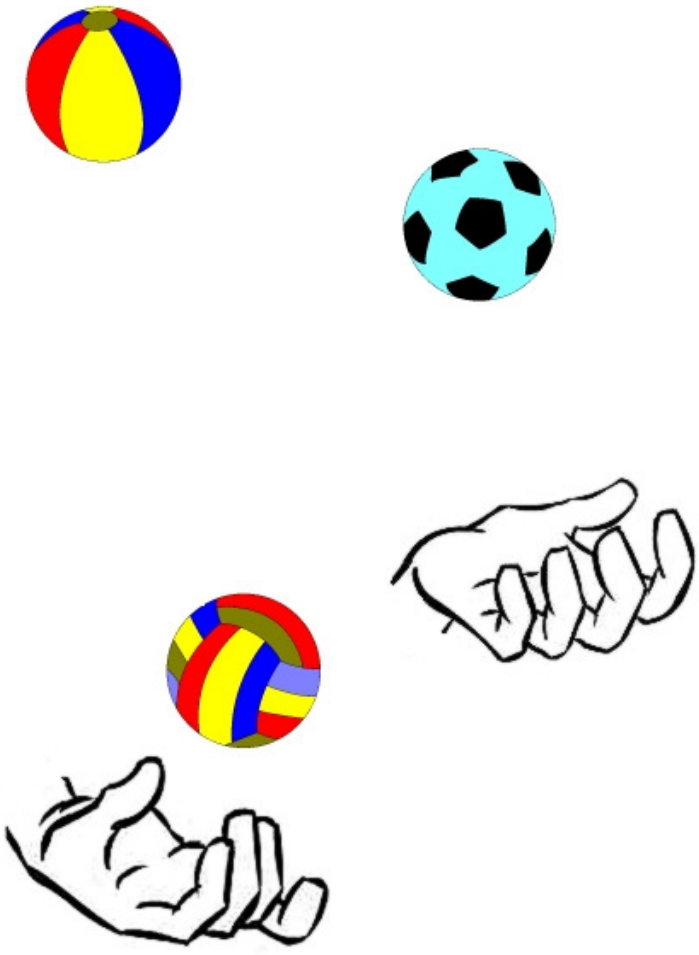
JUGGLER



A Realtime Pico Kernel

V 1.0 – Oct 2010

1 *Juggler* overview

For microcontrollers, such as the Microchip PIC families, a micro kernel can help solving concurrency problems. Think of cases where specifics tasks need to asynchronously wait for events to happen, like I/O, timed actions, etc. In some cases this can be implemented through Interrupt Service Routines, but this may cause prioritization and concurrency problems. In other words, having multiple independently running tasks can considerably ease the programming.

Microcontrollers can not rely on large amounts of resources, so the overhead required by the kernel must be minimal. *Juggler* is designed to provide only the most essential services, while using just a minimal amount of resources.

Services *Juggler* provides:

- concurrency, multi tasking, multi-threading
- timers
- inter-task synchronisation, event flags, semaphores

Concurrency is also referred to as muti-tasking or multi-threading. *Juggler* allows multipe functions (tasks, threads) to run in parallel, in time-sharing mode, according to a well defined set of rules.

Timers are essential for real-time applications, and hence considered a must-have for any RTOS.

Synchronisation is needed to let the different task signal to each other what they are doing. This RTOS does not have protected memory per task, so protection of shared resources must be done explicitly in the user tasks.

Juggler does not provide message passing, since this usually takes up too many resources. Also, hardware drivers are specific to the controller, and hence not part of *Juggler*. However, these services can easily be added, based on the services that *Juggler* does provide.

1.1 Tasks:

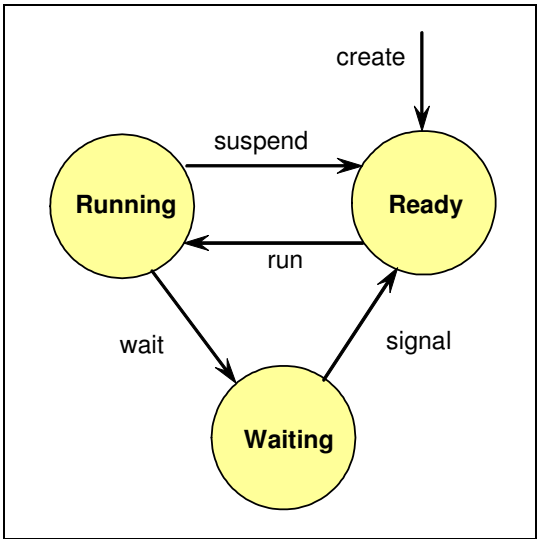
Like any kernel, *Juggler* enables concurrent execution of different tasks. The *Juggler* approach is to execute a thread of (subroutine), using its private context. This context includes own stack, own program counter and own copy of CPU registers. The subroutine usually is a non terminating loop, but it may equally well have an end. The start and the end of the subroutine execution is enclosed in the task context, which initialisation during task creation and terminated after returning from the subroutine.

