# EMBS Assignment

Jake Coxon

November 29, 2012

## 0.1 Development

todo: how did i come to these design decisions The point of the source node is to transmit a packet during the reception phase of one of the three sink motes. For a given mote, any time a beacon is received with a payload $n$, there is $nt$ time until the beginning of the reception phase and then $12t$ time until the beginning of the next sync phase. This can then be repeated indefinitely.

The first part of the program in this case is calculating $t$, which can be done by finding the difference in time between two separate beacons from a mote ($t_1$ and $t_2$). The beacons are not necessarily consecutive, so

$$t = \frac{t_2 - t_1}{n_1 - n_2}$$

$$\text{reception}_{\text{next}} = t_2 + n_2 t$$

$$\text{phase}_{\text{next}} = \text{reception}_{\text{next}} + 11t$$

If however $n_1$ is 1 then this is the last beacon of the synchronisation phase and so it is impossible to find $t$ during this cycle. Fortunately it is known that the next sync phase is $12t$ time where $t \geq 500$ so the program should wait $12 \times 500ms$ and try again.

The next part of the problem comes when trying to calculating these values for three motes in parallel since all motes are on different channels.

The program tracks a state for all motes; initially 'waiting-to-sync'. The program will arbitrarily start with $mote0$ and wait for two beacons. Following this, the state is set to 'successfully-synced', the reception phase is calculated and a timer is set to fire at the reception phase.

If the program is waiting to sync and does not receive a beacon in $1500ms$ it should switch to a new mote which has state 'waiting-to-sync'. This is because the program does not know if the mote is in synchronisation phase and therefore could be waiting for a beacon for up to $11 \times 1500ms$

When the reception-phase timer is fired for a mote, either of the other two motes could be waiting to sync and the radio will be in use. The program will override the sync by using the radio to transmit a packet to this mote and then return the radio back to receiving mode.

Once a packet is transmitted during the reception phase, it is known that the the next sync phase for this mote is at $11t$ time, so the timer can be started for this time and the whole process can be repeated. If this timer fires and the program is already trying to sync a mote then the new mote is 'queued' by setting the state to 'waiting-to-sync'

There is an optimisation that the program performs. After the first reception phase of a mote, the program can calculate the beginning of the next synchronization phase. Therefore the program knows whatever payload it receives will be $n_{max}$ (The given $n$ for the mote). Subsequently, the next reception phase can be calculated

$$\text{reception}_{i+1} = \text{reception}_i + (n_{\max} + 11)t$$

This mean the program does not have to waste time going to the syncing phase every cycle (although we may like it to after some amount of time.)

The final problem to consider is when $n = 1$. This is a problem because there will not be two beacons in a single cycle and secondly the program does not know for sure that $n = 1$. This can be alleviated by waiting for the first beacon in the following cycle and calculating $t$. However, the program could potentially have to wait for $12000ms$[1] *without switching channels* else the program could miss a beacon where $n = 2$

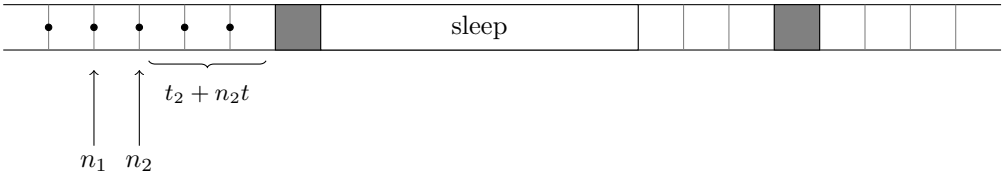


Figure 1: Timeline

## 0.2 Code

```
1  package embs;
2
3  import com.ibm.saguaro.system.*;
4  import com.ibm.saguaro.logger.*;
5
6  public class Source {
7
8    // TODO: Sync phase can get first beacon of mote0, first beacon of mote1,
9    // then second beacon of mote0, second beacon of mote2
10
11   private static final int MOTE0 = 0;
12   private static final int MOTE1 = 1;
13   private static final int MOTE2 = 2;
14
15   /* Sync states */
16   /** The program should wait for 2 beacons to calculate the next
17    * synchronization stage */
18   private static final int S_NORMAL = 0;
19   /** The program should wait for 2 beacons but the first beacon
20    * it receives should carry the highest n value for the mote. */
21   private static final int S_FINDN = 1;
22   /** The program should not wait for any beacons for this mote */
23   private static final int S_NONE = 2;
24
```

