Channel Activity Display: How to ensure your LoRa packets are sent properly

Background

Currently, the LoRa networks, either ad-hoc P2P networks, or under the LoRaWAN specification, use an ALOHA media access control mechanism, ie shoot and forget, defined by radio frequency, bandwidth and spreading factor (SF) of the chirp spread spectrum (CSS) modulation. While ALOHA allows for a simple network implementation, it can't keep up with the increasing demand of IoT devices. Even if devices conform to channel usage limitations (0.1% or 1% duty cycle in Europe), the ALOHA-based LoRa networks will have degraded network performance commensurate to the growth of IoT networks.

The Channel Activity Detection (CAD) feature available within the whole family of LoRa radios presents a possible CSMA mechanism for LoRa networks. CAD is an energy-efficient way to detect an incoming frame without resorting to power-hungry continuous receiving mode. Prior to performing a CAD, the LoRa radio is set on the desired Freq, SF and BW. The radio then switches to the CAD mode and performs a CAD operation, which lasts [Tsym + (32/BW)] milliseconds, during which the radio performs a receive operation correlation on the received samples. Tsym is the duration that is the airtime of a single LoRa chirp, depending on the SF value. BW is the bandwidth of the LoRa channel. The full range of BW values is {10.4, 15.6, 20.8, 31.25, 41.7, 62.5, 125, 250, 500} kHz for LoRa P2P, and restricted to the last 3 for LoRaWAN.

The CAD mode available on all of LoRa radios (like the SX126x, SX127x series) is primarily designed for energy-efficient preamble detection. Preamble detection is not a full-fledged carrier sense. However, studies show that CAD in the newer sx126x series can also detect reliably payload chirps of an ongoing transmission. CAD is also power efficient.

The CAD operation

The CAD operation itself involves the LoRa module listening, at the preset frequency and SF/BW settings, for a LoRa preamble. If it finds that, it returns a busy signal to the callback. In some chips ***(Olivier, est-ce le cas pour toutes ou seulement certains modèles ?)*** the LoRa module will also try to listen further in order to detect whether a LoRa packet perse, not the preamble. Again, if it is detected, it will return a busy status. If not, the callback receives an all-clear status, and the user can proceed with the transmission.

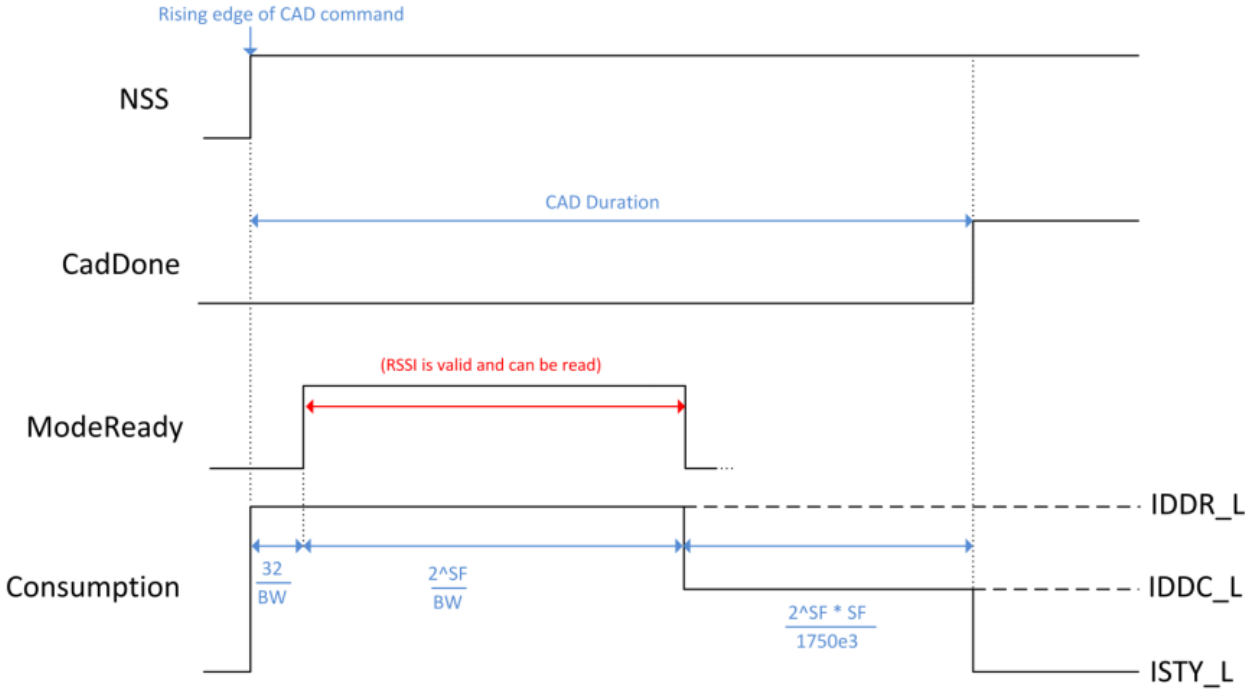

Figure 5. LoRa CAD Timing

From *SX1272/3/6/7/8: LoRa Modem, Low Energy Consumption Design* by Semtech.

