Pawel Gburzynski

# **Heart Monitor**
# Functionality and External Interfaces

Version 0.06
February 2007

## Preamble

This note describes the functionality and communication protocol used by the Heart Monitor Node (abbreviated in the sequel as *node*). This specification is tentative and reflects the (untested) version of firmware as of March 3, 2007.

## The operation

When switched on, the node starts up in the UNBOUND state. This means that it does not know how to talk to its Access Point (AP). The access point can identify itself to the node over its serial (RS232) interface or over the wireless (RF) link.

### *Binding*

While in the UNBOUND state, the node periodically, at slightly randomized 4 second intervals, broadcasts a HELLO message over both interfaces. Such a message includes the node's unique serial number (ESN). An AP perceiving a HELLO message will know that the node is in the neighborhood. Thus, it can bind the node to itself by sending it a BIND request.

A node can be bound on one interface at a time. While bound on one interface, the node does not send the HELLO messages on the other interface, but it still listens on it for a possible BIND request. Should such a request arrive, it will re-bind the node, i.e., the node will drop its previous binding and establish a new one. Any BIND requests arriving on a bound interface are ignored until the node is unbound (by hard reset or by receiving an UNBIND request from the AP).

Binding consists in assigning to all transmissions from/to the node (over the bound interface) a specific *link identifier* (LI), which is a nonzero 16-bit number. When unbound, the node transmits and listens using a reserved LI of zero. A message sent with this LI is received by the recipient even if its specific LI is nonzero. Thus, the AP can receive all HELLO requests regardless of the LI to which it is specifically tuned at the time. Also, an unbound node (using the LI of zero) can receive a BIND request arriving with a nonzero LI. This feature may not appear very useful for communication over RS232 (at least as long as there are only two parties), but with the RF interface it allows us to have multiple access points claiming different nodes from the global pool.

### *Protocol and reliability*

The message format is identical over both interfaces. The protocol is also identical, with minor differences regarding the timing of some intervals. In both cases, the protocol provides for reliable communication; in particular, it does not assume that the RS232 interface is error-less.

The reliability is accomplished in a relatively simple manner. Instead of a complicate structure of dedicated acknowledgments, the protocol relies on the idempotent nature of requests (sent from the AP to the node) and holistic nature of the status reports sent from the node to the AP. The simple idea is that in order to make sure that its request has been accepted by the node, the AP may simply repeat the same request over and over again until it receives a status report indicative of a success. The same idea has been incorporated into the operation of reliable streaming of sampled data from the node to the AP.

***Node states***

A bound node can be in one of three states:

| | |
|---|---|
| READY | meaning idle, i.e., neither collecting a sample, nor sending a sample to the AP, |
| SAMPLING | meaning collecting a sample and storing it in flash memory, |
| SENDING | meaning transmitting a sample from flash memory to the AP. |

Independently of its state, a node may be collecting heart rate pulses, calculating the heart rate, and transmitting it to the AP at 3 sec intervals. This can also be done when the node is READY. There is a way to synchronize heart rate monitoring to SAMPLING (meaning that the two can be started and stopped simultaneously), but it is essentially an independent feature.

***Samples***

By a sample we understand here a complete stream of data, i.e., a take of a series of numbers from the ADC, as requested by the AP, collected 500 times per second (8 16-bit values) and stored in flash memory at the node. To avoid confusion, a single set of 8 numbers contributing to a sample, i.e., a single ADC strobe, will be called a *sampling*. Samples are identified by integer numbers from 0 to 255. The sample identifiers (IDs) play two roles. First, they allow a single node to store multiple independently identifiable samples (up to eight) at any given time. Extracts from those samples can be requested by the AP by specifying their IDs. Second, they are needed to make AP requests idempotent. By specifying sample IDs in its requests, the AP can differentiate between an old request (e.g., dealing with a different sample) and a new one. Also, sample IDs are included in STATUS messages sent by the node to the AP, where they avoid ambiguities, e.g., in interpreting requests being in a partial state of fulfillment.

At any moment the node may have certain samples, possibly none, stored in its local database. The identifiers of those samples are presented as STATUS reports to the AP. When requesting a new sample, the AP must specify an ID that is different from those already present. When that ID shows up in a STATUS report sent from the node, it will mean that the request has been accepted (or processed).

