# OPERATING-SYSTEM-INDEPENDENT MODULAR PROGRAMMING METHOD FOR ROBUST JUST-IN-TIME RESPONSE TO MULTIPLE ASYNCHRONOUS DATA STREAMS


## ABSTRACT

An operating-system-independent modular programming method is disclosed, which includes providing one or more tasks, one or more task queues, and zero or more condition queues. Each task is a program that is run in sequence. Each task queue includes a task queue program and a queue containing zero or more tasks. Each condition queue includes a condition queue program and a queue containing zero or more tasks and associated conditions. Each task includes task ending code that refers to zero, one, or more than one successor task, and the task queue program or the condition queue program handles each such successor task by either running it or placing it in a task queue or a condition queue. The programming method further includes providing a fan and an end fan construct to enable a parent task to generate a plurality of child sequences. These program constructs may be used to form pseudothreads, each pseudothread being a single sequence of control, with interruptions allowed but no concurrency of execution, implemented in an operating system independent way. Also disclosed is an application of this programming method to accomplish robust just-in-time response to multiple asynchronous data streams in a RAID control program. The RAID control program includes a plurality of requesters, a resource allocator, and a plurality of stripe parity and IO masters, and an optional chainer, each being implemented as a pseudothread.

# APPENDIX A

```
/*
 * occam channels and alternations among independent pseudo-threads!
 * Add timers. LJD 4/23/01
 * Add output-and-end-process, used only if communications partner
 * is an ALT, because it requires the input come last. LJD 7/9/01
 */
#include <memory.h>
#include "rncommon.h"
/*{{{  instructions on outer format*/

/*
 * Initialization required for workspaces:
 * code       Points to allocated struct code with following two
 *            values initialized:
 * code->run  Initialized to following code block
 * code->data Initialized to following code block (same as prev)
 *            Enclosing code must set code->exitcond correctly:
 *            see chan_input, chan_output, chan_altwt.
 * tranadr    Valid address for transmission
 * altstate   Initialized to 0 unless internal to alt sequence
 *            in which case the alt calls take care of it after
 *            initializing to 0 before chan_alt
 * timerstate Initialized to 0 (no timer alt running) but can
 *            be set nonzero by chan_talt.
 * timeout    timeout: when time becomes >= timeout, reschedule
 *            or make alt ready.
 *
 * ------------------------------------------------------------
 * Ordering of a talt (channel and timer case):
 * void talttopRun(void *data, code_t *thiscode) {
 *   ....
 *   mywsp->code->data = data;
 *   mywsp->code->run = altresultRun;
 *   chan_talt(mywsp);
 *   chan_enbt(mywsp,timeout);
 *   chan_enbc(mywsp,channel1);
 *   thiscond->head = NULL;  no queue if not timer
 *   if (chan_taltwt(mywsp)==ALT_READY) {
 *     thiscond->test = NULL;
 *     thiscode->exitcond = thiscond;
 *   } else if (mywsp->timerstate) {
 *     thiscond->data = (void *)mywsp;
 *     thiscond->head = find_timer_head();
 *     thiscode->exitcond = thiscond;
 *     rncco_enter_atomic();
 *     thiscond->test = timerTest;
 *     attach_timer_condition(thiscond);
 *     rncco_exit_atomic();
 *   } else {
 *     thiscode->exitcond = NULL;
 *   }
 * }
 * void taltresultRun(void *data, code_t *thiscode) {
 *   int branch = 0;
 *   ....
 *   detach_condition(thiscond); remove if in timer queue
 *   if (chan_dist(mywsp,timeout)) branch = 2;
 *   if (chan_disc(mywsp,channel1)) branch = 1;
 *   chan_altend(mywsp);
 *   if (branch==1) {
 *     input from channel1;
 *     execute branch 1;
 *   } else if (branch==2) {
 *     execute branch 2;
 *   } else {
 *     CROAK;
 *   }
 *   ....
```

```
* }
*
* ------------------------------------------------------------
* Ordering of an alt (2 channel case):
* void alttopRun(void *data, code_t *thiscode) {
*   ....
*   mywsp->code->data = data;
*   mywsp->code->run = altresultRun;
*   chan_alt(mywsp);
*   chan_enbc(mywsp,channel2);
*   chan_enbc(mywsp,channel1);
*   if (chan_altwt(mywsp)==ALT_READY) {
*     thiscond->test = NULL;
*     thiscode->exitcond = thiscond;
*   } else {
*     thiscode->exitcond = NULL;
*   }
* }
* void altresultRun(void *data, code_t *thiscode) {
*   int branch = 0;
*   ....
*   if (chan_disc(mywsp,channel2)) branch = 2;
*   if (chan_disc(mywsp,channel1)) branch = 1;
*   chan_altend(mywsp);
*   if (branch==1) {
*     input from channel1;
*     execute branch 1;
*   } else if (branch==2) {
*     input from channel2;
*     execute branch 2;
*   } else {
*     CROAK;
*   }
*   ....
* }
*
* ------------------------------------------------------------
* Setting unconditional timer wait code->exitcond correctly,
* with mywsp->code=nextcode, thiscode->parent=NULL,
* thiscode->fan=NULL:
*
* thiscond->code = nextcode;
* thiscond->data = (void *)mywsp;
* thiscond->head = find_timer_head();
* thiscode->exitcond = thiscond;
* rncco_enter_atomic();
* if (chan_tin(mywsp,timeout)) {
*   thiscond->test = NULL;
* } else {
*   thiscond->test = timerTest;
*   attach_timer_condition(thiscond);
* }
* rncco_exit_atomic();
* <return from thiscode->run>
* ------------------------------------------------------------
* Setting code->exitcond correctly, with mywsp->code=nextcode,
* thiscode->parent=NULL, thiscode->fan=NULL:
*
* if (chan_input(mywsp,channel,length)) {
*   thiscond->test = NULL;
*   thiscode->exitcond = thiscond;
* } else {
*   thiscode->exitcond = NULL;
* }
* <return from thiscode->run>
*
* if (chan_output(mywsp,channel,length)&1) {
*   thiscond->test = NULL;
```

```
 *   thiscode->exitcond = thiscond;
 * } else {
 *   thiscode->exitcond = NULL;
 * }
 * <return from thiscode->run>
 *
 * if (chan_altwt(mywsp)==ALT_READY) {