Resuming, what is minimally required to enable multitasking:

- A Task Control Block per task, containing its' context
- A Task Skeleton, that handles the entry and exit of a task and allocates the task specific stack
- A Context Switcher, a subroutine that unloads one task from the CPU and loads another.

The set of states a task can be in:

- **Idle** task is created but waiting to be started
- **Ready** task is ready to use the CPU
- **Running** task is currently using the CPU
- **Waiting** task is waiting for an event to be signaled



Task State Diagram

A task that has just been created will be put in **Idle** state. When started, it is moved to **Ready** state, after which the scheduler determines whether it will get the CPU or not. Only when a task is **Running**, it may at some point wait on a event flag (semaphore), and possibly get into the **Waiting** state. A **Running** task may be suspended to **Ready** state when the scheduling algorithm decides this. This can for example happen when a higher priority task goes from **Waiting** to **Ready** state.

1.1.1 Task Control Block

The task control block stores the following information:

- saved stack pointer
- top of stack
- task state
- task event bits
- timeout clock value
- timeout interval

The total number of TCBs is configurable but fixed. Like all other allocation for a *Juggler* configuration, number of tasks and the associated task ids' are defined in the header file `juggler.h`.

1.1.2 Skeleton

This is a piece of code that initialises the TCB and creates an initial stackframe. This stackframe defines the initial task context, which is loaded when the task is scheduled in.

1.1.3 Context Switch

A context switch saves the context of the running process in its TCB and on the stack, and restores the context of the next task to run. This subroutine is to a large extent progammed in assembly, because of the switch between stacks and the hard jump to the restored PC address. The time it takes to switch context is determined by the amount of context

(registers) that has to be saved/restored. What an dhow exactly this has to be done depends on the CPUitself and on register usage by the C-compiler. It is mainly this part which has to be adapted when porting to a different processor.

1.1.4 Scheduler

The scheduler is a function that determines which task will run next. It is called in every system call, for example when waiting on or signalling an event, and on every timer tick.

There are several policies and algorithms possible, but for *Juggler* a simple strict priority based strategy is chosen. This means that each task runs on a different priority level, determined by its position in the TCB table. This position is determined by the task id, defined in `juggler.h`. Juggler goes through the TCB table from high to low, and selects the first ready task to run. Ultimately this may be the idle task (prio=0) which is always ready to run.

The general idea is that only a few tasks will be ready at any time. Most tasks will be waiting for some event to be signalled. Signalling of the event usually is done from ISRs or from timeouts. This makes the system built on *Juggler* fully event driven and very well suited for typical microcontroller applications.

In case a task takes long before blocking, it will be pre-empted by a system call or a clock-tick. This means that at least every clock-tick there will be a reschedule evaluation. Tasks that take a long time should have lower priority, so they can be interrupted by higher priority tasks.

1.2 Timers

An essential service in any realtime kernel is the timer. In a bigger OS, expiration of a timer can interrupt the flow of a task, but in *Juggler* it will merely signal an event. Since events are local to a task, only one timer can be started, by and for the task itself. For this purpose the TCB contains the timeout value and (optionally) a reload counter.

1.3 Events

Events are the simplest way of synchronisation between tasks. Signalling an event sets a bit and makes a task ready to run. Events are private to a task, so waiting for an event is local, but the event can be signalled by another task or an ISR.

1.4 Devices

Since these are rather hardware specific, device drivers are not part of *Juggler*. However, they could be created in a generic fashion, using the services that *Juggler* offers. To make devices abstract, a common set of access routines is shared and made available through a so-called device switch table (DST). A device number represents the offset of the associated entry in the table, and the generic access function can be accessed through a hook in the structure table element.