While collecting new samples, the node erases (forgets) old ones in a round-robin fashion. First, the maximum number of samples stored at a node is 8 regardless of their space requirements. Second, the new sample may overwrite one or more previous samples because of the limited storage size. In such a case, the overwritten samples will be discarded. Again, this overwriting is carried out in a cyclic fashion interpreting the flash storage as a single circular buffer. This approach also spreads the utilization of flash memory evenly over time.

***Heart rate monitoring***

The heart rate monitor is an independent thread, which can be started and stopped by separate requests from the AP, as well as (optionally) by sample collection requests. When active, the thread collects pulses from a digital port and averages them into the heart rate expressed in beats per minute. The resultant number is transmitted at 3 second intervals as a separate message and also displayed on the LCD. The algorithm for calculating (averaging) the heart rate is as follows.

An array consisting of four slots stores the number of beats received within the last 12 seconds, with 3 seconds per slot. Let $R_i$ be the value in slot $i$, with slot 0 being the most recent one. The thread calculates this number:

$$B = 10 \times R_0 + 6 \times R_1 + 3 \times R_2 + 1 \times R_3,$$

which can be viewed as an approximate exponential moving average over the four slots. Note that the factors add up to 20, which multiplied by 3 (seconds per sample) yields 60. Thus, the average is normalized to a minute. Then, the thread sets

$$Rate = B \times 0.75 + Rate \times 0.25,$$

where $Rate$ is the updated average rate (its previous value is from 3 seconds ago).

***LCD display***

The LCD display is partitioned in the following manner. Line 1 shows the status of the device. Characters 0 through 7 can be: “UNBOUND”, “READY”, “SAMPLING”, “SENDING”, in straightforward correspondence to the node's state. All texts except “READY” (considered the stable idle state) blink. For states SAMPLING and SENDING, characters 9-11 display the 3-digit decimal ID of the *current sample*. This is the sample currently being processed by the node. For state READY, these characters are blanks. Character 13 is a single digit 0-8 showing the number of samples currently stored in the node's database. Character 15 is either R or S, depending on whether the RF (radio) or RS232 (serial) interface is bound at present.

The second line displays two numbers. If the heart rate monitor is on, characters 0-3 display the current heart rate. For states SAMPLING, and SENDING, characters 20-31 display the number of samplings still remaining to be processed (collected or sent).

## The protocol

Both interfaces use the same protocol. This gives us the highest flexibility. In principle, a node can be connected directly to a PC via its RS232 interface. It can also be connected via the RF link to an AP, connected in turn to a PC via USB. The AP can act a simple forwarder of commands between the PC and the node.

***Message format***

A message (or a packet)[1] consists of an even number of bytes. If the actual content of a message calls for an odd number of bytes, a dummy byte is added (always immediately before the CRC) to round it up to an even number. As we said earlier, the message format is identical in both cases, i.e., RS232 and RF, with a minor exception regarding the framing. The exact framing of the packet for RF transmission is of no concern here (PicOS drivers take care of that).

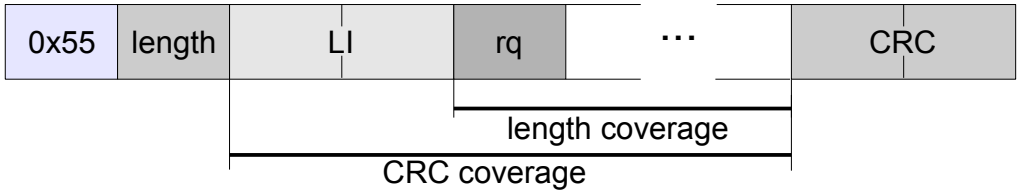


**Figure 1: RS323 packet format.**

[1] In our design, a packet always conveys a complete message (a single message is never split over multiple packets).

Figure 1 shows the complete structure of a packet sent over the RS232 interface. The *length* field (one byte) gives the number of bytes in the packet payload, which extends from *rq* up to the last byte preceding CRC. Note that this number is always even. All information exchange between a node and an AP is carried out using packets like this. The first two bytes can be viewed as the "physical framing" of the packet, which is specific to RS232 and replaced with something different for RF transmission. The maximum length of a packet, including the physical framing bytes and the CRC, is 62 bytes.

When the node praxis[2] (run under PicOS) creates a packet for transmission or acquires a received packet from the interface, it never sees the physical framing bytes, and it never worries about the CRC. Thus, from its viewpoint, it makes no difference over which interface the packet has arrived (or will go). However, if you want to program a PC application capable of talking directly to a node over RS232, you have to be aware of all the details contributing to the packet format shown in Figure 1, including the calculation of CRC. Note that RF transmission requires a special AP interface. A simple approach to implementing such an interface is to make it act as a packet converter between RS232 and RF (which, of course, will run under PicOS).