[1] Because $500ms \leq n \leq 1500ms$, the next reception phase could be anywhere between $12 \times 1500ms = 6000ms$ and $12 \times 500ms = 18000ms$

```
25    private static final int SYNC_PHASE = 0x10;
26    private static final int RECEP_PHASE = 0x20;
27
28
29    private static final long MAX_BEACON_TICKS = Time.toTickSpan(Time.MILLISECS, 1500);
30    private static final long MIN_BEACON_TICKS = Time.toTickSpan(Time.MILLISECS, 500);
31    private static final long PADDING_TICKS = Time.toTickSpan(Time.MILLISECS, 50);
32
33    private static final byte[] xmit;
34    private static final byte my_address = 0x10;
35
36    private static final Radio radio = new Radio();
37
38    /* Timer per mote */
39    private static final Timer tsend0 = new Timer();
40    private static final Timer tsend1 = new Timer();
41    private static final Timer tsend2 = new Timer();
42
43    @Immutable
44    private static final byte[] CHANNELS = new byte[] {0, 1, 2};
45    @Immutable
46    private static final byte[] PANIDS = new byte[] {0x11, 0x12, 0x13};
47
48    /** The t value for each mote **/
49    private static final long[] TIMES = new long[] {-1, -1, -1};
50    /** The next absolute sync for each mote **/
51    private static final long[] SYNC_TIMES = new long[] {-1, -1, -1};
52    /** The next absolute reception time of each mote **/
53    private static final long[] RECEP_TIMES = new long[] {-1, -1, -1};
54    /** The n value of each mote **/
55    private static final int[] NS = new int[] {1, 1, 1};
56    /** The synchronization state of each mote **/
57    private static final int[] SYNC_STATES = new int[] {S_NORMAL, S_NORMAL, S_NORMAL};
58
59    private static final long[] PREV_TIMES = new long[] {-1, -1, -1};
60    private static final int[] PREV_N = new int[] {-1,-1, -1};
61    private static final int[] QUEUES = new int[] {1, 1, 1};
62
63    /** The id of the current mote that is syncing **/
64    private static int sync_id = -1;
65
66    /**
67     * We need a way of telling whether the radio switched off because
68     * we told it to, or the radio timed out, the easiest way is to set
69     * this to true every time we turn the radio off. It will be set back
70     * to false we receive the off notice.
71     */
72    private static boolean manual_off = false;
73
74    static {
```

```
75
76      // Prepare data frame
77      xmit = new byte[12];
78      xmit[0] = Radio.FCF_DATA;
79      xmit[1] = Radio.FCA_SRC_SADDR | Radio.FCA_DST_SADDR;
80      Util.set16le(xmit, 5, 0xFFFF); // broadcast address
81      Util.set16le(xmit, 9, 0x10); // own short address
82      Util.set16le(xmit, 3, 0x0); // destination PAN address
83      Util.set16le(xmit, 7, 0x0); // own PAN address
84
85
86      tsend0.setCallback(new TimerEvent(null){
87        public void invoke(byte param, long time){
88          Source.timerCallback(param, time); }
89      });
90      tsend1.setCallback(new TimerEvent(null){
91        public void invoke(byte param, long time){
92          Source.timerCallback(param, time); }
93      });
94      tsend2.setCallback(new TimerEvent(null){
95        public void invoke(byte param, long time){
96          Source.timerCallback(param, time); }
97      });
98
99      // Radio handlers
100     radio.setRxHandler(new DevCallback(null){
101       public int invoke (int flags, byte[] data, int len, int info, long time) {
102         return Source.onReceive(flags, data, len, info, time); }
103     });
104     radio.setTxHandler(new DevCallback(null) {
105       public int invoke(int flags, byte[] data, int len, int info, long time) {
106         return Source.onSent(flags, data, len, info, time); }
107     });
108
109     // Open the default radio
110     radio.open(Radio.DID, null, 0, 0);
111     radio.setShortAddr(my_address);
112
113     setRadioForSync(MOTE0);
114
115   }
116
117   /**
118    * @return Whether there is a sync currently happening
119    */
120   private static boolean shouldWaitForSync() {
121     return sync_id != -1;
122   }
123
124   /**
```

```
125     * Stops the radio at sets {@link #manual_off} to true
126     * @see #manual_off
127     */
128    private static void stopRadioManually() {
