Propeller Programming Using the Digi XBee S6B Wi-Fi Module

Abstract

Since 2006, the Propeller P8X32A microcontroller was typically programmed via a desktop computer host with downloads achieved over a connected serial or USB cable. Recent years have seen expanded host options and wireless download methods. This document details a process of wireless downloading over a Wi-Fi medium using Digi's XBee S6B Wi-Fi Module.

Table of Contents

This remainder of this document is organized into the following sections:

Background

The Propeller contains a ROM-resident boot loader that is responsible for receiving new Propeller Applications and booting up existing applications from an external EEPROM. When receiving new applications from a host computer, this boot loader uses a sophisticated identification process, a specially-encoded payload, and a well-timed transmission protocol to properly receive and validate Propeller Applications.

The key to programming the Propeller is to generate a reset pulse (that reboots the Propeller), then start communication around 100 ms after the transmitted reset pulse, and also deliver the application image as a stream of bytes in the proper order without any large delays in-between bytes. For the Propeller's built-in boot loader, any delay between any two bytes of that stream greater than 100 ms is too much.

This process works well over wired connections to ensure consistent and valid application images, but the inherent timing constraints often interfere with successful delivery over wireless mediums.

The technique described here demonstrates one way to overcome these timing issues using features of the XBee Wi-Fi and attributes of Internet Protocol.

Concept

Wi-Fi is a network medium that transports information using various protocols, including the well-known TCP/IP and UDP/IP protocol combinations. These protocols transmit data in bursts, called packets. Though each packet is transmitted as a complete unit, they are each limited in size, causing most communications to require multiple packets.

Many conditions cause packets to be received out of sequence, arrive corrupted, or be lost in transmission. Conveniently, the receiving side automatically verifies packets and throws away those that are corrupted; however, lost or out-of-sequence conditions need to be handled automatically, by using TCP, or manually, when using UDP.

The XBee Wi-Fi can receive packets up to 1,400 bytes in size. When packets are transmitted to the XBee Wi-Fi's Serial Service, their payloads (data) are stored in a serial output buffer and transmitted serially in a strictly-timed fashion (at a preset baud rate) to the Propeller. The serial output buffer is limited to about 2,000 bytes and empties at a predictable rate according to baud rate and flow control conditions. If a new packet's payload can not completely fit into the serial output buffer's remaining space at the moment it's received, it is thrown away.

When using IP to transmit, the delays between any two packets can be unpredictable and extensive. Even with use of the serial output buffer, delays between bytes that span packet borders can far exceed the 100 ms time limit of the Propeller's built-in boot loader.

All of these factors make for plenty of opportunities to fail, and each must be overcome to ensure a consistent and reliable experience for the Propeller developer.

The solution employed here focuses on the attributes of XBee Wi-Fi and IP communication that are speedy, regular, and fall within the expected behavior of the built-in boot loader. These attributes are exploited at the start of every download process to reliably deliver a small download-compatible application stream. The resulting application then runs immediately to assist with the remainder of the download in an IP-compatible fashion and launches the final target application in its place.

This solution can be summed as the following techniques:

1. Using UDP instead of TCP for more consistent and faster timing.
2. Utilizing XBee's time-to-hold I/O option to generate consistent pulses.
3. Enabling XBee's serial flow-control, preventing early arrival of the serial stream after reset pulse.
4. Filling the first packet with a complete handshake sequence and micro boot loader application.
5. Issuing a second packet containing only timing templates to complete the initial delivery.
6. Transceiving the actual target application with the temporarily-running micro boot loader.

The micro boot loader application stuffed into the first packet is a small, specially-designed boot loader replacement. It launches right after delivery and carries on the remainder of the download communication. The micro boot loader runs at a fast, accurate speed (based on the Propeller's development board crystal), receiving the target application image from the host at swift pace with built-in accommodation for IP traffic delays and packet loss.

From the user's perspective, the result is a quick wireless download that a behaves similar to a cable-based download, with the only caveat being the need for an on-board crystal.

Details

The program and source code written to develop and test this downloading process is provided as an example project archive. You can also find it from the Parallax download page by searching with "Propeller IP Loader" in the *Download Title* field. At this time, it is still a work-in-progress and will continue to be updated as necessary to refine the experience. Check back and download again if you're interested in updates. The test program executable currently runs in Windows (32 or 64-bit) and will likely be expanded to support Mac before development is finished.

For an overview of the source code and development tips, watch this source code overview video.

The following sections give needed background information and details that are needed to fully understand the details of the download process.

Application Service

The XBee Wi-Fi S6B modules feature an "Application Service" communication channel. The Application Service is used to configure the XBee's settings over Wi-Fi and can also be used to send information intended for Wi-Fi-to-Serial transfer. The solution employed in our case uses UDP to engage this Application Service from a network client to do both of these things.

This is what is meant wherever the instructions in the Download Process section say "Send...packet," or when the source code looks like "`XBee.SendUDP`...," or "`XBee.SetItem`...," or "`XBee.GetItem`..."

The Application Service is not the same as the Application Programming Interface (API). Do not confuse the two. The Application Service is documented in Digi's Wi-Fi RF Module manual in the XBee IP Services chapter. Ignore the Local Host section; focus attention on the Network Client section.

Transmitted Packets

UDP packets meant for the Application Service must be sent to port 3054 ($BEE, or cleverly, 0xBEE; i.e.: <xbee_ip_address>:3054) and must include an additional 8-byte header preceding the data. The header is in this format:

Byte #	0	1	2	3	4	5	6	7

Field	**Header_ID**	**Header_ID^**	**n/a**	**n/a**	**Command**	**Options**

The field names shown here and below are renamed for clarity, but their meaning and purpose matches the documented nature in the Network Client section of the wi-Fi RF Module manual.

- **Header_ID** : Two-byte random number; any random number will do.
- **Header_ID^** : Two-byte number equal to Header_ID's value XOR'd with $4242.
- **n/a** : One-byte value; reserved for future use; set to 0 for now.
- **Command** : One-byte purpose indicator; use $00 for Serial Data, or $02 for Remote Command.
- **Options** : One-byte option indicator; use $00 for none, or $02 to request packet acknowledgement.

Using packet acknowledgement is recommended and can be advantageous when using XBee Wi-Fi's UDP-based Application Service.

Additionally, if the Command is $02 (Remote Command), the bytes immediately following this header (bytes 8..11+) must be in this format:

Byte #	8	9	10	11	12+
Field	**Frame_ID**	**Apply**	**AT_Command**		**Params**

- **Frame_ID** : One-byte value; set to 1.
- **Apply** : One-byte action indicator; use 0 to queue command only, or 2 to apply command(s).
- **AT_Command** : Two-byte (two character) AT command
- **Params** : Parameter value(s); this field exists only if the AT_Command has additional parameters.

If the Command is $00, the above format does not apply; rather, serial data should immediately follow the header, starting with byte 8.

Received Packets

Packets received from the Application Service follow a similar format as with transmitted packets.

The leading 8-byte header is a duplicate of the header from the packet to which it is responding to, except that the Command field has its bit 7 set and the Options field is $00. For example, if a packet meant as a remote command ($02) was transmitted, the response packet would have $82 in the Command field. For a response to a serial data packet ($00), the Command field would contain $80.