The following subroutines should be provided for each device driver:

- read read data from the device
- write write data to the device
- ioctl change or retrieve device settings
- isr the interrupt service routine (deferred actually, or upper half)

Devices usually are pre-allocated and are associated with physical peripherals (such as a serial port interface).

The *Juggler* installation contains implementation of the 30F4011 USART and an EEPROM devices.

2 Juggler API

2.1 General

```
void  _init(void);
```

Initializes the Juggler configuration, clock, taask control blocks, etc.

```
void  _start(void);
```

Starts the kernel, should be the last call in main(), which causes main() to become the idle thread with lowest priority. Before the kernel is started, the tasks should have been created.

```
void  _resched(void);
```

Causes the kernel to re-evaluate the list of tasks. The highest prio ready task will get the CPU. This function is normally only called in (other) system calls.

```
void  _tick(void);
```

Is called by the kernel clock every time an interrupt happens. This is the unit of time used for example for timers. The function checks whether there are any timers to time out. It also calls a `_resched()`.

2.2 Task control

```
void  _yield(void)
```

This function can be called from any task to free the CPU for another higher prio task, should there be one ready. Normally not needed since we have the tick() function which does the same every clocktick.

```
void  _mktask(int pid, ushort *stack, ushort size, void (*task)(void *), void *par)
```

This function creates the task skeleton, for the task indicated by the `pid`. It will use the `stack` that hes been pre-allocated and create the context for the task. Optionally, the `task` can be passed a pointer sized startup parameter `par`.

```
_getpid()
```

Obtains the calling tasks' own task id. Although not ogten needed, the retrieved PID could be used in communication with other tasks, and to make code more change-proof.

2.3 Timers

Each task has one timer which can be set to an integral number of clock ticks. An interval setting contains a reload timeout which is repeated until the timer is stopped. A timer is great for task synchronisation, when tasks need to run at certain intervals. No polling is needed, since the task will do nothing until the timeout event is flagged. The timer can be started concurrently with other events. Typically a loop will wait for an event, which includes timeout or any other cause, like for example the availability of input.

```
void   timer_start(ulong ticks, ulong period);
```

Starts the tasks' timer to a timeout of ticks. Waiting for the predefined EVENT_TIMER event blocks the task until the timeout happens. Optionally, the timer is reloaded with the period value. If period is 0, the timer acts in a one-shot mode.

```
void   timer_stop(void);
```

To cancel the tasks' timer.

2.4 Events

```
ushort event_wait(ushort mask);
```

The task that calls this function blocks until one of the events indicated in the mask is signalled. There can be multiple events ORed in the mask, and the return value will set a bit for every event that is set. It is up to the task to check all possible bits.

```
void   event_signal(int pid, ushort mask);
```

This function is called by a task or ISR in order to set an event flag for another task. This task will be made ready to run, and in fact it will when priority is higher than that of the signalling task. The signalled task is indicated by `pid`, and the events to raise are bits set in the `mask` parameter.

```
void   event_clear(ushort mask);
```

This function has to be called explicitly after dealing with an event, to reset the bitflag. Multiple bitflags may be reset at the same time, they should be set in the `mask` parameter.

3 Example application

4 Juggler 30F4011 sources

The Juggler environment consists of only two files: `juggler.c` and `juggler.h`. These need to be included in the project, and the header file must be adatpted to match application specifics, such as PIDs, number of tasks etc. The files contain many comments and guidelines for use, a little view on the inside is provided next.