129      manual_off = true;
130      radio.stopRx();
131    }
132
133    /**
134     * Records the state of the first beacon
135     * @param mote_id
136     * @param number the received n value
137     * @param time the time the beacon was received
138     */
139    private static void firstBeacon(int mote_id, int number, long time) {
140      log_firstBeacon(mote_id, number);
141
142      // Record the highest number of n we have found
143      // TODO: Don't need this
144      //NS[mote_id] = number > NS[mote_id] ? number : NS[mote_id];
145
146      /*
147       * If first beacon has n=1 then we know there is no more syncs but we don't know t
148       */
149      if (number == 1) {
150
151        if (SYNC_STATES[mote_id] == S_NORMAL) {
152          // Set the state to look for a n = 1
153          SYNC_STATES[mote_id] = S_FINDN;
154          Logger.appendString(csr.s2b("Find n for mote"));
155          Logger.appendInt(mote_id);
156          Logger.flush(Mote.WARN);
157        }
158
159        // There's no point trying to sync again until it has completed a new cycle
160        // TODO: Double check this
161        long time_diff = TIMES[mote_id] != -1 ? TIMES[mote_id] : MIN_BEACON_TICKS;
162
163        long min_sync_time = time + 11 * time_diff - PADDING_TICKS;
164        startTimer(mote_id, SYNC_PHASE, min_sync_time);
165
166        QUEUES[mote_id] = 0; // UNQUEUE
167
168        pickNextSync(mote_id, false);
169      }
170
171      PREV_TIMES[mote_id] = time;
172      PREV_N[mote_id] = number;
173    }
174
```

```
175   /**
176    * Records the state of the second beacon
177    * @param mote_id
178    * @param number the received n value
179    * @param time the time the beacon was received
180    */
181   private static void secondBeacon(int mote_id, int number, long time) {
182
183     log_secondBeacon(mote_id, number);
184
185
186     long time_delta = time - PREV_TIMES[mote_id];
187
188     // Divide this up by the difference in n. Usually this will just
189     // be 1 but there's a chance we missed a beacon (due to transmitting
190     // a packet) or we got 2 n=1s and should divide by 12
191     int n_delta = PREV_N[mote_id] - number;
192     if (SYNC_STATES[mote_id] == S_FINDN) n_delta = PREV_N[mote_id] + 11;
193
194     time_delta /= n_delta;
195
196     TIMES[mote_id] = time_delta;
197     QUEUES[mote_id] = 0;
198
199
200     // Completed 1 normal sync, so next time try and find n
201     if (SYNC_STATES[mote_id] == S_NORMAL) {
202       SYNC_STATES[mote_id] = S_FINDN;
203
204       Logger.appendString(csr.s2b("Find n for mote"));
205       Logger.appendInt(mote_id);
206       Logger.flush(Mote.WARN);
207     }
208
209     /*
210      * Record time difference (t) and set up the next sync time for
211      * this mote. Start the timer for reception phase in n*t time
212      * but add 1% of t to make sure we are actually in the time frame.
213      * (Sometimes the timer starts 0.006ms before the time frame)
214      */
215     long reception_time = RECEP_TIMES[mote_id] = time + number * time_delta;
216     SYNC_TIMES[mote_id] = reception_time + 11 * time_delta;
217
218     log_timeDiff(mote_id, time_delta, PREV_TIMES[mote_id], n_delta);
219     log_startRecepTimer(mote_id, reception_time);
220
221     startTimer(mote_id, RECEP_PHASE, reception_time + PADDING_TICKS);
222
223     PREV_TIMES[mote_id] = time;
224     PREV_N[mote_id] = number;
```

```
225
226       pickNextSync(mote_id, false);
227     }
228
229     /**
230      * Picks another mote to sync or if there is none queued then just stop
231      * @param mote_id The currently syncing mote
232      * @param timedout Whether this call is because of a timeout, if this is the
233      * case then retrying the same mote is preferred
234      */
235     private static void pickNextSync(int mote_id, boolean timedout) {
236
237       if (radio.getState() != Radio.S_STDBY)
238         Source.stopRadioManually();
239
240       long time = Time.currentTicks();
241
242       /*
243        * Pick a mote that needs syncing:
244        * We should try this mote again is there is a possibility that we
245        * could receive the second sync beacon