The bytes immediately following the received header are different, depending on the nature of the response.

For responses to remote commands, the bytes following the header follow this format:

Byte #	8	9	10	11	12+
Field	**Frame_ID**	**AT_Command**		**Status**	**Params**

- **Frame_ID** : One-byte value; same as used in transmitted packet.
- **AT_Command** : Two-byte (two character) AT Command value; same as used in transmitted packet.
- **Status** : One-byte value; 0 = OK, 1 = ERROR, 2 = Invalid Command, 3 = Invalid Parameter.
- **Params** : Zero or more bytes of data, in binary or ASCII format depending on the command. If the command was to set (ie; not requesting a "write"), this field doesn't exist.

For responses to serial data, a packet is returned only if a packet acknowledgement was requested (Option field was $02), in which case only the header exists in the response; the response is only 8 bytes.

The bytes in the examples below are shown in transmitted order, left-to-right.

Example #1

A transmitted request to get (read) the XBee's IP Address (using the "MY" command; IP Network Address) looks like this 11-byte sequence:

Field	Header_ID	Header_ID^	n/a	n/a	Command	Options	Frame_ID	Apply	AT_Command
Value	$81 $14	$C3 $56	$00	$00	$02	$02	$01	$02	$4D $59

The Header_ID / Header_ID^ values are different every time because we choose values randomly.

The received packet acknowledgement looks like this 8-byte sequence:

Field	Header_ID	Header_ID^	n/a	n/a	Command	Options
Value	$EB $1C	$A9 $5E	$00	$00	$80	$00

The Header_ID / Header_ID^ values used by acknowledgements like this one intentionally do not match the transmitted packet.
Strangely, the Command doesn't reflect that of the transmitted packet either; this is normal.

And the received packet response looks like this 16-byte sequence:

Field	Header_ID	Header_ID^	n/a	n/a	Command	Options	Frame_ID	AT_Command	Status	Params
Value	$81 $14	$C3 $56	$00	$00	$82	$02	$01	$4D $59	$00	$C0 $A8 $01 $89

The Header_ID / Header_ID^ values match the transmitted packet that the packet is responding to.
The Command is that of the transmitted packet but with bit 7 set.
A Status of $00 means OK.
The returned sequence in Params is the requested data; in this case $C0, $A8, $01, $89 means the XBee's IP address is 192.168.1.137.

Example #2

A transmitted request to set (write) the XBee's DIO6 Configuration (to set it to RTS flow control (1)) looks like this 13-byte sequence:

Field	Header_ID	Header_ID^	n/a	n/a	Command	Options	Frame_ID	Apply	AT_Command	Params
Value	$B8 $2E	$FA $6C	$00	$00	$02	$02	$01	$02	$44 $36	$01

The Header_ID / Header_ID^ values are different every time because we choose values randomly.
Since this is a set (write) command, the Params field follows; in this case $01 means I/O 6 should be set to perform RTS flow control functionality.

The received packet acknowledgement looks like this 8-byte sequence:

Field	Header_ID	Header_ID^	n/a	n/a	Command	Options
Value	$C7 $48	$85 $0A	$00	$00	$80	$00

The Header_ID / Header_ID^ values used by acknowledgements like this one intentionally do not match the transmitted packet.
Strangely, the Command doesn't reflect that of the transmitted packet either; this is normal.

And the received packet response looks like this 12-byte sequence:

Field	Header_ID	Header_ID^	n/a	n/a	Command	Options	Frame_ID	AT_Command	Status
Value	$B8 $2E	$FA $6C	$00	$00	$82	$02	$01	$44 $36	$00

The Header_ID / Header_ID^ values match the transmitted packet that the packet is responding to.
The Command is that of the transmitted packet but with bit 7 set.
A Status of $00 means OK.
Since this in response to a set (write) command, the Params field does not exist.

Encoding

There are two different data encoding techniques used for this transmission.

Packets 1..2

The first two packets of the transmission (handshake + micro boot loader and timing templates) are encoded using a special data translation that represents bit values (0 and 1) as two different low-bit patterns. In the example source code, the first two packets are generated in their proper encoding by the Main unit's TForm1.GenerateLoaderPacket method.

> The standard Propeller download protocol is designed to use the Propeller's internal R/C oscillator and be compatible with both RS-232-like and TTL/CMOS serial signalling. Since R/C oscillators are known to be inaccurate (very sensitive to voltage, temperature, and process), the protocol relies on a receive-measure-respond mechanism where only low pulses matter (high pulses are ignored) and the Propeller doesn't speak unless spoken to at that moment. The Propeller's built-in boot loader measures the low pulses at specific points in the host computer's transmissions to determine the widths of two pulse types (called 't' and '2t') which themselves represent binary 1 and binary 0 values, respectively. The timing of these measured low-pulse widths is reproduced in the Propeller's responses to the host computer. This frees the Propeller from the need for an accurate clock source and a specific baud rate at startup. The downside is that only a few data bits can be expressed per logical byte of transmission; 3-bits per byte (11 bytes per long) for the normal encoding scheme used by legacy software.

For those familiar with the standard Propeller download encoding, the data in the first packet will appear different due to more optimal encoding. This is just like normal Propeller download image encoding, but with data bits packed as tightly as possible; 3, 4, or 5 bits per byte as opposed to strictly 3 bits per byte. The first packet is encoded very tightly in order to squeeze the handshake and micro boot loader application image into its limited space. For example, normally the handshake sequence is two back-to-back blocks of 250 bytes each, plus an additional 8 bytes to transfer the Propeller version number; 508 bytes all-together. But with optimal encoding, this is squeezed down to just 198 bytes, leaving more space afterwards for the micro boot loader code.

Packets 3..n

The third through the remaining packets are encoded differently. Since they are transferred after the micro boot loader starts, and since it runs based on an accurate clock source, the data is encoded in normal fashion (bit for bit; 8-bits per byte; 4 bytes per long). This amounts to a 2.75x speed increase for target application transfer and easier handling on both sides of the stream. From the developer's perspective, this data is easy to prepare; it can simply be copied as-is from the binary image generated by the compiler since it's already in the correct format and byte order.

Micro Boot Loader

The source code of the micro boot loader (called IP_Loader.spin) is included with the download noted at the top of the Details section. It is written in Propeller Assembly, except for the single Spin statement to get it to launch. Once launched, Main RAM is completely free to be rewritten with the target application image that the micro boot loader receives.

Unfortunately, even with clever programming techniques, there's not enough space in the first packet to fit a complete loader; that is, one that can receive, acknowledge, program RAM, program EEPROM, verify, finalize, and launch target applications.

Instead, the micro boot loader is actually delivered in parts, with the first part (first packet) being the core (which handles reception, acknowledgement, and programming of RAM) and the remaining features are delivered only as-needed in special executable packets that follow the target application's packets.

At this time, the program EEPROM feature is not implemented. Appropriate modifications will be made soon to add this feature and updated downloads and documentation will be released at that time.

This code is written in a very tight manner, opting for careful balance of code size, timing, and protocol efficiency. For those up to the challenge, this loader can be modified or replaced with new code using similar techniques to implement even more advanced loaders.

