds.

If you need a period closer to 1 second you could reduce initial value of kTImer[Blink]. We know that each half cycle is .012 seconds too long. TMR0 interrupts evey 3,906 times per second. The inverse of that is 0.000256 to 3 significant digits. The number of kTimer counts needed to reduce the time by 0.512 seconds is (.012/.000256) or 47 in round numbers. 2000 less 47 is 1953.

Delays up to 16 seconds are possible.

## A Second Task

```
void taskBeep(void)
{
  static byte seq=0;
  switch (seq)
  {
    STATE 0: // state INIT
      kTimer[DUR]=500;
      seq=1;
      break;
    STATE 1: // state ON
      SPEAKER_BIT_ON;
      kTimer[BEEP]=10;
      seq=2;
      break;
    STATE 2: // state OFF
      SPEAKER_BIT_OFF;
      if(kTimer[DUR])
      {
        // another pulse
        kTimer[BEEP]=10;
        seq=1;
      }
      else
      {
        // generate a rest
        kTimer[BEEP]=1000;
        seq=0;
      }
    }
}
```

Code 7:

We can make a tone by rapidly pulsing an output bit connected to a speaker through a 47uF capacitor.

Two kTimers are required.

Timer `kTimer[BEEP]` is used to set the time when the task will next run.

Timer `kTimer[DUR]` is the duration of each tone.

We initialize `kTimer[DUR]` each time we generate a tone. This requires the additional state **INIT**.

Illustration *2* and the accompanying text explain how tastBeep works.

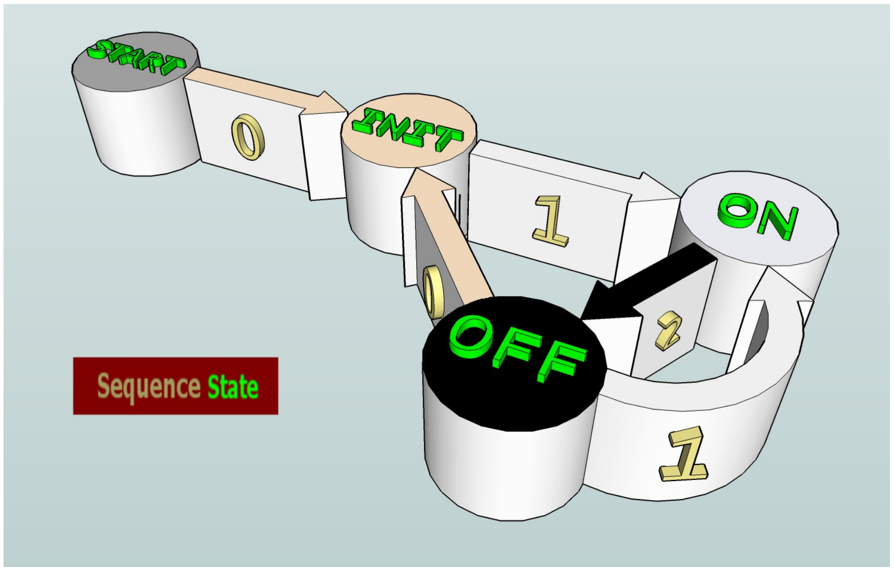

*Illustration 4:*

State **INIT** sets `kTimer[DUR]` to 500 and seq to 1.

State **ON** makes the speaker bit high, `kTimer[BEEP]` to 10 and seq to 2.

State **OFF** makes the speaker bit low.
If `kTimer[TONE]` has not expired `kTimer[BEEP]`is set to 10 and seq is set to 1.

If `kTimer[TONE]` has expired `kTimer[BEEP]` is set to 1000 which provides a period of silence. Seq is set to 0 which will cause INIT to start a new tone the next time it is called.

## Writting Complex Tasks

The simple tasks we have looked at so far resulted in 3 or 4 states. That might lead one to think that a longer more complex task would result in many more states. This is not necessarily the case.

### *Delaying States*

Anytime a tasks needs to delay it must set the associated kTimer and return. This code section is one state.