246        *
247        * Otherwise pick another mote that needs to be synced, with this
248        * mote as a last resort.
249        */
250       // TODO: check this wtf
251       boolean retry = timedout && time - PREV_TIMES[mote_id] < MAX_BEACON_TICKS
252                                    && PREV_TIMES[mote_id] != -1 && PREV_N[mote_id] != 1;
253
254
255       int next_id = retry ? mote_id :
256           QUEUES[(mote_id + 2) % 3] == 1 ? (mote_id + 2) % 3 :
257           QUEUES[(mote_id + 1) % 3] == 1 ? (mote_id + 1) % 3 :
258           QUEUES[(mote_id + 0) % 3] == 1 ? (mote_id + 0) % 3 : -1;
259
260       if (timedout) {
261         log_syncTimeout(mote_id, retry);
262       }
263
264
265       /* Set prev time to -1 unless we are retrying
266        * This means we can still record 2 beacons after a timeout */
267       // TODO: check this after moving to PREV_SYNCS
268       if (!retry && SYNC_STATES[mote_id] != S_FINDN)
269         PREV_TIMES[mote_id] = -1L;
270
271       if (QUEUES[mote_id] == 1
272           && SYNC_STATES[mote_id] == S_FINDN && next_id != mote_id) {
273         /*
274          * If we were waiting for max_n but now switching channels
```

```
275         * we will probably miss it so cancel that.
276         * */
277        SYNC_STATES[mote_id] = S_NORMAL;
278        Logger.appendString(csr.s2b("Cancel find n"));
279        Logger.flush(Mote.WARN);
280      }
281
282      Source.sync_id = -1;
283
284      // Update the radio
285      if (next_id > -1)
286        Source.setRadioForSync(next_id);
287      else
288        log_noQueue();
289    }
290
291    /** Called when a packet is received, a null is received when the radio switches off */
292    private static int onReceive(int flags, byte[] data, int len, int info, long time) {
293
294        // We want to know if we manually shut off the radio, if so do nothing.
295      if (data == null && manual_off) {
296        manual_off = false; return 0;
297      }
298
299      // Timeout
300      if (data == null) {
301
302        pickNextSync(sync_id, true);
303          return 0;
304      }
305
306      // Beacon
307      int number = data[11];
308      int mote_id = sync_id;
309
310      if (SYNC_STATES[mote_id] == S_FINDN) {
311        /*
312         * We found n, if we got this far we are probably sure that this
313         * is the correct value
314         */
315        NS[mote_id] = number;
316
317        log_foundN(mote_id, number);
318        secondBeacon(mote_id, number, time);
319
320        SYNC_STATES[mote_id] = S_NONE;
321        return 0;
322      }
323
324      // Record the state of the beacon
```

```
325       if (PREV_TIMES[mote_id] == -1)
326         firstBeacon(mote_id, number, time);
327       else
328         secondBeacon(mote_id, number, time);
329
330       return 0;
331
332     }
333
334     /** Called after a packet is sent */
335     private static int onSent(int flags, byte[] data, int len, int info, long time) {
336
337       byte mote_id = (byte) (data[3]-0x11);
338       LED.setState(mote_id, (byte) (1 - LED.getState(mote_id)));
339
340       stopRadioManually();
341
342       // Switch back the radio because we are in sync phase
343       if (shouldWaitForSync())
344         setRadioForSync(sync_id);
345
346       return 0;
347     }
348
349     /**
350      * Start a timer for a mote
351      * @param mote_id
352      * @param phase A *_PHASE const
353      * @param abs_time The absolute time the timer should be tired
354      */
355     private static void startTimer(int mote_id, int phase, long abs_time) {
356       Timer t = mote_id == 0 ? tsend0 :
357                 mote_id == 1 ? tsend1 :
358                 mote_id == 2 ? tsend2 : null;
359
360       t.setParam((byte) (mote_id | phase));
361       t.setAlarmTime(abs_time);
362     }
363
364
365     /** Called when a timer has fired */
366     private static void timerCallback(byte param, long time) {
367       if ((param & 0xF0) == SYNC_PHASE)
368         startSyncPhaseFromTimer((byte) (param & 0xF), time);
369       else if ((param & 0xF0) == RECEP_PHASE)
370         receptionPhase((byte) (param & 0xF), time);
371     }