The executable images of each of the parts of this micro boot loader are included in the example application's Main unit, inside the TForm1.GenerateLoaderPacket method. The GenerateLoaderPacket method's job is to create each of these special packets when requested; automatically performing all the necessary touch-up and encoding. Read the many comments within this method to learn more details.

The process used to get the executable images into the method is:

- Compile and save the micro boot loader (IP_Loader.spin) as a binary file using Propeller Tool's "View Info," "Save Binary File" feature.
- Run the PropellerStream.exe program (included in the download archive).
- Click the "Load Propeller Application" button and select the saved binary file.
- Copy the desired sections of code generated by the previous step and paste them in place of their related sections in the GenerateLoaderPacket method.

Transmission Logistics

Some care needs to be taken to transmit each packet to its destination. As said earlier, many conditions cause packet interruption. Luckily, UDP automatically handles corruption by dropping the affected packets, but we must handle retransmission due to dropped packets, dropping of duplication packets, and out-of-order packets.

There are two phases of transmission involved here.

- Phase 1: <u>Before</u> the micro boot loader is loaded and running.
- Phase 2: <u>After</u> the micro boot loader is loaded and running.

Phase 1

During this phase, the host computer transmits packets and expects a response in the form of an Application Service packet acknowledgement and/or a second response in the form of a command response. If it doesn't receive the expected response after a short timeout, it retransmits the packet again. This happens a limited number of times; 3 times max, for example. If confirmation is ultimately not obtained, the host gives up with a hard error condition.

This is happening by design in the example code implemented in the calls like "`XBee.SendUDP`...," or "`XBee.SetItem`...," or "`XBee.GetItem`..."

Phase 2

Once the micro boot loader is loaded and running, our task focuses on delivering packets in a slightly different fashion. Though it may still use the Application Service's packet acknowledgement feature, the automatic retransmission handled by the lower-level code (references like "`XBee.SendUDP`") is disabled.

Instead, the host follows this process:

- Host: transmits a packet and expects a positive or negative acknowledgement in response.
    - No acknowledgement is treated like a negative acknowledgement.
- Host: transmits the next packet only after receiving positive acknowledgement of the previous.
- Host: retransmits the previous packet when it sees a negative acknowledgement. This is a soft error condition.
    - This happens a limited number of times; 3 times max, for example. If confirmation is ultimately not obtained, the host gives up with a hard error condition.

Meanwhile, the running micro boot loader follows this process:

- Micro boot loader: receives a packet and transmits a positive or negative acknowledgement
    - Positive is when it expected the received packet, negative is otherwise.
- Micro boot loader: writes the expected packet to RAM and discards the unexpected packet

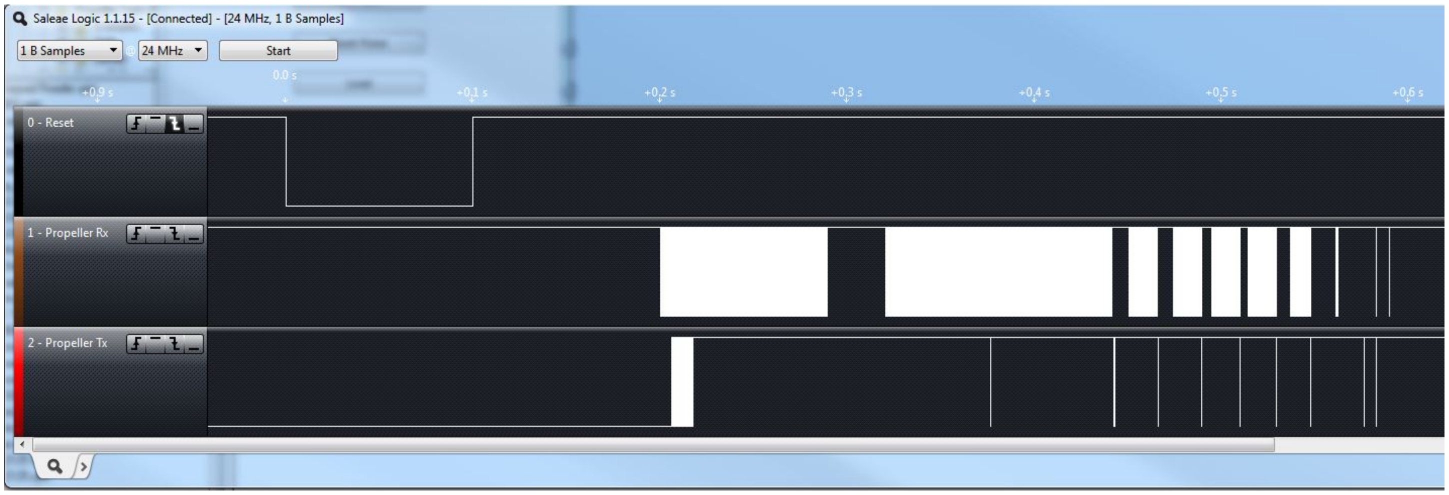
The host's part of this process is handled by the code detailed by the steps in the Download Process.

Download Process

The numbered steps below describe the major points of the wireless download process, with subitems giving more details for that point. For a summary view, just read the numbered steps.

The example program's source code (written in Delphi; a Pascal variant) is referred to in the steps below with references to the containing unit and method names *[in brackets]*.

Many of the major points relate directly to the logic analyzer capture below– a zoomed-out view of a sample application download from the Propeller's perspective. Annotated versions of this image will appear below the major points that drive them.



Some non-essential details, such as implementation choices and certain error handling, are intentionally left out of the description below. Study the example code to see everything in detail.

Points to remember:

- For every wireless download, there are actually two applications delivered– a micro boot loader (first packet + second verification packet), and the target application (third and later packets + remaining verification packets).
- The first packet and final packets are created by the TForm1.GenerateLoaderPacket method. This is explained in the Encoding and Micro Boot Loader sections, above.
- Transmissions take place over UDP/IP and target the XBee's Application Service, explained in the Application Service section, above.
- Retransmissions are handled as noted in the Transmission Logistics section, above.

The download process is as follows:

1. Calculate total packet count for the target application
    a. Equation: binary image size (in bytes) / (max packet size - packet header)
        i. max packet size is likely 1392 bytes (obtained from XBee's NP attribute)
        ii. packet header is 4 bytes
        iii. round up the result to include last partial packet.
    b. Code: *[Main.pas - TForm1.TransmitButtonClick]*
        ```
        TotalPackets := Round(FBinSize*4 / (XBee.MaxDataSize-4*1));
        ```
2. Set initial Packet ID to the total packet count
    a. Note: Packet ID is needed for generating the micro boot loader and for each packet that follows delivery of the micro boot loader. The first long (4 bytes) of each packet contains a unique packet identifier. The first such packet is numbered (identified) with the total packet count of the target application. The Packet ID value decrements for each successive packet.
3. Calculate target application checksum as a long value
    a. Note: This value is needed later for target application verification.
    b. Algorithm: Simple additive checksum.
        - Clear checksum (long; 4 bytes, unsigned)
        - Iteratively add the value of each target application image byte to checksum
        - Iteratively add the value of each byte of the Initial Call Frame (eight bytes = $FF, $FF, $F9, $FF, $FF, $FF, $F9, $FF) to checksum
            - Note: This is a static sequence whose checksum could be precalculated
    c. Code: *[Main.pas - TForm1.TransmitButtonClick]*
        ```
        Checksum := 0;
        for i := 0 to FBinSize*4-1 do inc(Checksum, FBinImage[i]);
        for i := 0 to high(InitCallFrame) do inc(Checksum, InitCallFrame[i]);
        ```
4. Open UDP socket to XBee's Serial Service
    a. Note: Abort on error
    b. Destination: XBee's IP address and serial service port; defined by DE attribute; typically 9750 ($2616)
    c. Code: *[Main.pas - TForm1.TransmitButtonClick]*
        ```
        if not XBee.ConnectSerialUDP then...
        ```
5. Generate Loader Packet
    a. Note: This is the initial packet that contains a compressed Propeller handshake stream plus micro boot loader, and more. Before insertion into the packet, the micro boot loader image is