4.1 juggler.c

```
/*============================================================================
JUGGLER is a very low overhead kernel for embedded applications, created
specifically for PIC micro controllers. A task control block takes 8 words,
and a typical stackframe would be 64 words. The OS itself uses 8 more words
for its own internal management.
The context switcher saves only WREG 0-15 and the SR, but this can be extended
or decreased when needed. This will influence the minimum stack frame size.

JUGGLER provides only the most elementary services:
 - scheduling,
 - timing,
 - event-based task synchronisation.
Tasks all have different priorities, determined by their PID.
The main() function ends as the idle task, with lowest priority 0.
This prevents the need for complicated round-robbin or fifo schemes.

Each task can have one timer and up to 8 different events. Timers can be
started as single shot or repetitive. User-defined events can be flagged
by other user tasks or ISRs.
A task can be in the following states:
 - running,
 - ready, waiting for higher priority tasks to finish
 - waiting, for an event to happen (either OS or user defined)

Rescheduling may occur in any system call, or on purpose by calling yield()
or in any case at every system clock tick (typically 1msec). Reschedule can
be prevented by locking the tasks' context. Also interrupts can be disabled,
but this should only be done for very short periods.

The context that is switched in a reschedule is fairly basic: just W0..W13
and SP/FP/SR. Upon application need, this set can be easily extended in the
functions _resched() and _mktask().

To be done:
- Add a device switch table, to abstract peripheral access.
- Enable nested interrupts, i.e. change ctx/int locking
- Check PSV usage (see compiler warnings)
============================================================================*/
#include <juggler.h>

tcb_t tcb_list[_NR_TASKS];            /* Allocate task control blocks       */

int _context_;                        /* >0 when context locked             */
```

```
int volatile _interrupt_;             /* >0 when interrupts locked          */

ushort _cur_pid;                      /* Holds running task PID             */
ushort _tmp_pid;                      /* Temporary PID for _resched()       */
tcb_t  *_cur_tcb_p;                   /* Points to running task TCB         */

ushort volatile _clock;               /* Clock counter, 65536 ticks         */
ulong  volatile _epoch;               /* Seconds since midnight 1-1-1970    */
ushort volatile _timint;              /* True if timer interrupt active     */

#define EPOCH_100101   1262304000     /* Epoch at UTC 00:00 1 jan 2010      */


/*****************************************************************************/
/*------------------------- Juggler Kernel ----------------------------------*/
/*****************************************************************************/

/*****************************************************************************/
/* _init()                                                                   */
/* Initializes kernel variables, allocates task control blocks.              */
/* Should be called in main() before task and driver initialisation.         */
/*****************************************************************************/

void _init(void)
{
  int ntask = _NR_TASKS;

  _interrupt_ = 0;                    /* Initialize interrupts lock       */
  _context_   = 0;                    /* Initialize context lock          */

  _lockctx();                         /* Lock context                     */
  _lockint();                         /* Lock interrupts                  */
  _clock      = TICKSPERSECOND;       /* Initialize clock counter         */
  _timint     = 0;                    /* Reset timer interrupt overrun    */

  while (ntask!=0)                    /* Initialize task control blocks   */
  {
    ntask--;
    _cur_tcb_p = &tcb_list[ntask];
    _cur_tcb_p->state    = TASK_IDLE;
    _cur_tcb_p->event    = 0;
    _cur_tcb_p->timeout  = 0;
    _cur_tcb_p->interval = 0;
  }
}


/*****************************************************************************/
/* _start()                                                                  */
/* Starts the scheduler and enters an endless loop.                          */
/* Should be the last call from main()                                       */
/*****************************************************************************/
void _start(void)
```

```
{
  INTCON1bits.NSTDIS = 0;             /* Enable nested interrupts           */
  asm("bclr CORCON, #0x0003": : );    /* CPU interrupt level < 7            */


  /* System clock settings */
  asm("clr T1CON": : );               /* Clear Timer control register       */
  asm("clr TMR1": : );                /* Clear TMR1 register                */