 *   thiscond->test = NULL;
 *   thiscode->exitcond = thiscond;
 * } else {
 *   thiscode->exitcond = NULL;
 * }
 * <return from thiscode->run>
 */
/*}}}  */
/*{{{  chan_tin(workspace_t *,timecycle_t)*/

/*
 * It returns 1 if the timeout is expired, 0 if not. In either case
 * it loads up the timeout in mywsp. The caller should load
 * cond->test = timerTest and deschedule himself if 0 is
 * returned, and fall through to next task if 1 is returned.
 */
int chan_tin(workspace_t *mywsp, timecycle_t timeout) {
  mywsp->altstate = ALT_NOALT;    /* UNCONDITIONAL */
  mywsp->timerstate = 0;          /* UNCONDITIONAL */
  mywsp->timeout = timeout;
  return rncco_expired(timeout);
}
/*}}}  */
/*{{{  chan_input(workspace_t *,workspace_t **,int)*/

/*
 * It returns 1 if the transmission took place, 0 if other side not
 * there. If other side is there, it calls attachTask after
 * transmission. The caller should deschedule himself if 0 is
 * returned, and fall through to next task if 1 is returned.
 *
 * For the opposing output, it sets mywsp->altstate = ALT_NOALT
 * which should be the default anyway except between chan_alt and
 * chan_altend.
 *
 * IMPORTANT: if incoming->code is NULL, this is output-and-end-process.
 * In this case, attachTask is not called. It is assumed that
 * the outputting process has descheduled itself unconditionally.
 * This should be used only if this input is part of an ALT. LJD 7/9/01
 */
int chan_input(workspace_t *mywsp, workspace_t **chanadr, int length) {
  workspace_t *incoming = *chanadr;
  if (incoming) {
    memcpy(mywsp->tranadr,incoming->tranadr,length);
    if (incoming->code) attachTask(incoming->code);
    *chanadr = NULL;
    return 1;
  }
  mywsp->altstate = ALT_NOALT;
  *chanadr = mywsp;
  return 0;
}
/*}}}  */
/*{{{  chan_output(workspace_t *,workspace_t **,int)*/

/*
 * It returns 1 if the transmission took place, 0 if other side not
 * there, 2 if an alt was underway on the other side. If other side is
 * there (and, if alt, waiting), it calls attachTask after
 * transmission. The caller should deschedule himself if 0 or 2 is
 * returned, and fall through to next task if 1 is returned.
```

```
 *
 * IMPORTANT: if mywsp->code is NULL, this is output-and-end-process.
 * In this case, the caller should deschedule himself unconditionally.
 * This should be used only if the other side is an ALT. LJD 7/9/01
 */
int chan_output(workspace_t *mywsp, workspace_t **chanadr, int length) {
  workspace_t *incoming = *chanadr;
  if (incoming) {
    int altstate = incoming->altstate;
    if (altstate) {
      incoming->altstate = ALT_READY;
      if (altstate==ALT_WAITING) {
        attachTask(incoming->code);
      }
      *chanadr = mywsp;
      return 2; /* basically equivalent to 0 from this side */
    }
    memcpy(incoming->tranadr,mywsp->tranadr,length);
    attachTask(incoming->code);
    *chanadr = NULL;
    return 1;
  }
  *chanadr = mywsp;
  return 0;
}
/*}}}  */
/*{{{  chan_talt(workspace_t *)*/

/*
 * It starts an alt by setting mywsp->altstate = ALT_ENABLING.
 * It makes sure alt time marked not set.
 */
int chan_talt(workspace_t *mywsp) {
  mywsp->altstate = ALT_ENABLING;
  mywsp->timerstate = 0;
  return ALT_ENABLING;
}
/*}}}  */
/*{{{  chan_alt(workspace_t *)*/

/*
 * It starts an alt by setting mywsp->altstate = ALT_ENABLING.
 */
int chan_alt(workspace_t *mywsp) {
  mywsp->altstate = ALT_ENABLING;
  return ALT_ENABLING;
}
/*}}}  */
/*{{{  chan_enbt(workspace_t *,timecycle_t)*/

/*
 * It enables a timer alternative. It does NOT set ready: taltwt will
 * do that. This merely checks the timeout is the lowest.
 */
int chan_enbt(workspace_t *mywsp, timecycle_t timeout) {
  if (mywsp->timerstate) {
    if ((mywsp->timeout)>timeout) mywsp->timeout = timeout;
  } else {
    mywsp->timerstate = 1;
    mywsp->timeout = timeout;
  }
  return 0;
}
/*}}}  */
/*{{{  chan_enbc(workspace_t *,workspace_t **)*/

/*
 * It enables an alternative channel (the boolean should be in the
```

```
 * calling code). If nothing there, it places its descriptor in it,
 * and returns 0. If something already there, it writes ready, and
 * returns ALT_READY. It assumes ALT_ENABLING or ALT_READY when
 * called, without checking.
 *
 * Does not check if incoming descriptor = mywsp (same channel
 * ALTed twice).
 */
int chan_enbc(workspace_t *mywsp, workspace_t **chanadr) {
  workspace_t *incoming = *chanadr;
  if (incoming) {
    mywsp->altstate = ALT_READY;
    return ALT_READY;
  }
  *chanadr = mywsp;
  return 0;
}
/*}}}  */
/*{{{  chan_enbs(workspace_t *)*/

/*
 * It enables an alternative skip (the boolean should be in the
 * calling code). Always returns ALT_READY.
 */
int chan_enbs(workspace_t *mywsp) {
  mywsp->altstate = ALT_READY;
  return ALT_READY;
}
/*}}}  */
/*{{{  chan_taltwt(workspace_t *)*/

/*
 * If not ready, it does the following two things:
 *   if timeout set, checks it;
 *     if timed out, sets ALT_READY and returns 3 (ALT_READY);
 *     if not timed out, sets ALT_WAITING and returns 2 (ALT_WAITING).
 *   if timeout not set:
 *     sets ALT_WAITING and returns 2 (ALT_WAITING).
 * If ready, it returns ALT_READY (3).
 * If ALT_WAITING, the process should place itself on the timer queue
 * if timeout set, and then deschedule. Otherwise it should fall
 * through (to a short task that completes the ALT and goes on to
 * branch among selections).
 */
int chan_taltwt(workspace_t *mywsp) {
  if ((mywsp->altstate)==ALT_READY) return ALT_READY;
  if (mywsp->timerstate) {
    if (rncco_expired(mywsp->timeout)) {
      mywsp->altstate = ALT_READY;
      return ALT_READY;
    }
  }
  mywsp->altstate = ALT_WAITING;
  return ALT_WAITING;
}
/*}}}  */
/*{{{  chan_altwt(workspace_t *)*/

/*
 * If not ready, it sets ALT_WAITING and returns 2 (ALT_WAITING).
 * If ready, it returns ALT_READY (3).
 * If ALT_WAITING, the process should deschedule, otherwise fall
 * through (to a short task that completes the ALT and goes on to
 * branch among selections).
 */
int chan_altwt(workspace_t *mywsp) {