"touched up" to set its timing parameters and it's initial expected Packet ID, then it's checksum is recalculated and the entire resulting image is translated into compressed standard Propeller download encoding. See Encoding and Micro Boot Loader.

b. Code: *[Main.pas - TForm1.TransmitButtonClick]*

```
GenerateLoaderPacket(ltCore, TotalPackets);
```

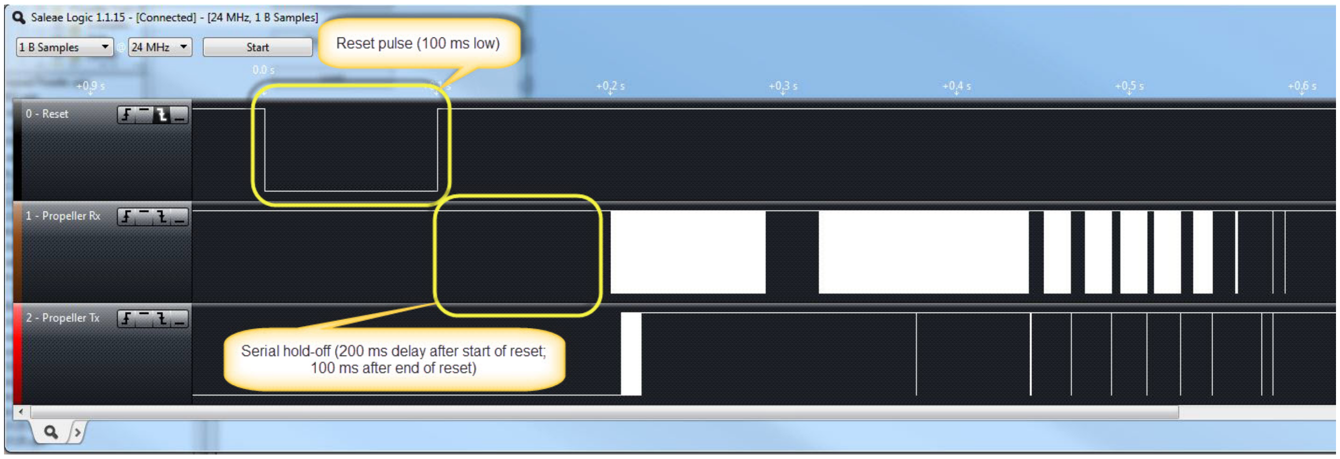
6. Set XBee Configuration and Generate Reset and Serial Hold Signal
   a. Algorithm: Set configuration first…
      i. Set XBee's Serial Service to use UDP packets (IP = $00)
         1. *Note: This step is desired, but at this time, an XBee Wi-Fi firmware bug makes this dangerous. Until it's fixed, do not set this programmatically, it must be set manually via XCTU. Digi intends to fix this bug soon.*
      ii. Set Serial-to-IP destination to host IP address (DL = ip_address)
      iii. Set output mask to default (OM = $7FFF)
      iv. Enable RTS flow control pin (D6 = $01)
      v. Set serial hold pin to output low (D4 = $04)
      vi. Set reset pin to output high (D2 = $05)
      vii. Set serial hold pin's timer to 200 ms (T4 = 2)
      viii. Set reset pin's timer to 100 ms (T2 = 1)
      ix. Set Serial Mode to transparent (AP = $00)
         1. *Note: This step is desired, but at this time, an XBee Wi-Fi firmware bug makes this dangerous Until it's fixed, do not set this programmatically, it must be set manually via XCTU. Digi intends to fix this bug soon.*
      x. Set baud rate to initial speed (BD = 115200)
      xi. Set parity to none (NB = $00)
      xii. Set stop bits to one (SB = 0)
      xiii. Set packetization timeout to three character times (R0 = 3)
   b. Algorithm: ...then set reset pin low and serial hold pin high (IO = $0010). Abort on error.
   c. Note: Pulse and hold timing is controlled by XBee's pin timers (set in configuration step).
   d. Code: *[Main.pas - TForm1.TransmitButtonClick]*

```
GenerateResetSignal;
```

   e. Code: *[Main.pas - TForm1.GenerateResetSignal]*

```
if EnforceXBeeConfiguration then
  if not XBee.SetItem(xbOutputState, $0010) then
    raise Exception.Create('Error Generating Reset Signal');
```

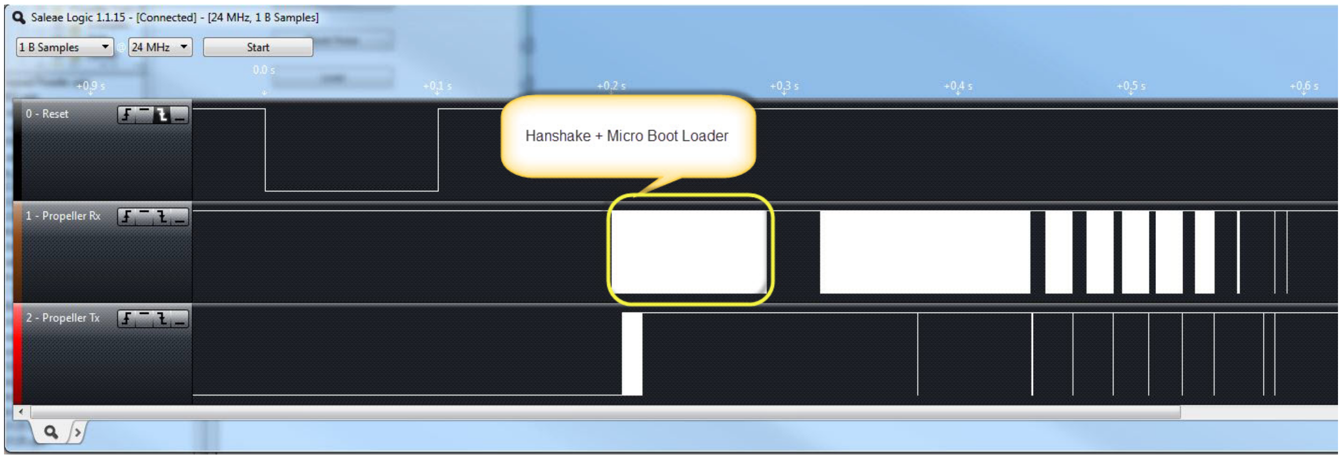
   f. Code: *[Main.pas - TForm1.EnforceXBeeConfiguration]*


Saleae Logic 1.1.15 - [Connected] - [24 MHz, 1 B Samples]
1 B Samples
24 MHz
Start
Reset pulse (100 ms low)
0 - Reset
1 - Propeller Rx
2 - Propeller Tx
Serial hold-off (200 ms delay after start of reset; 100 ms after end of reset)

* Serial hold pin signal is not shown here, but the effects are seen in the delay of the serial stream on the Propeller's Rx pin.