When the CAD operation starts, the first 32/BW are not available for use. Then for the next (2^SF)/BW the RSSI is valid and readable. The chip calls the ModeReady interrupt. At the end of ModeReady, there is a short processing time, (2^SF*BW)/1,750e3, spent at a reduced consumption level. Once the channel activity detection process is complete the radio returns to standby mode and the CadDone interrupt is set. At this point the CadDetected interrupt can be checked – indicating the presence, or otherwise, of a valid preamble upon which to wake the receiver.

Note that the radio returns to standby mode to allow the CadDetected and CadDone interrupts to be read, these are then cleared automatically upon returning to sleep mode.

In Practice

If you're not an RF engineer (hi! I'm not an engineer!) how does this work in practice? Basically, if you've been doing LoRa P2P transmissions, or had a look at ping-pong code, you will have noticed that most of the time, the code sending a packet just starts a transmission (which is really just a request to get ready for now), sends data to the LoRa module, and closes, which starts the transmission in earnest. What happens next is left to the LoRa module.

Likewise, when receiving packets, it is done synchronously: you ask the LoRa module whether there's anything incomin, and if there is, you read from it. If not, you skip, until the next time. This usually happens in the `loop()` event, or similar, depending on the framework you use.

With CAD, not so much. You can have a code example that has a completely empty `loop()` event, and still be properly functional. The secret is to set up a `CadDone()` callback, which will handle the result of a CAD operation. Likewise, you set up callbacks for all Rx and Tx events (RxDone, RxError, TxDone, TxError). Say you want to implement a LoRa machine that sends a small set of data every xx minutes. Like, a sensor pod in a farm. Sounds easy. Except you don't have 1 pod, but 50. And even if you stagger the transmissions, 50 within 15 minutes, for a transmission time of 20 to 30 seconds, this is going to be messy, and you will probably have lots of errors.

Setting up radio events on a RAK4631

```
// Initialize the Radio callbacks
RadioEvents.TxDone = OnTxDone;
RadioEvents.RxDone = OnRxDone;
```

```
  RadioEvents.TxTimeout = OnTxTimeout;
  RadioEvents.RxTimeout = OnRxTimeout;
  RadioEvents.RxError = OnRxError;
  RadioEvents.CadDone = OnCadDone;
  // Initialize the Radio
  Radio.Init(&RadioEvents);
```

So instead of sending unconditionally, when it's time to send, you ask for a CAD operation. When this is done it returns with a status code: busy or available. If the channel is busy, you queue the transmission and try again later. If it's free, you send the data – and wait until the `TxDone` callback is called to remove that packet from the queue: between the time you did the CAD operation and the time you started the transmission, things may have gone wrong. Here's what the whole process looks like:

Sending a LoRa packet the right way

```
/**
   @brief main send function that sets things up for sending.
*/
void send() {
  // Set up a buffer of 64 bytes with a string, and the remainder made up
  // of random bytes to make a nice, fat packet that shows up nicely on an SDR.
  sprintf((char*)TxdBuffer, "Received at RSSI %d, SNR %d", myRSSI, mySNR);
  uint8_t bufLen = strlen((char*)TxdBuffer) + 1;
  uint8_t remainderLen = 64 - bufLen;
  fillRandom(TxdBuffer + bufLen, remainderLen);
  Serial.println("Sending:");
  // show the buffer in hex format
  hexDump(TxdBuffer, 64);
  Radio.Standby();
  // set up CAD
  Radio.SetCadParams(LORA_CAD_08_SYMBOL, LORA_SPREADING_FACTOR + 13, 10, LORA_CAD_ONLY, 0);
  // To determine how long the CAD operation took. Optional.
  cadTime = millis();
  Radio.StartCad();
}

/**
   @brief CadDone callback: is the channel busy?
*/
void OnCadDone(bool cadResult) {
  time_t duration = millis() - cadTime;
  if (cadResult) {
    // true = busy / Channel Activity Detected
    Serial.printf("CAD returned channel busy after %ldms\n", duration);
    // At this junction we should re-schedule a send. Here, we'll just ignore.
    // Set the Radio back to listening mode.
    Radio.Rx(RX_TIMEOUT_VALUE);
  } else {
    Serial.printf("CAD returned channel free after %ld ms\nSending...", duration);
    // Good to go: let's send!
    Radio.Send(TxdBuffer, 64); // strlen((char*)TxdBuffer)
    Serial.println(" done!");
  }
}

/**
   @brief Function to be executed on Radio Tx Done event
*/
void OnTxDone(void) {
  Serial.println("OnTxDone");
  Radio.Rx(RX_TIMEOUT_VALUE);
}
```

```
/**
   @brief Function to be executed on Radio Tx Timeout event
*/
void OnTxTimeout(void) {
  Serial.println("OnTxTimeout");
  Radio.Rx(RX_TIMEOUT_VALUE);
}
```