  if ((mywsp->altstate)==ALT_READY) return ALT_READY;
  mywsp->altstate = ALT_WAITING;
```

```
  return ALT_WAITING;
}
/*}}}  */
/*{{{  chan_dist(workspace_t *,timecycle_t)*/

/*
 * Returns 1 if and only if timed out.
 * Calling code should unconditionally remove the queued timeout
 * condition (if it was never attached, there is no problem.)
 */
int chan_dist(workspace_t *mywsp,timecycle_t timeout) {
  return rncco_expired(timeout);
}
/*}}}  */
/*{{{  chan_disc(workspace_t *,workspace_t **)*/

/*
 * It disables an alternative channel (the boolean should be in the
 * calling code). If it was ready it returns 1 (selectable). Otherwise
 * it returns 0. Does not check whether it has already been selected;
 * this should be done by calling code.
 *
 * Assumes it was enabled (so *chanadr==NULL is not a possibility) and
 * that the other side is not mywsp.
 */
int chan_disc(workspace_t *mywsp, workspace_t **chanadr) {
  workspace_t *incoming = *chanadr;
  if (incoming==mywsp) {
    *chanadr = NULL;
    return 0;
  }
  return 1;
}
/*}}}  */
/*{{{  chan_diss(workspace_t *)*/

/*
 * It disables an alternative skip (the boolean should be in the
 * calling code). Always returns 1 for selectable.
 */
int chan_diss(workspace_t *mywsp) {
  return 1;
}
/*}}}  */
/*{{{  chan_altend(workspace_t *)*/

/*
 * It concludes the alting by returning the altstate to ALT_NOALT=0.
 * Assumes calling code is acting upon the branch information found.
 * Note that nothing in the alting code actually does a transmission
 * of data, only pinpoints it ready.
 */
int chan_altend(workspace_t *mywsp) {
  mywsp->altstate = ALT_NOALT;
  mywsp->timerstate = 0;
  return 0;
}
/*}}}  */
/*{{{  twochan_alttop(workspace_t *,int,int,w.. **,w.. **)*/
/*
 * It returns 0 if waiting, nonzero if ready.
 * channel1 has higher priority.
 * bool1 OR bool2 must be true.
 */
int twochan_alttop(workspace_t *mywsp, int bool1, int bool2,
  workspace_t **channel1, workspace_t **channel2) {
  chan_alt(mywsp);
  if (bool2) chan_enbc(mywsp,channel2);
```

```
  if (bool1) chan_enbc(mywsp,channel1);
  if (chan_altwt(mywsp)==ALT_READY) return 1;
  return 0;
}
/*}}}  */
/*{{{  twochan_altbottom(workspace_t *,int,int,w.. **,w.. **)*/
/*
 * It returns 0 if error, 1 if channel1 preferred, 2 if channel2.
 * channel1 has higher priority.
 * bool1 OR bool2 must be true.
 * All formals must be the same as in the twochan_alttop call with
 * no value changes between.
 */
int twochan_altbottom(workspace_t *mywsp, int bool1, int bool2,
  workspace_t **channel1, workspace_t **channel2) {
  int branch = 0;
  if (bool2) {
    if (chan_disc(mywsp,channel2)) branch = 2;
  }
  if (bool1) {
    if (chan_disc(mywsp,channel1)) branch = 1;
  }
  chan_altend(mywsp);
  return branch;
}
/*}}}  */
/*{{{  twochan_talttop(workspace_t *,int,int,int,w.. **,w.. **,time..)*/
/*
 * It returns 0 if waiting, nonzero if ready.
 * channel1 has higher priority, timeout has lowest.
 * bool1 OR bool2 OR bool3 must be true.
 * If ALT_WAITING and mywsp->timerstate both valid after calling
 * this, the calling code must attach condition to timer queue.
 */
int twochan_talttop(workspace_t *mywsp, int bool1, int bool2,
  int bool3, workspace_t **channel1, workspace_t **channel2,
  timecycle_t timeout) {
  chan_talt(mywsp);
  if (bool3) chan_enbt(mywsp,timeout);
  if (bool2) chan_enbc(mywsp,channel2);
  if (bool1) chan_enbc(mywsp,channel1);
  if (chan_taltwt(mywsp)==ALT_READY) return 1;
  return 0;
}
/*}}}  */
/*{{{  twochan_taltbottom(workspace_t *,int,int,int,w.. **,w.. **,time..)*/
/*
 * It returns 0 if error, 1 if channel1 preferred, 2 if channel2.
 * channel1 has higher priority.
 * bool1 OR bool2 OR bool3 must be true.
 * All formals must be the same as in the twochan_alttop call with
 * no value changes between. The timeout condition must have been
 * removed (unconditionally) from the timer queue before calling
 * this.
 */
int twochan_taltbottom(workspace_t *mywsp, int bool1, int bool2,
  int bool3, workspace_t **channel1, workspace_t **channel2,
  timecycle_t timeout) {
  int branch = 0;
  if (bool3) {
    if (chan_dist(mywsp,timeout)) branch = 3;
  }
  if (bool2) {
    if (chan_disc(mywsp,channel2)) branch = 2;
  }
  if (bool1) {
    if (chan_disc(mywsp,channel1)) branch = 1;
  }
```

```
  chan_altend(mywsp);
  return branch;
}
/*}}}  */
```

```
/*
 * INTERRUPT SPACE CODE FOR RAID
 * Larry Dickson
 * Land-5 Corporation 2/27/2001-3/1/2001, 3/19/2001, 3/20/2001
 * For the time being I will embed the structure definitions in this
 * file. LJD 2/27/2001
 * Add timecycle_t, and add timeout to condition_t. LJD 4/20/2001
 * Took the structure definitions out of the file. LJD 4/24/2002
 */
#include "rncommon.h"
/*{{{  attach_condition*/

/*
 * ATTACH THE CONDITION
 * Assumes thiscond->head is already valid head of the queue to
 * which this will be attached.
 * Should be called from within the immediate code block, after
 * thiscond is set up with test, data, code, head, AND WHILE
 * THE CONDITION IS STILL FALSE. If condition goes true before
 * the attach takes place, then don't attach, just set
 * thiscond->code to NULL! Otherwise you may wait forever.
 */
void attach_condition(condition_t *thiscond) {
  condition_t **thisheadp = thiscond->head; /* must be real */
  condition_t *thishead = *thisheadp;
  if (thishead) {
    /* this is being attached to end */
    condition_t *end = thishead->prev;
    end->next = thiscond;
    thishead->prev = thiscond;
    thiscond->next = thishead;
    thiscond->prev = end;
  } else {
    /* this is being attached to an empty chain */
    *thisheadp = thiscond;
    thiscond->prev = thiscond;
    thiscond->next = thiscond;
  }
}
/*}}}  */
/*{{{  runtests*/

/*
 * CHECK IN AN ISR AND QUEUE CODE.
 * All code is ASSUMED atomic. This includes SMP effects.
 * The formal points to the condition queue (initially NULL).
 * This routine traverses the entire condition queue - the
 * tests better be fast and the queue rather short.
 *
 * WARNING - If the underlying disks share a request queue,