  asm("mov #0x2710, W0": : );         /* Load period, from juggler.h        */
  asm("mov W0, PR1": : );
  asm("bclr IPC0, #0x000C": : );      /* Set T1 interrupt priority level    */
  asm("bset IPC0, #0x000D": : );      /*   to 6                             */
  asm("bset IPC0, #0x000E": : );      /*                                    */
  asm("bclr IFS0, #0x0003": : );      /* Clear T1 flag                      */
  asm("bset IEC0, #0x0003": : );      /* Enable T1 interrupt                */
  asm("mov #0xA000, W0": : );         /* Synch mode, int clock, no presc.   */
  asm("mov W0, T1CON": : );

  _cur_pid = PID_IDLE;                /* Now running: idle task             */
  _cur_tcb_p = &tcb_list[PID_IDLE];
  _cur_tcb_p->state = TASK_READY;

  _freeint();
  _freectx();
  _resched();                         /* Jump to highest prio ready task    */

idle_loop:                            /* Enter endless loop (idle task)     */
  asm("nop" : : );
  goto idle_loop;
}

/*************************************************************************/
/* _mktask()                                                             */
/* Create a stack as part of the task creation                           */
/* Set up the stackframe as if the task context has just been saved      */
/*  tcb_p points to TCB to be initialised                                */
/*  stack points to the allocated stack area                             */
/*  task points to the entry address of the task function                */
/*  par points to a pointer type parameter of the task                   */
/* The current stackpointer is saved in the TCB                          */
/*************************************************************************/
void _mktask(int pid, ushort *stack, ushort size, void (*task)(void *), void *par)
{
  tcb_t  *tcb_p;
  ushort *sptr;
  ushort *fptr;

  tcb_p = &tcb_list[pid];

  /*
   * W14 points to current frame: i.e. start of local vars
   * Before function call:
   *   parameters are stacked on current frame, last one first
```

```
   *   return address is stacked (24 bits, i.e. 2 shorts)
   * In called function:
   *   LNK #x instruction,
   *      current FP is stacked
   *      FP <- SP (new frame)
   *      SP += #x (allocate space for locals)
   * Construct stackframe as it would be in _resched:
   *   delimiter (old FP points here)
   *   task parameter
   *   return address [0..15]
   *   return address [16..23]
   *   old FP
   *   _reched local var (temp) (new FP points here)
   */
  sptr = stack;
  *sptr++ = 0x5aa5;                     /* 0x00, delimiter                  */
  *sptr++ = (ushort) (par);             /* 0x02, task parameter             */
  *sptr++ = (ushort)task;               /* 0x04, task entry point [ 0..15]  */
  *sptr++ = 0x0000;                     /* 0x06, task entry point [16..23]  */
  *sptr++ = (ushort) stack;             /* 0x08, old FP                     */
  fptr = sptr;                          /* save new FP for later            */
  *sptr++ = (ushort)pid;                /* 0x0A, _resched temp (=PID)       */

  /*
   * Now add the context save as done by _resched()
   */
  *sptr++ = 0x0000;                     /* 0x0C, WREG0                      */
  *sptr++ = 0x1111;                     /* 0x0E, WREG1                      */
  *sptr++ = 0x2222;                     /* 0x10, WREG2                      */
  *sptr++ = 0x3333;                     /* 0x12, WREG3                      */
  *sptr++ = 0x4444;                     /* 0x14, WREG4                      */
  *sptr++ = 0x5555;                     /* 0x16, WREG5                      */
  *sptr++ = 0x6666;                     /* 0x18, WREG6                      */
  *sptr++ = 0x7777;                     /* 0x1A, WREG7                      */
  *sptr++ = 0x8888;                     /* 0x1C, WREG8                      */
  *sptr++ = 0x9999;                     /* 0x1E, WREG8                      */
  *sptr++ = 0xaaaa;                     /* 0x20, WREG10                     */
  *sptr++ = 0xbbbb;                     /* 0x22, WREG11                     */
  *sptr++ = 0xcccc;                     /* 0x24, WREG12                     */
  *sptr++ = 0xdddd;                     /* 0x26, WREG13                     */
  *sptr++ = SR;                         /* 0x28, SR (take current value)    */
  *sptr++ = (ushort)fptr;                   /* 0x2A, _resched based FP          */

  tcb_p->sptr  = (ushort)sptr;          /* Save stack pointer in TCB        */
  tcb_p->stop = (ushort)(stack + size);
  tcb_p->state = TASK_READY;
}


/*****************************************************************************/
/* _resched()                                                                */
/* Check tcb_list for new task to run                                        */
/* Assumes that the task state has already been changed.                     */
