Propeller Programming Protocol

Software developers use the Propeller Programming Protocol to build downloader systems that can deliver compiled applications to an embedded Propeller P8X32A microcontroller. This document describes the protocol in detail, covering basic principles, timing constraints, and specific implementation for different transmission mediums.

NOTICE: This document is still a work-in-progress. All items currently documented are accurate; however, there are many enhancements and tips still being developed for inclusion here.

Table of Contents

Programming Interface (Electrical Connections)

The Propeller P8X32A is in-circuit programmable via a simple electrical interface. Programming can be done by a host device (PC, Mac, microcontroller, etc.) without the need for any special programmer device. In its simplest form, the programming interface may be only 4-wires. More sophisticated communication interfaces (like RS-232 or USB-to-Serial) may be used, but will require extra circuitry and data manipulation for proper electrical and protocol conversion.

Programming Interface (Protocol)

The Propeller Programming Protocol (referred to as "protocol" from here on) is a variable bit-size, flexible bit-rate, asynchronous serial communication mechanism that functions even with inaccurate clock sources. It is designed for communicating through direct-connect digital circuitry (simple wires) and also through RS-232 devices. The protocol is described here in the following ways:

- General Principles - attributes that apply to all forms of the protocol
- Protocol Proper - its pure form, compatible with digital I/O pins
- Protocol RS-232 - compatible with UART and USB Virtual COM Port based serial mediums

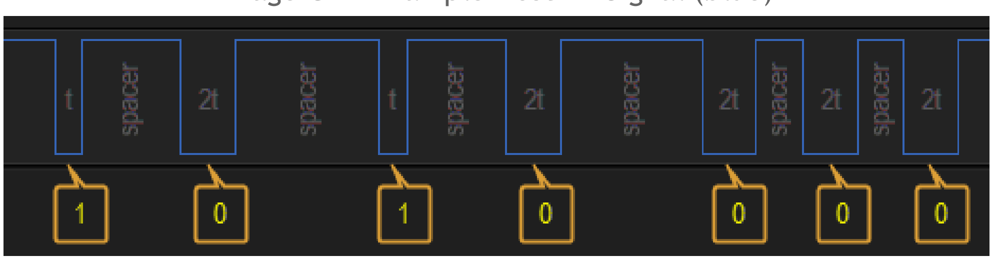
Developers should deploy the form most appropriate for an application's needs. However, it is recommended that they study General Principles and Protocol Proper in order to fully understand other variations.

General Principles

Protocol assumes two participants; a Propeller to be programmed and a “host” that initiates contact and serves the new program content. To maximize throughput, communication consists mostly of host transmissions and very few Propeller transmissions (host receptions). Throughout the transaction, the Propeller doesn’t speak unless spoken to, even during a response phase of the protocol.

Data bits are transferred as low pulses of two different widths, with values of 1 and 0 being communicated as low pulses of time t and 2t (twice the width of t), respectively. The width of t is flexible as long as it remains consistent within the given serial stream. High pulses serve only as data bit spacers (separators) whose widths are unimportant. The high pulse between any two low pulses must not exceed 100 ms or the Propeller will give up and move on, ignoring any further communication.

Image G1 - Example Host Tx Signal (blue)



Only low pulses matter; t-width pulse means binary 1, 2t-width pulse means binary 0.
High pulses vary in width and are interpreted as spacers between data bits.

The first half of the protocol always communicates the same information (calibration, handshake, connection, and version) and the second half communicates differing information (command, application size, application image, and acknowledgments).

Calibration

At strategic points in the communication, the host transmits calibration pulses; conveying the two bit values (1 and 0) as their t-sized and 2t-sized low pulses. The Propeller expects and measures these calibration pulses in order to reliably read, and occasionally respond to, the transmission.

Handshake and Connection

The handshake and connection streams are specific patterns of bits used to validate the host and the Propeller. See Appendix A - Handshake and Connection for detailed information on this pattern.

Version

The Propeller version is an 8-bit value transmitted by the Propeller LSB-First; value of 1 = P8X32A.

Command

The protocol command is issued by the host to indicate the intent of the communication.

Table G1: Protocol Commands (32-bits each)

Name	Value	Definition
Shutdown (aka Identify)	$00000000	Terminate all cogs until next reset/power-up. This command is issued immediately after receiving the Propeller's Version value if identification was the only intent.
LoadRun	$00000001	Load application into RAM, then execute application.
ProgramShutdown	$00000002	Load application into RAM, program into EEPROM, then terminate all cogs until next reset/power-up. This command is usually not implemented by hosts.
ProgramRun	$00000003	Load application into RAM, program into EEPROM, then execute application.