 * then a separate head will need to be shared too, and all
 * room-on-queue requests chained to it. Then every ISR cleanup
 * will need to call runtests (ordered==FALSE) for its (done check)
 * queue AND call runtests (ordered==TRUE) for the shared request
 * queue.
 *
 * Bug fix - LJD 4/16/01 - null connections of thiscond in right place.
 * Bug fix - LJD 4/16/01 - must test for cond==tail.
 */
void runtests(condition_t **head, int ordered) {
  condition_t *thisnew = *head;
  if (thisnew) {
    condition_t *thistail = thisnew->prev;
    condition_t *thiscond;
    do {
      /* thisnew is guaranteed to be nonNULL here */
      thiscond = thisnew;
      thisnew = thiscond->next;
```

```
      if (thiscond->test(thiscond->data)) {
        /* place thiscond->code on immediate queue */
        attachTask(thiscond->code);
        thiscond->head = NULL; /* makes it illegal ! */
        if (thisnew==thiscond) {
          /* detach thiscond from the queue */
          thiscond->next = thiscond->prev = NULL;
          /* it's the only thing left: empty the queue */
          *head = NULL;
          thiscond = NULL;
          break; /* should not be needed */
        } else if (thisnew) {
          /* the queue had at least 2 members, doesn't empty */
          condition_t *last = thiscond->prev;
          /* detach thiscond from the queue */
          thiscond->next = thiscond->prev = NULL;
          last->next = thisnew;
          thisnew->prev = last;
          if ((*head)==thiscond) *head = thisnew;
        } else {
          break; /* should never happen */
        }
      } else if (ordered) {
        break; /* if ordered, first failure ends run */
      }
    } while (thiscond!=thistail);
  }
}
/*}}}  */
/*{{{  attach_timer_condition*/

/*
 * ATTACH THE TIMER BASED CONDITION
 * This is a specialist version which attaches the condition, not
 * at the end (necessarily), but after the last condition whose
 * (absolute) timeout is <= the one found here, in circular ordering.
 * Assumes thiscond->head is already valid head of the queue to
 * which this will be attached.
 * Should be called from within the immediate code block, after
 * thiscond is set up with test, data, code, head, timeout AND WHILE
 * THE CONDITION IS STILL FALSE. If condition goes true before
 * the attach takes place, then don't attach, just set
 * thiscond->code to NULL! Otherwise you may wait forever.
 * Should be called with interrupts disabled otherwise timer may
 * trash the chain in the middle of the calculation.
 * Normally called ordered, with test=timerTest,
 * data=(void *)wsp, where thiscond->timeout==wsp->timeout,
 * and (LJD 5/3/01) workspace_t *wsp contains alt information
 * as appropriate.
 */
void attach_timer_condition(condition_t *thiscond) {
  condition_t **thisheadp = thiscond->head; /* must be real */
  condition_t *top = *thisheadp;
  timecycle_t time = thiscond->timeout;
  if (top) {
    condition_t *end;
    if ((top->timeout)>time) {
      /* attach to front in front of something */
      end = top->prev;
      end->next = thiscond;
      top->prev = thiscond;
      thiscond->next = top;
      thiscond->prev = end;
      *thisheadp = thiscond;
    } else {
      /* this is not being attached to front */
      condition_t *thishead = top;
      do {
```

```
        if ((thishead->timeout)>time) break;
        thishead = thishead->next;
      } while (thishead!=top);
      end = thishead->prev;
      end->next = thiscond;
      thishead->prev = thiscond;
      thiscond->next = thishead;
      thiscond->prev = end;
    }
  } else {
    /* this is being attached to an empty chain */
    *thisheadp = thiscond;
    thiscond->prev = thiscond;
    thiscond->next = thiscond;
  }
}
/*}}}  */
/*{{{  detach_condition*/

/*
 * DETACH CONDITION (FROM THE MIDDLE?)
 * This removes a condition from the middle (or beginning or end)
 * of a condition queue. It MUST be called atomically! It is
 * normally used to remove something (like a failed alt branch)
 * from the timer queue.
 *
 * If the condition isn't there it does nothing. Assumes without
 * testing that the condition is nonNULL and really attached to
 * the queue in question.
 */
void detach_condition(condition_t *thiscond) {
  condition_t **thisheadp = thiscond->head;
  if (thisheadp) {
    condition_t *top = *thisheadp;
    condition_t *thisnew = thiscond->next;
    thiscond->head = NULL; /* makes it illegal ! */
    if (top) {
      if (top==thiscond) {
        if (thisnew==thiscond) {
          /* detach thiscond from the queue */
          thiscond->next = thiscond->prev = NULL;
          /* it's the only thing left: empty the queue */
          *thisheadp = NULL;
        } else if (thisnew) {
          /* the queue had at least 2 members, doesn't empty */
          condition_t *last = thiscond->prev;
          /* detach thiscond from the queue */
          thiscond->next = thiscond->prev = NULL;
          last->next = thisnew;
          thisnew->prev = last;
          *thisheadp = thisnew;
        } else {
          return; /* should never happen */
        }
      } else { /* top != thiscond: not the start condition */
        /* the queue had at least 2 members, doesn't empty */
        condition_t *last = thiscond->prev;
        /* detach thiscond from the queue */
        thiscond->next = thiscond->prev = NULL;
        last->next = thisnew;
        thisnew->prev = last;
      }
    }
  }
}
/*}}}  */
```

```
#ifndef RNCOMMON_H
#define RNCOMMON_H

#ifndef NULL
#define NULL (void *)0
#endif

/* Internal clock time - must be bigger than ever needed
 */
typedef long long timecycle_t;
#define RN_TIMECYCLE 1

/* Bitmap used in RAID striped command sets to underlying disks
 */
#ifndef RN_DOWNMASK
#define RN_DOWNMASK 1
typedef unsigned long downmask_t;
#endif

/* Structures needed for pseudothread sequencing
 */

/*
 * THE code STRUCTURE
 * normally called by
 *   code_t *thiscode;
 *   thiscode->run(thiscode->data,thiscode);
 * thiscode->run points to a block of code using real allocated and