***CRC calculation***

The CRC does not cover the physical framing bytes, i.e., its coverage starts with the link ID (LI). A packet always consists of an even number of bytes; thus, the CRC is calculated assuming that the packet is an array of 16-bit words. The algorithm is an industrial standard named ISO-3309. The source code is available in PICOS/Libs/Lib/checksum.c.

For checksum insertion (for an outgoing packet), the algorithm is run through all words, starting from LI and including the last word preceding the CRC field. Then, the calculated value is inserted into the CRC field. Here is the reference code:

```
void insert_checksum (word *packet, int length) {
       packet [length – 1] = w_chk (packet + 1, length – 2, 0);
}
```

where **`packet`** represents a full packet (including the physical framing bytes) viewed as a sequence of words, **`length`** is the the packet in words, including LI and CRC, and **`w_chk`** is the PicOS function implementing the 16-bit ISO-3309 CRC algorithm (see PICOS/Libs/Lib/checksum.c). Note that for this calculation the 16-bit words comprising the packet are viewed as **little-endian**. The same function will work on the MSP430 as well as on the PC.

For CRC verification in a received packet, the following code should be applied:

```
Boolean checksum_ok (word *packet, int length) {
       return w_chk (packet + 1, length, 0) != 0;
}
```

where **`packet`** and **`length`** are as before.

In the following discussion, we shall ignore the physical framing bytes, the link ID (LI) field, and the CRC. The packet stripped of those items is called the *payload* (it corresponds to the length coverage in Figure 1).

[2]PicOS terminology for "application."

***Commands directed from AP to node***

The *rq* field (a single byte) in Figure 1 identifies the packet type, i.e., the kind of information the packet carries. If the leftmost bit of *rq* is 0, it means that the packet originates at an AP and is addressed to a node. Otherwise, the packet originates at a node and is intended for an AP. We start from those packets that originate at Access Points.

**0x00 BIND** payload: [0x00, ESN, dummy]

This is a BIND request. The payload consists of the request code byte (0x00), a four-byte ESN, and one dummy byte to round up the payload length to an even number of bytes. The link ID field of the packet must be nonzero. The ESN is sent in the network (**big-endian**) format, with its most significant byte going first.

If the ESN matches the node's ESN, and the node is not bound already, it interprets this message as a request to bind itself to the interface over which the packet has arrived. The binding consists in assigning the link ID of the BIND packet to all subsequent communication over the interface.

Note that a node that is not bound uses the link ID of zero, which is promiscuous in the sense that it allows the node to receive packets with any link ID. Once a node becomes bound, is sets the link ID to the prescribed value. The link ID is interpreted internally by the interface driver. If nonzero, it will automatically block all packets with different link IDs, except for those with the link ID of zero. Thus, in particular, the node won't be able to receive other bind request over the interface, except for possible duplicates of the original request.

How can the AP know that its BIND request has been received and accepted by the node? There are no explicit acknowledgments in our scheme. However, a node that is not bound will be sending periodic HELLO messages over both interfaces. When a node becomes bound, it will stop sending HELLO messages and send a STATUS message, which will implicitly indicate that the BIND request has been accepted and honored. If the node receives another BIND request while bound (i.e., a duplicate of the previous request), it will respond with another STATUS message and ignore the duplicate. Thus that AP can simply send BIND requests to the node until it receives from it the first STATUS message.

**0x40 UNBIND** payload: [0x40, dummy]

This command will revert the node to the unbound state. The node will reset its link ID to zero and start sending HELLO messages on both interfaces.

**0x10 RESET** payload: [0x10, dummy]

The node will stop all operations, erase all samples from its database, and unbind itself. This effectively simulated the operation of powering the node up.

**0x20 STOP** payload: [0x20, dummy]

The node will stop all activities, including health rate monitoring, and send a STATUS message. If the node does not carry out any activities at the moment, it will just respond with a STATUS message.

**0x30 REPORT** payload: [0x30, dummy]

Without interrupting its present activities, regardless of its current state, the node will respond with a STATUS message.

**0x50 SAMPLE** payload: [0x50, sample ID, N, dummy]

This is a request to start sampling. The arguments are *sample ID* (1 byte) followed by the required number of samplings *N*, which corresponds to the sample duration expressed in 1/500 sec. For example, *N* = 4000 means 4000 samplings at 1/500 second intervals, which translates into 8 seconds. *N* is expressed as an unsigned 3-byte (**big-endian**) number from 0 to 16777215.