### *IO Blocked States*

When a task needs input to continue it sets the associated `kTimer` to `TASK_BLOCKED (0xFFFF)` and returns. The timer code does not decrement a timer with a value of `TASK_BLOCKED`. The task is unblocked by an ISR which sets the associated `kTimer` to 0. The either the ISR or the following state can perform the input operation. The setting

#### Task Variables

Earlier we said each task was like a `main()` function. An exception to this is local variables. Local variables declared in `main()` persist for the entire life of the program. We never exit main and the variables never go out of scope.

Task are called repeatedly. Each time a task exits the local variables go out of scope and become undefined.

To retain value between calls local variables are declared as static. The `seq` variable declared to each task is an example.

## Tuning the Code

With cooperative multitasking it is up to you the author to ensure that your tasks play well together.

The code presented in this tutorials requies just over 1 kTimer tick to make it through the main loop once.

xx

### *Macros*

`Macros can make the code easier to read but harder to debug.`

TASK

```
//x=timer y=taskName()
#define TASK(x,y) if(!kTimer[x]) y

while(1) // main loop
{
   // task list
   TASK(BLINK,taskBlink());
   TASK(BEEP,taskBeep());

   // sync
   while(waitForTimer);
   waitForTimer = TRUE;
}
```

## Apendix A: Source Code

### *DJ_coop2.c*

```
------------- start file DJ_coop2.c ---------------------------
/*
 * Junebug Demo
 * Purpose: Demonstrate cooporative multitasking
 *    1 Junebug LED + AC coupled speaker on RB1
 *
 * File: DJ_coop2.c
 * Software: BoostC or Microchip C18 compilers
 * Hardware: Junebug
 *  or other debugger + breadboarded circuit
 *
 * by Daniel Johnson
 * July 2008
 */

#include "multiCompiler.h"   // BoostC & C18
```

```
//
// defines & macros
//

#define TRUE  !0
#define FALSE 0

#define byte  unsigned char
#define uInt  unsigned int

#define state case
#define KMAX  3
#define BLINK 0
#define BEEP  1
#define DUR   2

#define TASK_BLOCKED 0xFFFF

#define LED_BIT_HI porta_=0b00000001
#define LED_BIT_LO porta_=0b00000000
#define SPEAKER_BIT_HI portb_=0b00000010
#define SPEAKER_BIT_LO portb_=0b00000000

// allocate KMAX kTimers
uInt kTimer[KMAX];
byte waitForTimer;
```

```
// Update kTimers
void interrupt(void)
{
  byte k;
  // clear the IF
  intcon_.TMR0IF = 0;
  for (k=0;k<KMAX;k++)
  {
    // hold at 0
    if((kTimer[k]) &&  // stop at zero
       (kTimer[k]!=TASK_BLOCKED))
    {
      kTimer[k]--;
    }
  }
  waitForTimer = FALSE;
}
```

```
void taskBlink(void)
{
  static byte seq=0;

  switch (seq)
  {
    state 0:
      LED_BIT_HI;
      seq=1;
      break;
    state 1:
      LED_BIT_LO;
      seq=0;
    }
    // set when to run again
    kTimer[BLINK]=1953;
    return;
}
```

```
void taskBeep(void)
{
  static byte seq=0;
  switch (seq)
  {
    state 0: // state INIT
      kTimer[DUR]=750;
      seq=1;
      break;
    state 1: // state ON
      SPEAKER_BIT_HI;
      kTimer[BEEP]=2;  // 1000 Hz
      seq=2;
      break;
    state 2: // state OFF
      SPEAKER_BIT_LO;;
      if(kTimer[DUR])  // another pulse
      {
        kTimer[BEEP]=2;
        seq=1;
      }
      else  // generate a rest
      {
        kTimer[BEEP]=750;
        seq=0;
      }
    }
}
```

```
void main (void)
{
  byte i;

  // setup
  {
    // speed up the clock to 8MHz, 18F1320
    osccon_.IRCF0=1;
    osccon_.IRCF1=1;
    osccon_.IRCF2=1;

    // configure ports
    adcon1_ = 0xFF;       // all digital
    trisb_ = 0xFD;        // speaker on RB1
    lata_ = 0;
    trisa_ = 0xBE;     // RA0 and RA6 ouputs 0b1011 1110;

    // configure Timer0
    t0con_ = 0xD0;
    intcon_.TMR0IF = 0;   // clear the IF
    intcon_.TMR0IE = 1;   // enable TMR0 overflow interrupt
    t0con__.TMR0ON = 1;   // turn on TMR0
    intcon_.GIE = 1;      // enable global interrupts

    // zero out the counters
    for (i=0; i<KMAX;i++)
    {
      kTimer[i] = 0;
    }
  }
  while(1) // main loop
  {
    if (!kTimer[BLINK])
    {
      taskBlink();
    }
    if (!kTimer[BEEP])
    {
      taskBeep();
    }
    while(waitForTimer);
    waitForTimer = TRUE;
  }
}
------------- end file DJ_coop2.c ---------------------------
```