372
373
374     /** Starts syncing after the timer has run */
```

```
375   private static void startSyncPhaseFromTimer(byte param, long time) {
376     int mote_id = param;
377
378     QUEUES[mote_id] = 1;
379
380     if (shouldWaitForSync()) {
381       /*
382        * If a mote is already trying to sync then we should
383        * not sync this mote but add it to a queue instead.
384        *
385        * UNLESS n is 2 because we will totally miss the sync
386        * phase otherwise
387        */
388
389       // TODO: check this
390       if (NS[mote_id] == 2 && NS[sync_id] == 1) {
391         stopRadioManually();
392         setRadioForSync(mote_id);
393         return;
394       }
395
396       log_missSync(mote_id, sync_id);
397
398       /*
399        * We might have been expecting to get n here, in which case
400        * we will probably miss n because we are queueing it for later.
401        * Just do a normal sync instead
402        */
403       if (SYNC_STATES[mote_id] == S_FINDN) {
404         Logger.appendString(csr.s2b("Cancel n=1 for mote "));
405         Logger.appendInt(mote_id);
406         Logger.flush(Mote.WARN);
407         SYNC_STATES[mote_id] = S_NORMAL;
408       }
409
410       return;
411     }
412
413     Logger.appendString(csr.s2b("Starting sync mote"));
414     Logger.appendInt(mote_id);
415     Logger.appendString(csr.s2b(" from timer"));
416     Logger.flush(Mote.WARN);
417
418     setRadioForSync(mote_id);
419   }
420
421   /** Called when the timer has fired because of a reception phase */
422   private static void receptionPhase(byte param, long time) {
423
424
```

```
425     if (shouldWaitForSync()) {
426       // I think sending a packet should override sync phase
427
428       if (SYNC_STATES[sync_id] == S_FINDN) {
429         /*
430          * The syncing phase is waiting for n, if it misses the first beacon
431          * then n will be WRONG which will mess up the reception phase
432          * There is a chance here that switching to transmit will
433          * miss the sync beacon. So I think we should try and find n next
434          * time instead.
435          */
436         SYNC_STATES[sync_id] = S_NORMAL;
437       }
438
439       // Note: Make sure to start syncing again after transmission
440       stopRadioManually();
441     }
442
443     /*
444      * RECEP_TIMES[mote_id] is the time the timer should have started,
445      * the timer may have been fired a bit late so don't use time.
446      */
447     int mote_id = param;
448
449     log_recepPhase(mote_id, RECEP_TIMES[mote_id], time);
450
451     log_sendPacket(mote_id);
452
453     byte pan_id = PANIDS[mote_id];
454
455     radio.setChannel(CHANNELS[mote_id]);
456     radio.setPanId(pan_id, false);
457     radio.startRx(Device.ASAP, 0, Time.currentTicks() + Time.toTickSpan(Time.SECONDS, 1));
458
459     Util.set16le(xmit, 3, pan_id); // destination PAN address
460     Util.set16le(xmit, 7, pan_id); // own PAN address
461     radio.transmit(Device.ASAP | Radio.TXMODE_POWER_MAX, xmit, 0, 12, 0);
462
463     /*
464      * If we wanted to sync again after a certain time, we could do it here
465      */
466
467     /* If we don't want the mote to sync then we sould start the reception
468      * phase again in 1 cycle */
469     if (SYNC_STATES[mote_id] == S_NONE) {
470       long next_recep_time = RECEP_TIMES[mote_id] + (11 + NS[mote_id]) * TIMES[mote_id];
471
472       log_startRecepTimer(mote_id, next_recep_time);
473       Logger.appendString(csr.s2b("11 + "));
474       Logger.appendInt(NS[mote_id]);
```

```
475       Logger.appendString(csr.s2b(" * "));
476       Logger.appendLong(TIMES[mote_id]);
477       Logger.flush(Mote.WARN);
478
479       RECEP_TIMES[mote_id] = next_recep_time;
480
481       startTimer(mote_id, RECEP_PHASE, next_recep_time + PADDING_TICKS);
482     }
483     else
484       startTimer(mote_id, SYNC_PHASE, SYNC_TIMES[mote_id] - PADDING_TICKS);
485   }
486
487   /**
488    * Starts the radio for syncing phase. Radio must be in standby
489    * at this point
490    * @param mote_id
491    */
492   private static void setRadioForSync(int mote_id) {
493
494     log_radioSync(mote_id);