Here is how the request is interpreted. If the node is currently busy (i.e., SAMPLING or SENDING), it ignores the request and responds with a STATUS message. Note that this message will notify the AP that the node is busy. If the node is READY, it will check whether a sample with the indicated ID is already present in its database. If that happens to be the case, the node will ignore the request and respond with a STATUS message. Note that a STATUS message sent by a READY node includes the list of samples in its database.

If the node is READY and it doesn't have a sample with the specified ID, it will immediately start sampling and send a STATUS message. That message will tell the AP that the node is now collecting the requested sample.

When the node completes the sample collection, it will send a STATUS message announcing its READY status. That message will include the new list of samples in the database. This way, the AP will be able to learn that the sample has been acquired.

Of course, one should be aware that the STATUS message indicating the end of sample collection can be lost. Thus, the AP should periodically inquire for the node's status (see the REPORT request) if it does not receive the STATUS report in due time.

**0x51 SAMPLEH** payload: [0x51, sample ID, N, dummy]

This is an alternative SAMPLE request which simultaneously starts the heart rate monitor. The health rate monitor will be terminated when the entire sample has been collected.

If the heart rate monitor is active when the request is received, it remains active, but it is marked to be terminated at the end of sample collection. If the heart rate monitor is active when a SAMPLE (not SAMPLEH) request is received, it will remain active during the sample collection and it **will not** be terminated when the sample has been collected.

**0x70 HRMON** payload: [0x70, dummy]

Having received this request, the node will start the heart rate monitor and send a STATUS message. If the monitor is already active, only the STATUS message will be sent. The request is accepted regardless of the node state (as long as the node is bound).

**0x71 HRMOFF** payload: [0x71, dummy]

This request unconditionally stops the heart rate monitor and triggers a STATUS message. No action, other than sending the STATUS message, is performed if the monitor is already off.

**0x60 SEND** payload: [0x60, sample ID, offset, length]

This is a request to send a fragment of the indicated sample to the AP. The same request format is used to convey a positive or negative acknowledgment to the node during a sample transmission. The arguments are: the single-byte *sample ID* (as for SAMPLE), the three byte big-endian *offset* giving the initial number of the sampling to be sent (counting from zero), and the three-byte big-endian *length*, which gives the total number of samplings to be sent. The arguments allow the AP to select an arbitrary continuous portion of the complete sample for transmission.

If the node is READY when it receives a SEND request, it looks up the indicated sample ID in its database. If the sample is found, the node checks whether the specified offset falls within the sample. If not, the node does nothing and responds with a STATUS message. Otherwise, it changes its state to SENDING and sends an initial portion of the requested chunk of samplings as a series of SDATA packets (as explained below).

If the node is SAMPLING when a SEND request is received, the node ignores the request and sends back a STATUS message.

If the node is SENDING while it receives a SEND request, it first checks whether the sample ID in the request matches the ID of the sample currently being sent. If not, the new request is ignored and a STATUS message is sent. Otherwise, the SEND request it is interpreted as a positive or negative acknowledgment, as described below.

***Messages addressed from node to AP***

A message sent by a node (as opposed to one sent by an AP) has the most significant bit in the *rq* field set to 1.

**0x80 HELLO** payload: [0x80, ESN, dummy]

The message includes the four-byte ESN of the node in the **big-endian** format. This message is sent periodically, with the link ID of zero, at randomized 4-second intervals, on both interfaces, for as long as the node remains unbound.

**0xD0 HRATE** payload: [0xD0, rate]

The message carries the last heart rate value as a single byte, i.e., an integer number of averaged beats per minute between 0 and 255. Such a message is sent by the node regularly at 3 second intervals while the heart rate monitor is on.

Note that the heart rate monitor may be active, and HRATE messages may be sent, while the node is transmitting a sample to the AP. In such a case, the HRATE messages will be interleaved with SDATA packets (see below).