 * structured data at thiscode->data;
 * thiscode->run uses its second formal to set AT MOST ONE of
 * thiscode->exitcond, thiscode->parent, and thiscode->fan according
 * as the end is a condition test, an END PAR or a PAR respectively;
 * if none are set the end is unconditional (a real termination).
 */
struct condition;
struct par;
typedef struct code {
  struct code *next; /* no sync flag needed LJD 3/20/01 */
  void (*run) (void *, struct code *);
  void *data;
  struct condition *exitcond;
  struct par *parent;
  struct par *fan;
} code_t;

/*
 * THE par STRUCTURE
 * This supports the following parallel branch, illustrated for
 * nbranches=3:
 *
 *           |
 *       PAR *---------*---------*---------\
 *           |         |         |         |
 *                  cbrnch[0] cbrnch[1] cbrnch[2]
 *           .         .         .
 *           .         .         .
 *    END PAR *        -         -         -
 *           |
 *       *endcode
 *           .
 *           .
 *
 * A PAR exit by the parent sets nleft equal to nbranches and loops
 * over the nbranches branch entries, which must be an already
 * initialized code_t array. An END PAR exit at the end of a child
 * quasi-thread decrements nleft and abandons its quasi-thread and,
 * if nleft goes to 0, unconditionally branches to *endcode, which
 * must be an already initialized code_t structure. If the
```

```
 * connections bit corresponding to a branch is not set, this is a
 * null branch (no code is run, only nleft is decremented).
 */
typedef struct par {
  int nbranches;
  int nleft;
  downmask_t connections;
  code_t *cbrnch;
  code_t *endcode;
} par_t;

/*
 * THE condition STRUCTURE
 * normally called by
 * condition_t *thiscond;
 * if (thiscond->test(thiscond->data)) {
 *   <code that may disconnect thiscond->prev and thiscond->next>
 *   <code that branches to or queues thiscond->code>
 * } else {
 *   <code that may connect thiscond to an ISR>
 * }
 *
 * This test takes place once in the overlying code execution. If
 * it fails there, it is attached to a disk ISR and will be retried
 * there till it succeeds.
 *
 * This structure is built under the assumption that a condition is
 * satisfied by a single disk's state. If several disks' states must
 * contribute, a PAR and END PAR are required with the conditions at
 * the end. Then the endcode of the par structure can assume all the
 * conditions are satisfied.
 *
 * Typically the data pointer points to a simple subset of the real
 * data that can be tested quickly. Typically there is a test map that
 * stays unchanged while awaiting completion, and a state map pointer
 * to state which is updated by asynchronous code, which also calls
 * the condition code.
 *
 * If thiscond->next==NULL, then this condition is being applied in
 * system  or immediate code, otherwise it is being applied in the
 * asynchronous code. If (*thiscond->head)==thiscond, then this
 * condition is at the head of its condition queue, and thiscond->prev
 * is the end of its condition queue. The next and prev
 * pointers are always circular.
 *
 * If thiscond->test==NULL, then the condition is always true, and
 * the result should be an unconditional branch to the following code.
 */
typedef struct condition {
  int (*test) (void *);
  void *data;
  code_t *code;
  struct condition **head;
  struct condition *next;
  struct condition *prev;
  timecycle_t timeout;
} condition_t;

/* workspace and alternation state values
 */
#define ALT_NOALT    0
#define ALT_ENABLING 1
#define ALT_WAITING  2
#define ALT_READY    3
typedef struct workspace {
  code_t *code;   /* run=next code, data=local data */
  char *tranadr;  /* address for channel transmissions */
  int altstate;   /* state of alternation */
```

```
  int timerstate; /* state of time selection (TALT) */
  timecycle_t timeout; /* restart time for process */
} workspace_t;

/* Trick to distinguish low priority from high - based on fact that
 * all tasks' data structures are at even addresses.
 */
#define CHAN_VOIDOR1(voidadr) ((void *)(((long)(voidadr))|((long)1)))

/* This EXTERNAL FUNCTION queues a task to the appropriate queue, as
 * determined by the actual data in *code. In this implementation,
 * there are only two queues, determined by ((int)(code->data))&1
 * which if zero means high priority and if nonzero means low priority.
 * Further elaboration is possible for multiprocessing.
 */
int attachTask(code_t *code);

/* These EXTERNAL FUNCTIONS dequeue a task from a task queue
 * (high priority for "detachTask" and low for "detachlpTask"),
 * returning the task address or NULL if the queue was empty.
 * Further elaboration would be needed for multiprocessing.
 */
code_t *detachTask(void);
code_t *detachlpTask(void);

/* This EXTERNAL FUNCTION sets system task status: 0 if no task is
 * running, 1 if a high priority task is running, 2 if a low priority
 * task is running. It is assumed in this implementation that tasks do
 * not run concurrently or interrupt each other. Multiprocessing or
 * interrupting high priority would require further elaboration.
 */
void settaskStatus(int value);

/*
 * atomic data access
 * These EXTERNAL FUNCTIONS enter and exit an atomic code block, which
 * must run with interrupts cleared and multiprocessing disabled. Thus,
 * the calling code must always assure that after rncco_enter_atomic
 * is called, every branch leads to an rncco_exit_atomic call.
 **/
int rncco_enter_atomic(void);
int rncco_exit_atomic(void);

/*
 * time function
 * This EXTERNAL FUNCTION must return 1 if timeout is less than or
 * equal to the current time, and 0 if timeout is greater than (after)
 * the current time. It is assumed that the time resolution and starting
 * point is such that the current time will never "fold over" from
 * mostpositive to mostnegative, given the resolution of a timecycle_t
 * and the practical limits on program duration.
 **/
int rncco_expired(timecycle_t timeout);

#endif /* #ifndef RNCOMMON_H */
```