```

```
/* If so: Save context, Load new task, Restore context                      */
/* The calling task state has been changed by the calling function!         */
/* TODO: handle saved interrupt status                                      */
/**************************************************************************/
void _resched( void )
{
  ushort temp;

  if (_context_ != CTX_FREE)            /* Is reschedule allowed?           */
    return;                             /*   no, so exit                    */

  if (_cur_tcb_p->state == TASK_RUN)    /* If currently running             */
    _cur_tcb_p->state = TASK_READY;     /*   make current task ready        */

  temp = _NR_TASKS;                     /* Go down the prio list to find    */
  while (temp != 0)                     /*   the highest prio ready task    */
  {
    temp--;
    if (tcb_list[temp].state == TASK_READY)
      break;
  }

  if (_cur_pid == temp)                 /* If it is us: just exit           */
  {
    _cur_tcb_p->state = TASK_RUN;       /* Set current task back to running */
    return;
  }

  /* NOTE: SHOULD SAVE INT PRIO LEVEL FOR LATER RESTORE !!! */
  asm("disi #3" : : );                  /* Disable interrupts:              */
  asm("bset  SR, #5" : : );             /*   and set intprio level to 7     */
  asm("bset  SR, #6" : : );
  asm("bset  SR, #7" : : );

  asm("mov.w w0,[w15++]" : : );         /* Save work registers on stack     */
  asm("mov.w w1,[w15++]" : : );
  asm("mov.w w2,[w15++]" : : );
  asm("mov.w w3,[w15++]" : : );
  asm("mov.w w4,[w15++]" : : );
  asm("mov.w w5,[w15++]" : : );
  asm("mov.w w6,[w15++]" : : );
  asm("mov.w w7,[w15++]" : : );
  asm("mov.w w8,[w15++]" : : );
  asm("mov.w w9,[w15++]" : : );
  asm("mov.w w10,[w15++]" : : );
  asm("mov.w w11,[w15++]" : : );
  asm("mov.w w12,[w15++]" : : );
  asm("mov.w w13,[w15++]" : : );

  asm("mov.w SR,w0" : : );              /* Save Status register on stack    */
  asm("mov.w w0,[w15++]" : : );

  asm("mov.w w14,[w15++]" : : );        /* Save frame pointer               */
```

```
  asm("mov.w __cur_tcb_p,w1" : : );     /* Save stack pointer               */
  asm("mov.w w15,[w1]" : : );           /*   of current task in TCB         */

  _cur_pid   = temp;                    /* Switch current task to the new   */
  _cur_tcb_p = &tcb_list[_cur_pid];     /*   task that was selected.        */
  _cur_tcb_p->state = TASK_RUN;


  asm("mov.w __cur_tcb_p,w1" : : );     /* Restore stack pointer for        */
  asm("mov.w [w1],w15" : : );           /*   new task from TCB              */
  asm("nop" : : );


  SPLIM  = (uint)(_cur_tcb_p->stop - 8);/* Set stack limit, reserve 8 words */
  asm("nop" : : );                      /*   for trap handling              */

  asm("mov.w [--w15],w14" : : );        /* Restore frame pointer            */

  asm ("mov.w [--w15],w0" : : );        /* Restore Status register          */
  asm ("mov.w w0,SR" : : );             /*   from stack                     */

  asm ("mov.w [--w15],w13" : : );
  asm ("mov.w [--w15],w12" : : );
  asm ("mov.w [--w15],w11" : : );
  asm ("mov.w [--w15],w10" : : );
  asm ("mov.w [--w15],w9" : : );
  asm ("mov.w [--w15],w8" : : );
  asm ("mov.w [--w15],w7" : : );
  asm ("mov.w [--w15],w6" : : );
  asm ("mov.w [--w15],w5" : : );
  asm ("mov.w [--w15],w4" : : );
  asm ("mov.w [--w15],w3" : : );
  asm ("mov.w [--w15],w2" : : );
  asm ("mov.w [--w15],w1" : : );
  asm ("mov.w [--w15],w0" : : );

  asm volatile ("disi #0x3");           /* Enable interrupts                */
  asm ("bclr  SR, #5" : : );            /*   and set intprio level to 0     */
  asm ("bclr  SR, #6" : : );
  asm ("bclr  SR, #7" : : );
}


/*************************************************************************/
/* _yield()                                                              */
/* Call scheduler to yield for potential higher prio ready task          */
/*************************************************************************/
void _yield( void)
{
  if (_context_)                        /* Cannot yield when context locked */
    return;
  _resched();
```

```
}


void _panic(void)
{
  while(1)
    asm("nop" : : );
}


/*************************************************************************/
/* _tick()                                                               */
/* Increment system clock tick counter, check rollover                   */
/* Decrement timers where appropriate, change task state if timeout      */
/*************************************************************************/
void __attribute__((__interrupt__)) _T1Interrupt(void)
{
  unsigned char tid;

  IFS0bits.T1IF = 0;                    /* Reset interrupt flag            */

  if (_timint > 0)                      /* Overrun: panic!                 */
    _panic();
  _timint++;                            /* Increment overrun counter       */