**0x9X STATUS** payload depends on the node state

A message whose first 4-bit nibble of the *rq* code is 9 carries information about the node state. It is returned at the reception and completion of most requests sent to the node by the AP. The entire 8-bit *rq* code is interpreted as shown in Figure 2, where the value of *state* can be:

0 – the node is READY
1 – the node is SAMPLING
2 – the node is SENDING

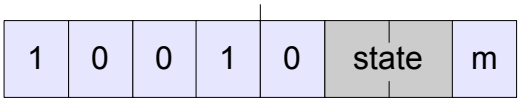


**Figure 2: Node status code.**

Additionally, the *m* bit indicates whether the heart monitor is on (1) or off (0). The remaining bytes of the payload depend on *state* as follows:

**READY:** the message contains the list of samples currently present in the node's database, with each sample occupying 4 consecutive bytes as illustrated in Figure 3. The total payload size is 2 + *N* x 4 bytes, where *N* is the number of samples currently stored at the node. The message ends with one dummy byte *(d)* rounding up its length to an even number of bytes.

The *length* field is a 3-byte unsigned **big-endian** integer from 1 to 16777215 (it is never zero) describing the sample length, i.e., the total number of samplings in the sample.


**Figure 3: READY status message format.**

Note that when the node happens to store no samples at the moment, the message will consist solely of the *rq* field immediately followed by a dummy byte. The number of samples never shows explicitly in the message and must be deduced from its length.

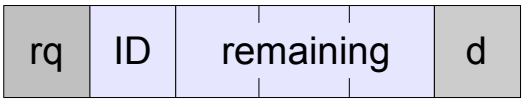


**Figure 4: SAMPLING status message format.**

**SAMPLING:** the message looks as shown in Figure 4 and informs the AP about the operation's progress. The *ID* field identifies the sample being acquired, and *remaining* is a **big-endian** 3-byte unsigned integer telling the number of samplings still remaining to be collected.


**Figure 5: SENDING status message format.**

**SENDING:** the message looks as shown in Figure 5 and informs the AP about the operation's progress. The *ID* field identifies the sample being sent. Both fields following the sample ID are **big-endian** 3-byte unsigned integer numbers. As before, *remaining* tells the number of samplings still remaining to be sent, while *next to ACK* identifies the first sampling that has not been acknowledged yet (see below).

**0xC0 SDATA** payload: [0xC0, sample ID, offset, remaining, s0, s1, s2]

This message contains a chunk of sample data, up to three consecutive samplings, being sent by the node to the AP. The *sample ID* field identifies the sample to which the fragment belongs, *offset* tells the number of the first sampling within the sample (counting from zero), and *remaining* is the number of samples still remaining to be sent, including the ones carried in the present message.

Normally, if *remaining* is greater than 2, the message contains three samplings. Each sampling is a sequence of 8 16-bit numbers, which means that the three samplings take 48 bytes altogether. Thus, the payload length is 56 bytes in that case. The 16-bit numbers in the samplings are in **little-endian** format, i.e., the less significant byte goes first.

The last SDATA message for a given sample may contain less than 3 samplings. This will happen when the requested number of samplings was not divisible by 3. In that case, *remaining* in the last message will be 1 or 2 (it can never be zero).

***The sample transfer protocol***

The protocol is quite straightforward and follows a simple window-based scheme with go-back retransmissions and acknowledgments. We are reluctant to call the latter "acknowledgments" because an acknowledgment looks exactly like a request to send a specific portion of the sample. Thus it is better to view the acknowledgments as prompts to send more fragments of the requested sample.

Having received the original SEND request in state READY, the node starts to transmit the selected portion of the sample in SDATA packets. When the AP receives a new chunk of data that adds to the continuous sequence of samplings belonging to the requested sample, it should "acknowledge" the reception with a new SEND request specifying the next "expected" offset. In a simple (window-less) case, a sample transmission might look as shown in Figure 6.

We assume that the initial SEND message is received by the Node in state READY. It requests 1000 samplings starting from the offset 0, i.e., from the beginning of the sample. The node responds with the first chunks of 3 samplings. Having received those samplings, the AP responds with an updated SEND request, and so on. Note the final request for zero samplings. That request provides the final acknowledgment to the node, which assumes that the transfer has been completed. Having received such a message, the node reverts to the READY state.

When we look at Figure 6, we see that the individual chunks of the requested sample arrive in response to separate requests, which look exactly as the original request with appropriately updated parameters. Thus there is no apparent reason why the node should have a separate SENDING state: it could simply view the individual chunks as independent replies to independent requests. The reason why we want to have the SENDING state is the need to speed up the transfer and reduce the number of feedback messages from the AP.