Coded realization of modular programming method

DESCRIPTION

This is a descriptive document, intended to accompany the code files

rncommon.h
tests.h
tests.c
intspace.c
channel.c
taskcode.c

which include known working components of the existing RAID core under development by Inostor Corporation. The files are slightly reorganized from those found in the RAID core, in order to make easier the teaching of the principles behind the code.

Every part of this invention in the code diagrammed in the eight pages of RAID core drawings sent to Ying Chen on 16 January 2003 can be (and has been) constructed using the components in these files. The only apparent exception, "User Process Wakeup", is actually prior art accomplished by a standard process wakeup call such as is normally used to signal completion of a driver's task.

Revision (January 20, 2003): I discovered an unnecessary operating system dependency in a structure and removed it, making the code set fully OS independent as in the invention title.

EXTERNAL FUNCTIONS REQUIRED

All the external functions, except rncco_enter_atomic, must be capable of running atomically (during disabling of interrupts and multiprocessing). And rncco_enter_atomic must exit in atomic state, while rncco_exit_atomic must exit in non-atomic state if the previous call to rncco_enter_atomic was from non-atomic state.

memcpy channel.c
This is the standard block copy between non-intersecting blocks.

attachTask channel.c, intspace.c, taskcode.c
This attaches a task to its proper immediate queue.

detachTask taskcode.c
detachIpTask taskcode.c
These detach a task from the front of an immediate queue, high priority or low priority respectively.

rncco_enter_atomic taskcode.c
rncco_exit_atomic taskcode.c
These enter and exit an atomic code block. They include the equivalent of CLI and STI respectively, but they must also disable multiprocessing if relevant for the duration of the block.

rncco_expired channel.c, tests.c
This returns TRUE if a timeout has expired.

rncco_request_commit tests.c
This returns TRUE if the underlying disk's request queue has enough room for a set of IO commands from a RAID. Whenever it returns TRUE, it must atomically reserve this space for the requester.

settaskStatus taskcode.c
This informs the system which level task if any is running: 0 no task, 1 high priority task, 2 low priority task (no multiprocessing or task interrupting task is handled in this implementation).

CAPABILITIES OF THE EXEMPLARY IMPLEMENTATION CODE

(A) tests.c

These are tests which are called from within condition queues or elsewhere. They must be run WITHIN an atomic block and may be run within hard interrupt code if necessary.

timerTest
This tests for timeouts.

requestTest
This tests an underlying disk's commit queue space availability, and commits the space if successful.

availableTest
This manages a binary semaphore queue.

map1Test
map2Test
map3Test
These test for completion of all of a set of IO operations on an underlying disk.

(B) intspace.c

These are the innermost nonblocking queue-handling capabilities, central to the implementation of the invention. They must be run WITHIN an atomic block and may be run within hard interrupt code if necessary.

attach_condition
This attaches a new condition (which must be tested and found false in an atomic branch including this call) to the end of a condition queue.

runtests
This runs a condition queue, attaching tasks waiting on successful conditions to their task queues. If run in ordered mode, it quits after reaching a failure. If run in non-ordered mode, it tests the whole queue. It is expressly intended to run within interrupt code.

attach_timer_condition
This attaches a task to a timer queue, keeping it ordered, and hence may attach it to the middle of the queue.

detach_condition
This removes a task unconditionally from anywhere in the queue. It is used in timer alternations, whether the timer branch works or fails.

(C) channel.c

This is the communication code related to the occam/CSP communication constructs. They must be run WITHIN an atomic block, but are not run within hard interrupt code in this implementation.
Of the contents of this file, the calls themselves are instances of well-known CSP/occam constructs. Their implementation in terms of task queues and condition queues is new art. See the extensive comments in this file for guidance on applying this art.

chan_tin
This is the unconditional timer wait.

chan_input
chan_output
These are the input and output sides of an unconditional channel data

transmission (synchronized and unbuffered). The output may be communicating with an alternation (select) on the other side, in which case it may be an output-and-end-process.

chan_talt
chan_alt
chan_enbt
chan_enbc
chan_enbs
chan_taltwt
chan_altwt
chan_dist
chan_disc
chan_diss
chan_altend
These are the components of an alternation, or select, construct. They do not transmit data, but determine the "winner" of a race to see who wakes up the connection first. The contestants may include one or more timeouts (chan_enbt and chan_dist), one or more input channels (chan_enbc and chan_disc), and/or one or more skips (chan_enbs and chan_diss). The order is chan_alt (or chan_talt), one or more chan_enb?, chan_altwt (or chan_taltwt), the same one or more chan_dis?, and chan_altend. The talt calls are used if and only if there is a timeout. The order of the chan_dis? calls determines priority (last is highest).

twochan_alttop
twochan_altbottom
twochan_talttop
twochan_taltbottom
These provide valid working examples of the above rules. The first two calls accomplish an alternation between two channels. The last two calls accomplish an alternation between two channels and a timeout.