  _clock--;                             /* Decrement clock counter         */
  if (_clock == 0)                      /* Second full?                    */
  {
    _clock = TICKSPERSECOND;            /* Reset counter                   */
    _epoch++;                           /* Increment epoch                 */
  }

  for (tid=1; tid<_NR_TASKS; tid++)     /* Check all tasks                 */
  {
    if (tcb_list[tid].timeout == 0)     /* No timer set?                   */
      continue;                         /*   then skip                     */

    tcb_list[tid].timeout--;            /* Decrement                       */
    if (tcb_list[tid].timeout == 0)     /* End reached?                    */
    {                                   /*   then set event                */
      tcb_list[tid].timeout = tcb_list[tid].interval;
      tcb_list[tid].event  |= EVENT_TIMER;
      tcb_list[tid].state   = TASK_READY;
    }
  }

  _timint--;                            /* Decrement overrun counter       */
  _resched();
}



/*************************************************************************/
```

```
/* --------------------------- Juggler Services ---------------------------*/
/*****************************************************************************/

/*****************************************************************************/
/* Event handling                                                            */
/* A task can block for an event to happen. The event is user definable and */
/* is identified by a mask. This mask has a bit for each event, where the   */
/* lower eight bits are user defined. The event can be raised by any other  */
/* task. After wait or signal event a reschedule will be done.              */
/* OS and user defined events: see juggler.h                                */
/* event_wait()    Returns in case there already is an event raised that    */
/*                 matches the mask, or waits for one and blocks.           */
/* event_clear()   Clears the event according to the mask, typically after  */
/*                 handling it.                                             */
/* event_signal()  Sets the event for a task indicted by pid and according  */
/*                 to the mask, and makes the task ready to run.            */
/*****************************************************************************/
ushort event_wait(ushort mask)
{
  _lockint();                           /* Make re-entrant: lock interrupts */
  while (!(_cur_tcb_p->event & mask))   /* Test if one of the events is set */
  {
    _cur_tcb_p->state = TASK_WAIT;      /* None set: move to waiting state  */
    _freeint();
    _resched();
    _lockint();
  }
  _freeint();                           /* Release interrupts               */
  return(_cur_tcb_p->event);            /* Return event flags to task       */
}

void event_clear(ushort mask)
{
  _cur_tcb_p->event &= ~mask;           /* Reset event flags                */
}

void event_signal(int pid, ushort mask)
{
  _lockint();                           /* Make re-entrant: lock interrupts */
  tcb_list[pid].event |= mask;          /* Set event flags                  */
  tcb_list[pid].state  = TASK_READY;    /* Task is now ready to run         */
  _freeint();
  _resched();
}


/*****************************************************************************/
/* Timer handling                                                            */
/* One timer is defined per task, which is intended for sleep or repetitive */
/* action. If more time controlled action is required, the application      */
/* should be designed to have a separate task per timer and sunchronise     */
/* tasks with events.                                                        */
/* The timer will raise the EVENT_TIMER event.                               */
```

```
/* timer_start()   Sets timout values, initial and interval. If the timer   */
/*                 was already running, the new values prevail. Any pending */
/*                 timer event is cleared.                                   */
/* timer_stop()    Clears all timer values, and resets pending EVENT_ALARM. */
/*****************************************************************************/
void timer_start(ulong iticks, ulong rticks)
{
  _lockint();
  _cur_tcb_p->timeout  = iticks;        /* Initial timeout value            */
  _cur_tcb_p->interval = rticks;        /* Repetitive timeout value         */
  _cur_tcb_p->event   &= ~EVENT_TIMER;  /* Reset alarm event                */
  _freeint();
}

void timer_stop(void)
{
  _lockint();
  _cur_tcb_p->timeout  = 0;             /* Initial timeout value            */
  _cur_tcb_p->interval = 0;             /* Repetitive timeout value         */
  _cur_tcb_p->event   &= ~EVENT_TIMER;  /* Reset alarm event                */
  _freeint();
}
```