Having sent the first SDATA message, the node is allowed to send a few messages ahead without waiting for separate requests for the subsequent chunks. This is what we mean by the window. While in the SENDING state, the node maintains two counters labeled *NextToGo* and *NextToAck*. The first counter, *NextToGo*, tells the number of next sampling to be transmitted, while *NextToAck* gives the first number of a sampling for which an acknowledgment (meaning SEND request) has not been received yet. The node is allowed to keep sending samples until *NextToGo* – *NextToAck* is below a

threshold called the window size. This threshold is set to 12 samples (i.e., 4 messages) for the RF interface and 48 samples (i.e., 16 messages) for the RS232 interface.

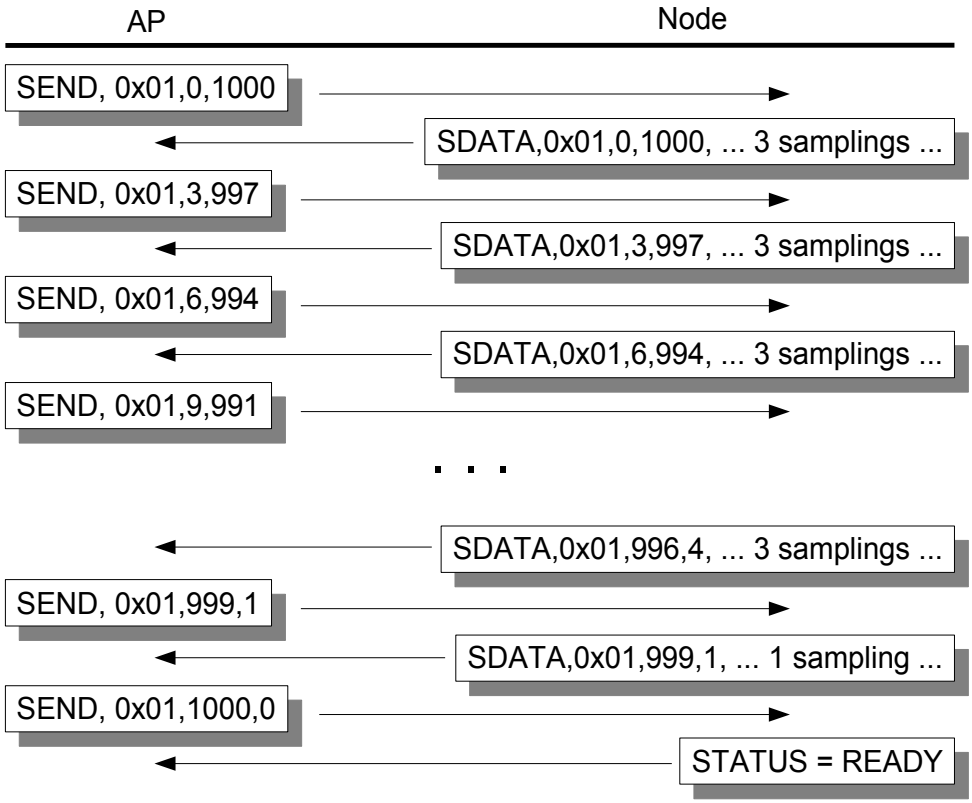


**Figure 6: A sample transmission scenario.**

If while SENDING the node receives a SEND message with *offset* > *NextToAck*, it sets *NextToAck* to *offset* and appropriately shifts the window. This means that the AP does not have to acknowledge every single message.

Having reached the end of window, for as long as *NextToAck* is not advanced, the node will keep retransmitting the last message. If the AP has missed some packets from the window, it should repeat the last SEND message (and keep repeating it until the missed fragment arrives). A duplicate SEND request for the *NextToAck* sampling will be viewed by the node as a request to retransmit all packets starting from *NextToAck*. Note that as soon as the hole (or holes) in the received data have been filled by the AP, it can issue a SEND message whose *offset* will jump to the end of the received continuous set, which will tell the node to abandon the retransmission and move ahead.

The key to the effectiveness of this scheme is in the right selection of intervals between SEND messages and timeouts. Note that the lowest sensible frequency of the SEND messages (viewed as acknowledgments) is one per full window of received data. Any SDATA message received by the AP out of sequence (i.e., leaving a hole in the stream of samplings) should trigger an immediate SEND for the first missed sampling. This SEND should be sent twice and then repeated with some delay until the missed fragment arrives. Then, as soon as the AP finds out that the holes have been filled, it should immediately issue a SEND request specifying the first sampling behind the already received continuous sequence, which will avoid unnecessary retransmissions by the node.