7. Send first packet (handshake + micro boot loader)
    a. This happens immediately after sending the reset and hold command packet, very close to the 0.0s mark (start of reset pulse) on the image below, but the serial hold-off pulse prevents serial data from leaving the XBee until 200 ms after start of reset.
    b. Note: Abort on error
    c. Code: *[Main.pas - TForm1.TransmitButtonClick]*

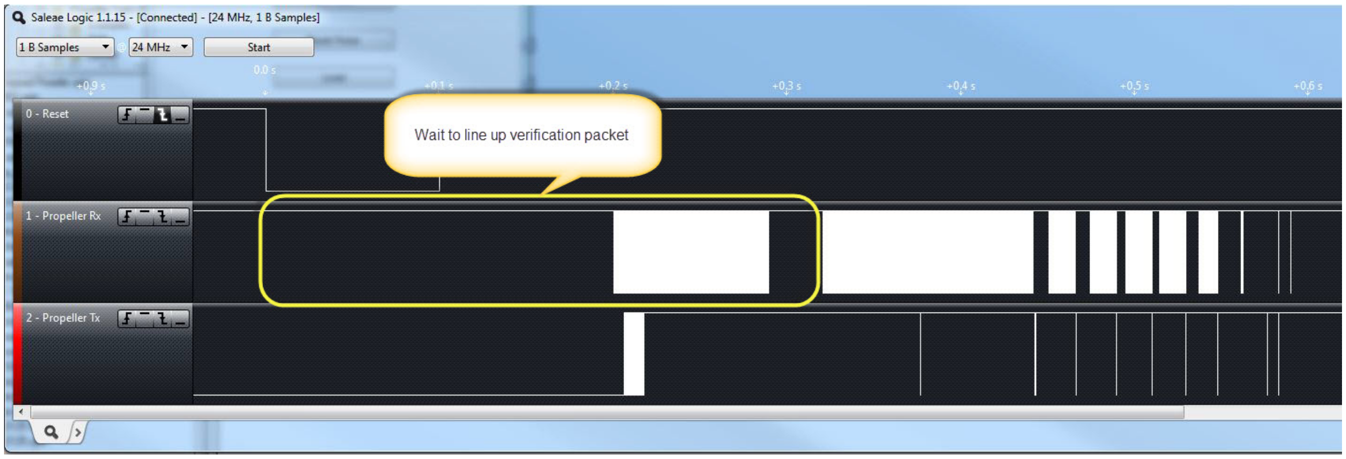
```
if not XBee.SendUDP(TxBuf, True, False) then
  raise EHardDownload.Create('Error: Can not send connection request!');
```



* The Propeller's handshake response (Propeller Tx signal) is generated concurrent with reception of host's Handshake + Micro Boot Loader packet (Propeller Rx signal) but still needs to transmit back through the network. We'll look for it later in the process.

8. Wait long enough to line up verification packet
    a. Period: Reset period (200 ms) + first packet's serial transfer time + 20 ms
    b. Code: *[Main.pas - TForm1.TransmitButtonClick]*

```
IndySleep(200 + Trunc(Length(TxBuf)*10 / InitialBaud * 1000) + 20);
```

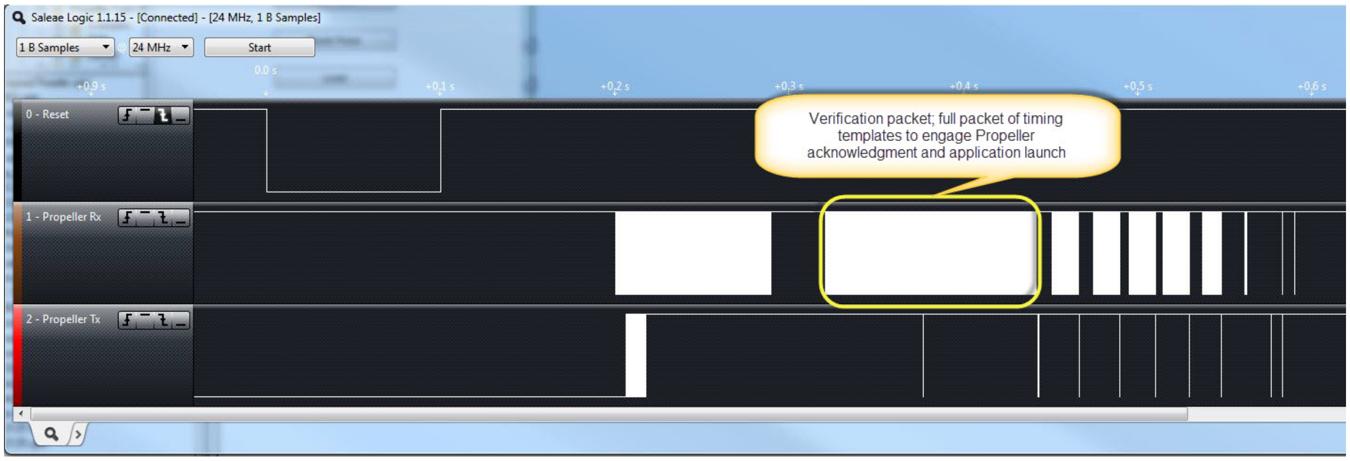


* The built-in booter is ready to deliver the RAM checksum response during a 100 ms window starting as early as 52 ms, and as late as about 163 ms, after the end of the first packet, depending on voltage, temperature, and process conditions. However, it needs to receive a timing template for it to respond. The waiting period of this step is meant to roughly align a full packet of timing templates (in the next step) across this time window.

9. Build and send second packet (verification), then wait for serial transmission period
    a. Note: This is a full packet of just the timing templates (bytes of value $F9) needed by the built-in boot loader to complete it's application verification and launch process. The delay afterward helps apply the majority of the next step's receive timeout to a valid time window in the communication sequence.

b. Code: *[Main.pas - TForm1.TransmitButtonClick]*

```
SetLength(TxBuf, XBee.MaxDataSize);
FillChar(TxBuf[0], XBee.MaxDataSize, $F9);
if not XBee.SendUDP(TxBuf, True, False) then
  raise EHardDownload.Create('Error: Can not request connection response!');
IndySleep(Trunc(Length(TxBuf)*10 / InitialBaud * 1000));
```


Saleae Logic 1.1.15 - [Connected] - [24 MHz, 1 B Samples]
1 B Samples
24 MHz
Start
0 - Reset
1 - Propeller Rx
2 - Propeller Tx
Verification packet; full packet of timing templates to engage Propeller acknowledgment and application launch

* The Propeller's RAM checksum response (Propeller Tx signal) is generated concurrent with reception of host's verification packet (Propeller Rx signal), but still needs to transmit back through the network. We'll look for it later in the process.