(D) taskcode.c

These are the task queue handlers. They must be called by NON-ATOMIC immediate queue code in this implementation, and may not be called within hard interrupt code, because they in turn call any of the pseudothread tasks, which may allow interruption and do not interrupt each other.

runcode
Called by the task cycling procedures to follow, it runs a task and places its successor(s), if any, in the appropriate queue, which may not be the same queue as that of the predecessor. Successors may be determined by one of the following: the condition queue or fall-through, the PAR construct, or the end PAR construct. In the case of the PAR construct, runcode calls itself recursively, but following the rule that the workspaces of PAR children must be non-overlapping subsets of their parent's workspace, this process cannot go on indefinitely.

taskCycle
taskIpCycle
These exhaust an entire task queue; "taskCycle" runs the high priority queue, and "taskIpCycle" runs the low priority queue, and therefore also the high priority queue. The requirement is that the high priority queue must be empty before the low priority queue can run. Queues and the task status are manipulated atomically.

taskattachCycle
This procedure is the same as "taskIpCycle" except it starts by atomically attaching a new task.

```
#include "rncommon.h"
/*{{{  runcode*/

/*
 * RUN THE CODE
 * The void * retyping of topcode (which should be a code_t *) is
 * because task queueing prototypes (i.e. Linux) may require
 * void qmember(void *) prototype.
 */
void runcode(void *topcode) {
  code_t *thiscode = (code_t *)topcode;
  while (thiscode) {
    /* remove lo pri bit if necessary */
    thiscode->data = (void *)(((long)(thiscode->data))&(~((long)1)));
    (thiscode->run)(thiscode->data,thiscode);
    if (thiscode->exitcond) {
      /*{{{  handle condition test*/
      condition_t *thiscond = thiscode->exitcond;
      if (thiscond->test) {
        thiscode = NULL; /* condition already attached */
      } else {
        thiscode = thiscond->code; /* unconditional success */
      }
      /*}}}  */
    } else if (thiscode->fan) {
      par_t *thisfan = thiscode->fan;
      int n = thisfan->nbranches;
      downmask_t testmask = 1;
      downmask_t valid = thisfan->connections;
      code_t *br = thisfan->cbrnch;
      int ilast = -1;
      int i;
      /*{{{  handle PAR including nothing set up*/
      thisfan->nleft = n;
      for (i=0;i<n;i++) {
        if (valid&testmask) {
          if (ilast>=0) runcode(br+ilast);
          ilast = i;
        } else {
          (thisfan->nleft)--;
        }
        testmask <<= 1;
      }
      if (ilast<0) {
        thiscode = thisfan->endcode;
      } else {
        /* important: nleft>0 here, therefore no cli needed. */
        thiscode = br+ilast; /* last branch continues of parent */
      }
      /*}}}  */
    } else if (thiscode->parent) {
      /*{{{  handle END PAR*/
      par_t *parent = thiscode->parent;
      rncco_enter_atomic();
      (parent->nleft)--;
      if (parent->nleft) {
        thiscode = NULL;
      } else {
        thiscode = parent->endcode; /* cannot change */
      }
      rncco_exit_atomic();
      /*}}}  */
    } else {
      thiscode = NULL; /* unconditional end */
    }
  }
}
/*}}}  */
```

```
/*{{{  taskCycle()*/
void taskCycle(void) {
  register code_t *task;
  /* task status changes atomically to 1 then back to 0 */
  rncco_enter_atomic();
  task = detachTask();
  settaskStatus(1); /* notify it is high priority */
  while (task) {
    rncco_exit_atomic();
    runcode(task);
    rncco_enter_atomic();
    task = detachTask();
  }
  settaskStatus(0); /* no task queue running */
  rncco_exit_atomic();
}
/*}}}  */
/*{{{  tasklpCycle()*/
void tasklpCycle(void) {
  register code_t *task;
  /* task status changes atomically between 1 and 2 then back to 0 */
  rncco_enter_atomic();
  task = detachTask();
  if (task) {
    settaskStatus(1); /* notify it is high priority */
  } else {
    task = detachlpTask();
    settaskStatus(2); /* notify it is low priority */
  }
  while (task) {
    rncco_exit_atomic();
    runcode(task);
    rncco_enter_atomic();
    task = detachTask();
    if (task) {
      settaskStatus(1); /* notify it is high priority */
    } else {
      task = detachlpTask();
      settaskStatus(2); /* notify it is low priority */
    }
  }
  settaskStatus(0); /* no task queue running */
  rncco_exit_atomic();
}
/*}}}  */
/*{{{  taskattachCycle(code_t *)*/
void taskattachCycle(code_t *newtask) {
  register code_t *task;
  /* task status changes atomically between 1 and 2 then back to 0 */
  rncco_enter_atomic();
  attachTask(newtask);
  task = detachTask();
  if (task) {
    settaskStatus(1); /* notify it is high priority */
  } else {
    task = detachlpTask();
    settaskStatus(2); /* notify it is low priority */
  }
  while (task) {
    rncco_exit_atomic();
    runcode(task);
    rncco_enter_atomic();
    task = detachTask();
    if (task) {
      settaskStatus(1); /* notify it is high priority */
    } else {
      task = detachlpTask();
      settaskStatus(2); /* notify it is low priority */
```

```
    }
  }
  settaskStatus(0); /* no task queue running */
  rncco_exit_atomic();
}
/*}}}  */
```

```
/*
 * TEST CODE that will be called from within underlying disk ISRs
 * and elsewhere in RAID.
 * Larry Dickson
 * Land-5 Corporation
 * 27 February 2001 - 28 February 2001
 * This code assumes that for underlying devices, requests MAY be
 * shared. Therefore it provides device, tag, diskind for each incoming