## *multiCompiler.h*

```
------------- end file multiCompiler.h ---------------------------
/*
 * Junebug Demo
 * Purpose: Demonstrate cooporative multitasking
 *
 * File: multiCompiler.h
 * Software: BoostC or Microchip C18 compilers
 * Hardware: Junebug (or other debugger + breadboarded circuit)
 *
 * These defiles allow the use of more then one compiler.
 * Register names ending with a single _ are byte referances.
 * Register names ending with a dual __ are bit referances.
 *
 * by Daniel Johnson
 * July 2008
 */
```

```
#ifdef _BOOSTC
  #define FOUND_COMPILER

  #define lata_     lata
  #define trisa_    trisa
  #define porta_    porta

  #define latb_     latb
  #define trisb_    trisb
  #define portb_    portb

  #define intcon_   intcon
  #define intcon2_  intcon2
  #define osccon_   osccon
  #define t0con_    t0con
  #define t0con__   t0con
  #define adcon1_   adcon1

  #include <system.h>
  #pragma CLOCK_FREQ 8000000
  #pragma DATA _CONFIG1H, _INTIO2_OSC_1H
  #pragma DATA _CONFIG2H, _WDT_OFF_2H
  #pragma DATA _CONFIG3H, _MCLRE_ON_3H
  #pragma DATA _CONFIG4L, _LVP_OFF_4L
#endif
```

# JuneBug – BoostC - C18 Tutorials : Cooperative Multitasking (1st Draft, 4rd Edit)

```
// MCC18
#ifdef __18CXX
  #define FOUND_COMPILER

  #define lata_     LATA
  #define trisa_    TRISA
  #define porta_    PORTA

  #define latb_     LATB
  #define trisb_    TRISB
  #define portb_    PORTB

  #define intcon_   INTCONbits
  #define intcon2_  INTCON2bits
  #define osccon_   OSCCONbits
  #define t0con__   T0CONbits
  #define t0con_    T0CON
  #define adcon1_   ADCON1
```

```
 #pragma       config OSC = INTIO2, WDT = OFF, LVP = OFF
  #include <p18f1320.h>

  // code to make C18 interrupt look like BoostC's
  void interrupt(void);

  #pragma code low_vector=0x18
  void low_interrupt (void)
  {
    _asm GOTO interrupt _endasm
  }
  #pragma code
  #pragma interruptlow interrupt

#endif // __18CXX

// in unknown compiler generate error
#ifndef FOUND_COMPILER
   error: unknown compiler
#endif
------------- end file multiCompiler.h ---------------------------
```

## Apendix B: Hardware

*To be added.*

*Schematic for speaker and LED.*

*Image of connecting speaker to junebug.*

*Edit History*

*2nd Edit: 7/15/2008 Walk Through With Krumlink*

*3rd Edit: 7/16/2008 Wording and spelling DJ*

*4d Edit: 7/17/2008 Added section on decomposing tasks DJ*