10. Receive handshake plus version response; loop to discard any leading garbage.
    a. Note: Discarding of leading garbage is necessary in case previous Propeller application was serially transmitting before it was reset.
    b. Algorithm: Receive packet of 129 bytes.
       - First 125 bytes must match the expected RxHandshake stream.
       - The last 4 bytes contain the 8-bit Propeller version in the following form (where 'x' bits should be ignored and numbered bits ('0', '1', etc.) indicate bit position of the corresponding 8-bit version value.

```
-byte 1-  -byte 2-  -byte 3-  -byte 4-
xx1xxxx0  xx3xxxx2  xx5xxxx4  xx7xxxx6
```

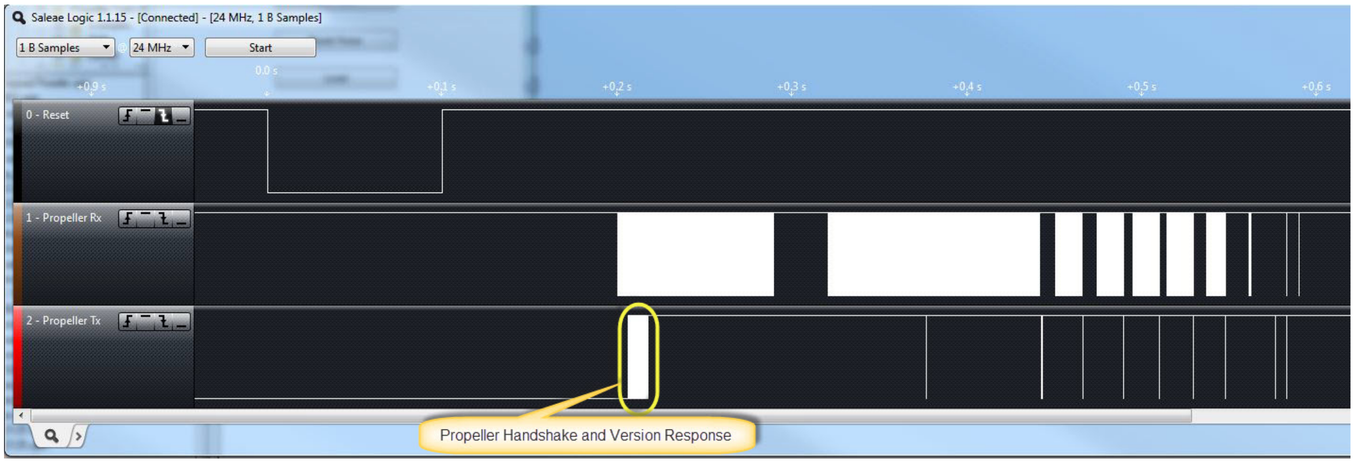
    c. Code: *[Main.pas - global const section]*

```
SerTimeout = 1000;
...
{The RxHandshake array consists of 125 bytes encoded to represent the expected
250-bit (125-byte @ 2 bits/byte) response of continuing-LFSR stream bits from
the Propeller, prompted by the timing templates following the TxHandshake
stream.}
RxHandshake : array[0..124] of byte =
             ($EE,$CE,$CE,$CF,$EF,$CF,$EE,$EF,$CF,$CF,$EF,$EF,$CF,$CE,$EF,$CF,
              $EE,$EE,$CE,$EE,$EF,$CF,$CE,$EE,$CE,$CF,$EE,$EE,$EF,$CF,$EE,$CE,
              $EE,$CE,$EE,$CF,$EF,$EE,$EF,$CE,$EE,$EE,$CF,$EE,$CF,$EE,$EE,$CF,
              $EF,$CE,$CF,$EE,$EF,$EE,$EE,$EE,$EE,$EF,$EE,$CF,$CF,$EF,$EE,$CE,
              $EF,$EF,$EF,$EF,$CE,$EF,$EE,$EF,$CF,$EF,$CF,$CF,$CE,$CE,$CE,$CF,
              $CF,$EF,$CE,$EE,$CF,$EE,$EF,$CE,$CE,$CE,$EF,$EF,$CF,$CF,$EE,$EE,
              $EE,$CE,$CF,$CE,$CE,$CF,$CE,$EE,$EF,$EE,$EF,$EF,$CF,$EF,$CE,$CE,
              $EF,$CE,$EE,$CE,$EF,$CE,$CE,$EE,$CF,$CF,$CE,$CF,$CF);
```

    d. Code: *[Main.pas - TForm1.TransmitButtonClick]*

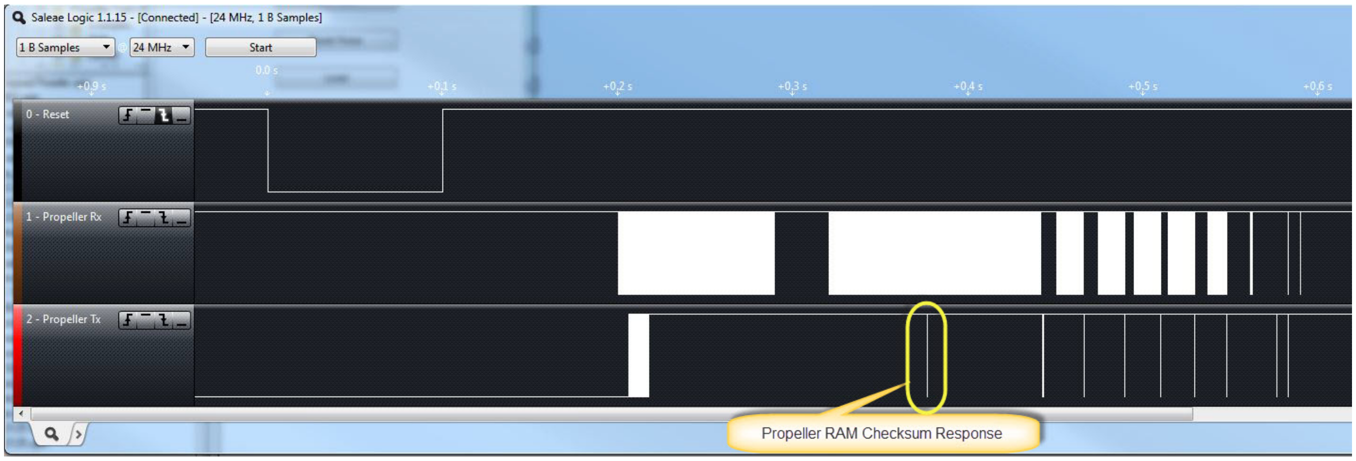
```
repeat {Flush receive buffer and get handshake response}
  {Receive response}
  if not XBee.ReceiveUDP(RxBuf, SerTimeout) then
```

```
      raise ESoftDownload.Create('Error: No connection response from
      Propeller!');

    {Validate response}
    if Length(RxBuf) = 129 then
      begin
      {Validate handshake response}
      for i := 0 to 124 do if RxBuf[i] <> RxHandshake[i] then
        raise EHardDownload.Create('Error: Unrecognized response - not a
        Propeller?');

      {Parse hardware version}
      for i := 125 to 128 do FVersion := (FVersion shr 2 and $3F) or ((RxBuf[i]
      and $1) shl 6) or ((RxBuf[i] and $20) shl 2);

      if FVersion <> 1 then
        raise EHardDownload.Create('Error: Expected Propeller v1, but found
        Propeller v' + FVersion.ToString); {Validate hardware version}

      end;

  {Loop if not correct (to flush receive buffer of previous data)}
  until Length(RxBuf) = 129;
```