 * condition->data and for the callout rncco_request_commit. A call to
 * this returns a reservation if successful (recoverable by any device,
 * tag, diskind call with the same values). The release of these
 * reserved spaces is accomplished by rncco_request_take.
 *
 * add stripe to requestQuery. 4 April 2001 LJD
 */

#include "rncommon.h"

#include "tests.h"

/*
 * In this case, cond->data = (void *)&wsp is required.
 * rncco_expired returns 1 iff timeout<=rncco_timecycle_current()
 * AND (LJD 5/3/01) it is not a ready alt. This is the only condition
 * queue test that is integrated with the occam/CSP communication
 * handling (alternation) in the current implementation.
 */
int timerTest(void *data) {
  workspace_t *thisdata = (workspace_t *)data;
  int altstate = thisdata->altstate;
  int ret = rncco_expired(thisdata->timeout);
  if (altstate) {
    if (ret) thisdata->altstate = ALT_READY;
    if (altstate!=ALT_WAITING) return 0;
  }
  return ret;
}

/*
 * All the values in the requestQuery structure must be filled:
 * device = device id
 * tag = tag of the RAID array
 * diskind = disk index within RAID[tag]
 * howmany = how many requests will be used on this device by the
 * RAID call to follow.
 *
 * The result is 0 if failed and >0 if space for that many requests is
 * now reserved for (device, tag, diskind). The space will be
 * released by ANY rncco_request_end_commit call for this underlying
 * device.
 * rncco_request_commit MUST store the returned value, rqshunt,
 * in the structure for later use in these calls. It is zero for failure
 * and nonzero global tag if successful.
 *
 * Used for disk commit condition queue.
 */
int requestTest(void *data) {
  return rncco_request_commit((requestQuery_t *)data);
}

/*
 * In this case, cond->data = (void *)&available is required, where
 * available is an int. This test returns old value and changes
 * value of int to 0, which means it succeeds at most once. Used for
 * binary semaphore.
 */
int availableTest(void *data) {
  int *avptr = (int *)data;
```

```
  int invalue = *avptr;
  *avptr = 0;
  return invalue;
}

/*
 * Three different done test routines are offered: map1Test, map2Test,
 * map3Test. It is assumed that the masks do not change over the life
 * of the condition, but the pointed-to states are changed by
 * asynchronous underlying disk code, and AND to zero with the masks
 * only when done. Used (map1Test) for disk done condition queue.
 */
int map1Test(void *mapData) {
  mapQuery_t *m = (mapQuery_t *)mapData;
  if ((m->mask1&(*(m->state1)))) return 0;
  return 1;
}
#ifndef IOCTL_ONLY
int map2Test(void *mapData) {
  mapQuery_t *m = (mapQuery_t *)mapData;
  if ((m->mask1&(*(m->state1)))||
      (m->mask2&(*(m->state2)))) return 0;
  return 1;
}
int map3Test(void *mapData) {
  mapQuery_t *m = (mapQuery_t *)mapData;
  if ((m->mask1&(*(m->state1)))||
      (m->mask2&(*(m->state2)))||
      (m->mask3&(*(m->state3)))) return 0;
  return 1;
}
#endif /* ndef IOCTL_ONLY */
```

```
#ifndef TESTS_H
#define TESTS_H

/* This is a job identifier used in commit queue handling.
 */
typedef unsigned long rnshunt_t;

/*
 * TEST CODE that will be called from within underlying disk ISRs
 * and elsewhere in RAID.
 * Larry Dickson
 * Land-5 Corporation
 * 27 February 2001 - 28 February 2001
 * This code assumes that for underlying devices, requests MAY be
 * shared. Therefore it provides device, tag, diskind for each incoming
 * condition->data and for the callout rncco_request_commit. A call to
 * this returns a reservation if successful (recoverable by any device,
 * tag, diskind call with the same values). The release of these
 * reserved spaces is accomplished by rncco_request_take.
 *
 * add stripe to requestQuery. 4 April 2001 LJD
 * give a special (unsigned long) typedef to rqshunt. 1 May 2002 LJD
 */

/* This structure uniquely identifies an underlying disk that is
 * driven by the RAID.
 */
typedef struct underlyingDevice {
  unsigned long device;
  int tag;
  int diskind;
} underlyingDevice_t;

/* This structure expresses how many underlying disk requests are
 * about to be shoved onto the queue of this underlying disk. Other
 * information may be found in the "struct stripeset" if applicable:
 * it deals with the fact that a RAID stripe may have more than one
 * disk block per disk.
 */
struct stripeset;
typedef struct requestQuery {
  underlyingDevice_t *thisdisk;
  struct stripeset *stripe;
  int howmany;
  rnshunt_t rqshunt;
} requestQuery_t;

/* This structure provides a bitmask showing which of the possible set
 * of disk blocks for this RAID stripe are in the set of underlying
 * disk IO commands, and allows each underlying disk command completion
 * to clear a state bit when done or failed. Up to three sets of
 * states are allowed. Disk done is signalled when all pertinent states
 * are fully cleared.
 */
typedef struct mapQuery {
  underlyingDevice_t *thisdisk;
  downmask_t mask1;
  downmask_t *state1;
  downmask_t mask2;
  downmask_t *state2;
  downmask_t mask3;
  downmask_t *state3;
} mapQuery_t;

/* Commit
 * This EXTERNAL FUNCTION must return 1 if the underlying disk can
 * accept the full set of requests implied by "req", and 0 otherwise.
 * If it returns 1, it must reserve space in its queue for these
```

```
 * requests, and hold it until told to release it. Thus, a nonzero
 * return value must always result in request processing and release,
 * or else the underlying queue will be leaked.
 */
int rncco_request_commit(struct requestQuery *req);

#endif /* #ifndef TESTS_H */
```