4.2 juggler.h

```
#ifndef _JUGGLER_H
#define _JUGGLER_H
#include <p30fxxxx.h>

/************************ User defined parameters ***************************/
/* Define task ids, including the default idle task (PID=0)                 */
/* Define number of tasks, this sizes the TCB table                         */
/* Define user events, mask is unsigned short, lower byte is user defined.  */
/****************************************************************************/
#define _NR_TASKS       5
#define PID_IDLE        0               /* Reserved for _start() loop       */
#define PID_MON         1
#define PID_LX2         2
#define PID_XFM         3
#define PID_SPC         4

#define EVENT_UPDATE    0x0001
#define EVENT_EERDY     0x0002
#define EVENT_USER2     0x0004
#define EVENT_USER3     0x0008
#define EVENT_USER4     0x0010
#define EVENT_USER5     0x0020
#define EVENT_USER6     0x0040
#define EVENT_USER7     0x0080



/********************** Kernel Settings below ******************************/
/************* If you have to change these, do it with care! ****************/
/****************************************************************************/

/************************** Clock settings **********************************/
/* Calculate as follows:                                                    */
/* 10MHz: 10000 instruction cycles per msec                                 */
/* No prescaler: PR load for 1msec ticks is 10000(0x2710)                   */
/****************************************************************************/
#define TICKSPERSECOND  1000            /* 1 msec ticks                     */
/* #define _TMR1_PERIOD    0x2710          10MHz instruction clock, 1msec   */
#define _TMR1_PERIOD    0x26f5          /* 1 sidereal msec: 9973            */

/************************ Kernel & driver events ****************************/
/* Kernel events, upper byte range                                          */
/****************************************************************************/
#define EVENT_TIMER     0x8000          /* Reserved for timer event         */
#define EVENT_UART      0x4000          /* Reserved for uart event          */

/************************** Kernel definitions ******************************/
/* Kernel internal definitions, used in macros and drivers                  */
/* Do not change any of these!                                              */
/****************************************************************************/
typedef unsigned short ushort;
```

```
typedef unsigned char  uchar;
typedef unsigned long  ulong;
typedef unsigned int   uint;

#define TASK_STATE      0x0003          /* State bits mask                  */
#define TASK_IDLE       0x0000          /* Should not happen                */
#define TASK_RUN        0x0001          /* Running task                     */
#define TASK_WAIT       0x0002          /* Waiting for an event             */
#define TASK_READY      0x0003          /* Ready to run                     */

typedef struct tcb tcb_t;
struct tcb                              /* Task Control Block               */
{
  ushort volatile sptr;                 /* Saved stack pointer              */
  ushort volatile stop;                 /* Top of stack                     */
  ushort volatile state;                /* Stores task state                */
  ushort volatile event;                /* Event bits                       */
  ulong  volatile timeout;              /* Clock value to timeout           */
  ulong  volatile interval;             /* Add to clock for next timeout    */
};
extern tcb_t tcb_list[_NR_TASKS];       /* Allocate task control blocks     */

extern int _context_;                   /* >0 when context locked           */
extern int volatile _interrupt_;        /* >0 when interrupts locked        */
extern ushort _cur_pid;                 /* Holds running task PID           */
extern tcb_t  *_cur_tcb_p;              /* Points to running task TCB       */

extern ulong volatile    _epoch;
#define EPOCH      (_epoch)


#define NULL (void *)0

/*****************************************************************************/
/* Juggler Kernel API                                                        */
/*****************************************************************************/
#define CTX_FREE         0
#define _lockctx() _context_++
#define _freectx() _context_--

#define _lockint()                                                          \
{                                                                           \
  if (_interrupt_ == 0)                                                     \
  {                                                                         \
    asm("disi #3" : : );                                                    \
    asm("bset  SR, #5" : : );                                               \
    asm("bset  SR, #6" : : );                                               \
    asm("bset  SR, #7" : : );                                               \
  }                                                                         \
  _interrupt_++;                                                            \
}

#define _freeint()                                                          \
```

```
{                                                                               \
  if (_interrupt_ == 1)                                                         \
  {                                                                             \
    _interrupt_ = 0;                                                            \
    asm volatile ("disi #3");                                                   \
    asm ("bclr  SR, #5" : : );                                                  \
    asm ("bclr  SR, #6" : : );                                                  \
    asm ("bclr  SR, #7" : : );                                                  \
  }                                                                             \
  else                                                                          \
    _interrupt_--;                                                              \
}

#define _getpid()         (_cur_pid)

/*************************************************************************/
/* Juggler System Calls                                                  */
/*************************************************************************/
ushort event_wait(ushort mask);
void   event_signal(int pid, ushort mask);
void   event_clear(ushort mask);

void   timer_start(ulong ticks, ulong period);
void   timer_stop(void);

void  _init(void);
void  _start(void);
void  _yield(void);
void  _resched(void);
void  _tick(void);
void  _mktask(int pid, ushort *stack, ushort size, void (*task)(void *), void *par);

/*************************************************************************/
/* Some useful macros                                                    */
/*************************************************************************/
#define MAX(a,b)   ((a)>(b)?(a):(b))
#define MIN(a,b)   ((a)>(b)?(b):(a))
#define SGN(a)            ((a)<0?-1:1)
#define ABS(a)            ((a)<0?(-a):(a))


#endif /*_JUGGLER_H */
```