Saleae Logic 1.1.15 - [Connected] - [24 MHz, 1 B Samples]
1 B Samples
24 MHz
Start
0 - Reset
1 - Propeller Rx
2 - Propeller Tx
Propeller Handshake and Version Response

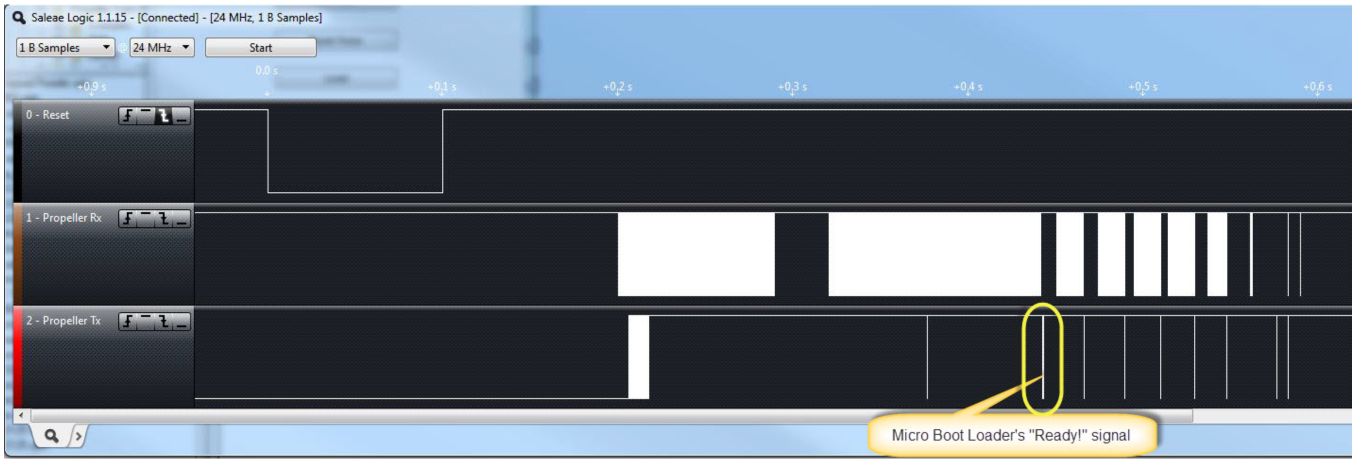
11. Receive RAM checksum response
    a. Algorithm: Receive packet of 1 byte, equal to $FE. Abort if different length response; error, no loader checksum response. Abort if different result; error, failed to deliver loader.
    b. Code: *[Main.pas - TForm1.TransmitButtonClick]*

```
{Receive Loader RAM Checksum Response}
if not XBee.ReceiveUDP(RxBuf, DynamicSerTimeout) or (Length(RxBuf) <> 1) then
  raise ESoftDownload.Create('Error: No loader checksum response!');
if RxBuf[0] <> $FE then
  raise EHardDownload.Create('Error: Failed to deliver loader');
```


Saleae Logic 1.1.15 - [Connected] - [24 MHz, 1 B Samples]
1 B Samples
24 MHz
Start
0 - Reset
1 - Propeller Rx
2 - Propeller Tx
Propeller RAM Checksum Response

12. Receive micro boot loader’s “ready” signal
    a. Algorithm: Receive packet of 4 bytes. Abort if different length response; error, no “ready” signal from loader. Abort if response not equal to initial Packet ID (total packet count); error, loader’s ready signal unrecognized.
    b. Code: *[Main.pas - TForm1.TransmitButtonClick]*

```
Acknowledged := XBee.ReceiveUDP(RxBuf, DynamicSerTimeout);
if not Acknowledged or (Length(RxBuf) <> 4) then
  raise ESoftDownload.Create('Error: No "Ready" signal from loader!');
if Cardinal(RxBuf[0]) <> PacketID then
  raise EHardDownload.Create('Error: Loader''s "Ready" signal unrecognized!');
```


Saleae Logic 1.1.15 - [Connected] - [24 MHz, 1 B Samples]
1 B Samples
24 MHz
Start
0 - Reset
1 - Propeller Rx
2 - Propeller Tx
Micro Boot Loader's "Ready!" signal

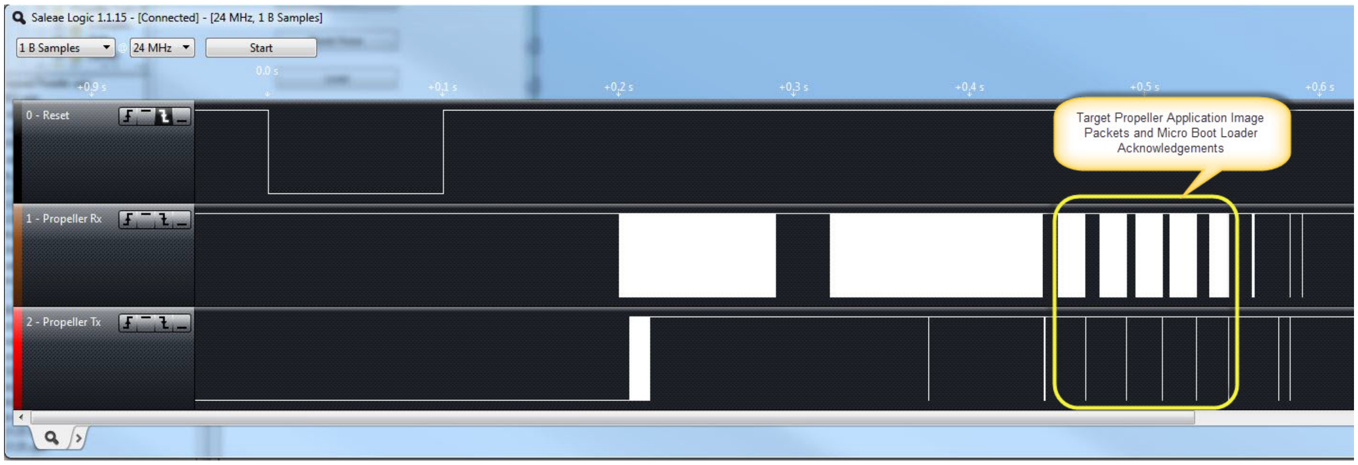
13. Switch to final baud rate
    a. Note: final baud rate must be determined first, before generating the loader packet. The default designed final baud rate is 921600 bps; based on the Propeller running at 80 MHz using a 5 MHz crystal and a PLL tap of 16x.
    b. Set baud rate to final speed (BD = 921600)
    c. Code: *[Main.pas - TForm1.TransmitButtonClick]*

```
if not XBee.SetItem(xbSerialBaud, FinalBaud) then
  raise EHardDownload.Create('Error: Unable to increase connection speed!');
```

14. Transmit all target application packets
    a. Algorithm: Determine packet length (header + lessor of packet limit or remaining image length). Set first long to Packet ID, followed by next section of image. Transmit packet (retransmit as necessary; see Transmission Logistics) Abort if unexpected response. Decrement Packet ID and prep for next section of image.