495
496     Source.sync_id = mote_id;
497
498     // Set channel
499     radio.setChannel(CHANNELS[mote_id]);
500     radio.setPanId(PANIDS[mote_id], false);
501
502     radio.startRx(Device.ASAP, 0, Time.currentTicks() + MAX_BEACON_TICKS);
503   }
504
505
506
507   /*
508    *
509    * Logging
510    *
511    */
512
513
514
515   private static void log_firstBeacon(int mote_id, int number) {
516     Logger.appendString(csr.s2b("Receive first beacon for mote"));
517     Logger.appendInt(mote_id);
518     Logger.appendString(csr.s2b(" where n is "));
519     Logger.appendInt(number);
520     Logger.flush(Mote.WARN);
521   }
522   private static void log_secondBeacon(int mote_id, int number) {
523     Logger.appendString(csr.s2b("Receive second beacon for mote"));
524     Logger.appendInt(mote_id);
```

```
525       Logger.appendString(csr.s2b(" where n is "));
526       Logger.appendInt(number);
527       Logger.flush(Mote.WARN);
528     }
529
530     private static void log_radioSync(int mote_id) {
531       Logger.appendString(csr.s2b("Waiting to sync mote"));
532       Logger.appendInt(mote_id);
533       Logger.flush(Mote.WARN);
534     }
535
536     private static void log_recepPhase(int mote_id, long recep_time, long now_time) {
537       Logger.appendString(csr.s2b("Starting recep phase for mote"));
538       Logger.appendInt(mote_id);
539       Logger.appendString(csr.s2b(" at time "));
540       Logger.appendLong(Time.fromTickSpan(Time.MILLISECS, recep_time));
541       Logger.appendString(csr.s2b("ms (+"));
542       Logger.appendLong(Time.fromTickSpan(Time.MILLISECS, now_time-recep_time));
543       Logger.appendString(csr.s2b("ms)"));
544       Logger.flush(Mote.WARN);
545     }
546
547     private static void log_timeDiff(int mote_id, long time_diff, long prev_time, int
          n_delta) {
548       Logger.appendString(csr.s2b("Diff for mote"));
549       Logger.appendInt(mote_id);
550       Logger.appendString(csr.s2b(" is "));
551       Logger.appendLong(Time.fromTickSpan(Time.MILLISECS, time_diff));
552       Logger.appendString(csr.s2b("ms (prev_time was "));
553       Logger.appendLong(Time.fromTickSpan(Time.MILLISECS, prev_time));
554       Logger.appendString(csr.s2b("ms and n_delta is "));
555       Logger.appendInt(n_delta);
556       Logger.appendString(csr.s2b(")"));
557       Logger.flush(Mote.WARN);
558     }
559
560     private static void log_syncTimeout(int mote_id, boolean retry) {
561       Logger.appendString(csr.s2b("Mote"));
562       Logger.appendInt(mote_id);
563       Logger.appendString(csr.s2b(" took too long to sync"));
564       if (retry)
565         Logger.appendString(csr.s2b(", but I will try again"));
566       Logger.flush(Mote.WARN);
567     }
568
569     private static void log_foundN(int mote_id, int number) {
570       Logger.appendString(csr.s2b("I am sure that mote"));
571       Logger.appendInt(mote_id);
572       Logger.appendString(csr.s2b(" has n = "));
573       Logger.appendInt(number);
```

```
574       Logger.flush(Mote.WARN);
575     }
576
577     private static void log_sendPacket(int mote_id) {
578       Logger.appendString(csr.s2b("Sending packet to mote"));
579       Logger.appendInt(mote_id);
580       Logger.flush(Mote.WARN);
581     }
582
583     private static void log_noQueue() {
584       Logger.appendString(csr.s2b("No queued motes. Sleep."));
585       Logger.flush(Mote.WARN);
586     }
587
588     private static void log_missSync(int mote_id, int cause_id) {
589       Logger.appendString(csr.s2b("Mote"));
590       Logger.appendInt(mote_id);
591       Logger.appendString(csr.s2b(" missed sync phase due to mote"));
592       Logger.appendInt(cause_id);
593       Logger.flush(Mote.WARN);
594     }
595
596     private static void log_startRecepTimer(int mote_id, long at_time) {
597       Logger.appendString(csr.s2b("Next recep phase for mote"));
598       Logger.appendInt(mote_id);
599       Logger.appendString(csr.s2b(" should be at "));
600       Logger.appendLong(Time.fromTickSpan(Time.MILLISECS, at_time));
601       Logger.appendString(csr.s2b("ms"));
602       Logger.flush(Mote.WARN);
603     }
604
605   }
```