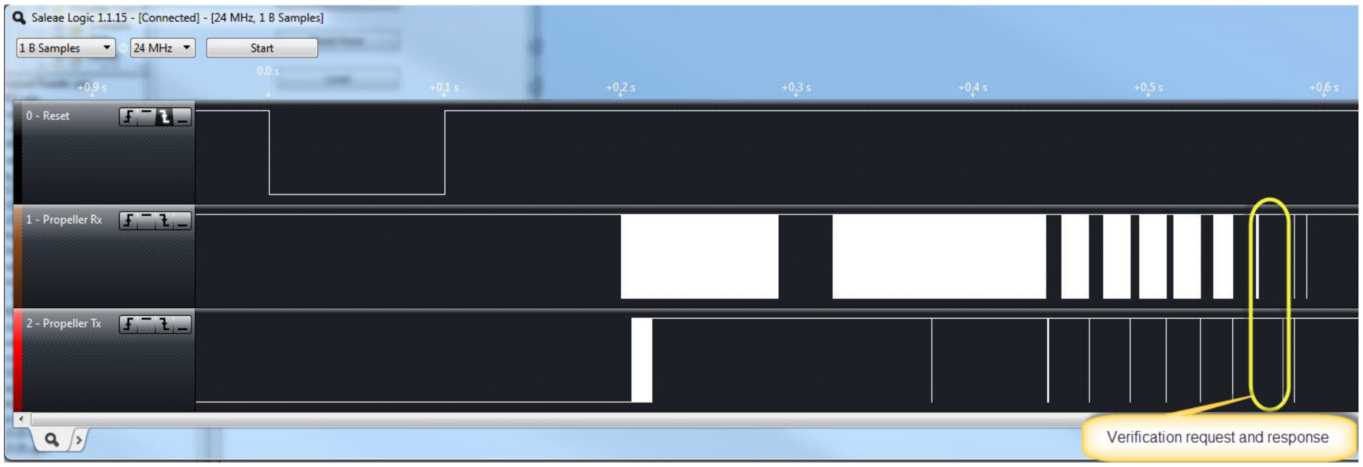
b. Code: *[Main.pas - TForm1.TransmitButtonClick]*

```
{Transmit packetized target application}
i := 0;
repeat
  {Determine packet length (in longs); header + packet limit or remaining data
  length}
  TxBuffLength := 1 + Min((XBee.MaxDataSize div 4)-1, FBinSize - i);
  {  Set buffer length (Packet Length) (in bytes)}
  SetLength(TxBuf, TxBuffLength*4);
  Move(PacketID, TxBuf[0], 4);
  Move(FBinImage[i*4], TxBuf[4], (TxBuffLength-1)*4);
  if TransmitPacket <> PacketID-1 then
    raise EHardDownload.Create('Error: communication failed!');
  {Increment image index}
  inc(i, TxBuffLength-1);
  {Decrement Packet ID (to next packet)}
  dec(PacketID);
  {repeat - Transmit target application packets...}
until PacketID = 0;
```


Saleae Logic 1.1.15 - [Connected] - [24 MHz, 1 B Samples]
1 B Samples
24 MHz
Start
0 - Reset
1 - Propeller Rx
2 - Propeller Tx
Target Propeller Application Image Packets and Micro Boot Loader Acknowledgements

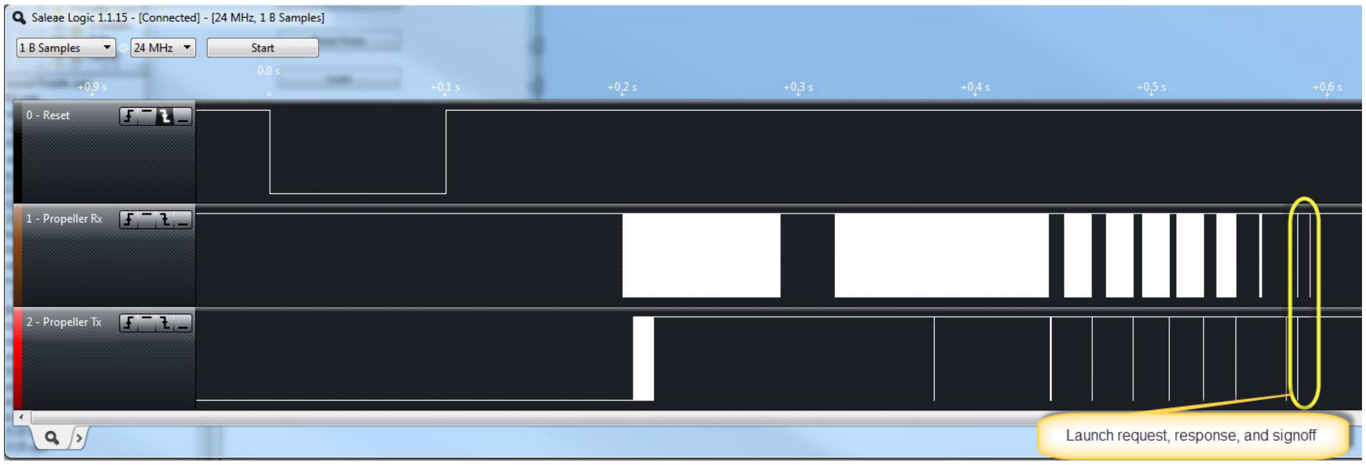
15. Send verify RAM command and receive response
    a. Algorithm: Generate verify RAM packet and transmit it. Abort if response not equal to negative of target application checksum; error, RAM checksum failure.
    b. Code: *[Main.pas - TForm1.TransmitButtonClick]*

```
{Send verify RAM command}
GenerateLoaderPacket(ltVerifyRAM, PacketID);
if TransmitPacket <> -Checksum then
  raise EHardDownload.Create('Error: RAM Checksum Failure!');
```


Saleae Logic 1.1.15 - [Connected] - [24 MHz, 1 B Samples]
1 B Samples
24 MHz
Start
0 - Reset
1 - Propeller Rx
2 - Propeller Tx
Verification request and response

16. Send launch command and receive response
    a. Code: *[Main.pas - TForm1.TransmitButtonClick]*

```
PacketID := -Checksum;
{Send verified/launch command}
GenerateLoaderPacket(ltLaunchStart, PacketID);
if TransmitPacket <> PacketID-1 then
  raise EHardDownload.Create('Error: communication failed!');
dec(PacketID);
{Send launch command}
GenerateLoaderPacket(ltLaunchFinal, PacketID);
XBee.SendUDP(TxBuf, True, False);
```


Saleae Logic 1.1.15 - [Connected] - [24 MHz, 1 B Samples]
1 B Samples
24 MHz
Start
0 - Reset
1 - Propeller Rx
2 - Propeller Tx
Launch request, response, and signoff