After version is received, the host sends a command (a 32-bit value from this table) telling the Propeller what it intends to do next.

Protocol Proper

Protocol Proper is the pure form of the protocol designed for communication over simple direct-connect digital circuitry. The Spin object PropellerLoader.spin implements Protocol Proper between one Propeller (acting as the host) and another Propeller (the one being programmed). Developers knowledgeable in Spin can gain insight into Protocol Proper by reading this code example; see Appendix B - PropellerLoader.spin.

Metrics

- **Reset Pulse** - Drive low > 10 µs, then release (hi-z)
- **Post-Reset Delay** - 60 ms to 210 ms (90 ms to 100 ms is recommended)
- **Data Bit Order** - LSB-First
- **Low Pulse (t)** - Represents bit value 1
- **Low Pulse (2t)** - Represents bit value 0
- **Low Pulse (t) Width** - 4.3 µs to 26 µs (recommend ≅ 8.6 µs)
- **High Pulse Width** - 4.3 µs to 90 ms (recommend ≥ t µs)

Protocol Proper Algorithm

The following describes the steps for Protocol Proper from the perspective of the host. The images afterwards contain additional vital information using actual annotated examples.

When implementing this protocol from scratch, try attempting only an identification operation (the first half of the algorithm) then the final steps (in the second half) will generally be easier to implement.

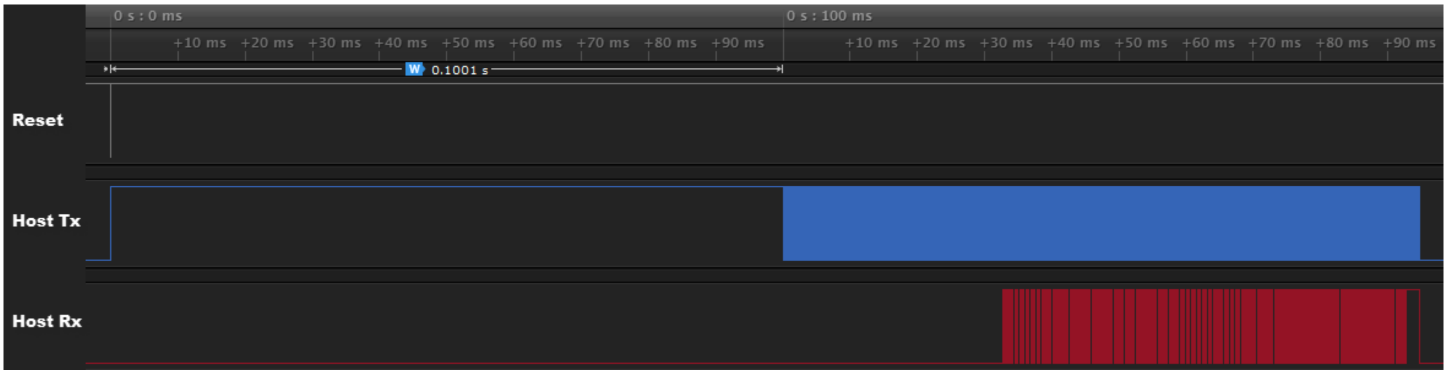
- **Reset the Propeller** [Image P1, Image P2]
  - Drive RESn low > 10 µs, set Tx high / Rx input, then release RESn (hi-z)
- **Wait for Post-Reset Delay** [Image P1, Image P2]
  - 90 to 100 ms recommended
- **Transmit Calibration Pulses** [Image P3]
  - 2-bits (%01) LSB-First
- **Transmit Handshake Pattern** [Image P4]
  - 250-bits; see Appendix A - Handshake and Connection
- **Receive Connection Response** [Image P5]
  - 250 iterations of: transmit 2-bit Calibration Pulse, receive 1-bit Connection Response Bit; see Appendix A
  - If Propeller does not respond to any Calibration Pulse pair, or Connection Response bit is invalid, abort communication; connection error
- **Receive Version** [Image P6]
  - 8 iterations of: transmit 2-bit Calibration Pulse, receive 1-bit Version Pulse
  - If Propeller does not respond to any Calibration Pulse pair, or if Version is bad, transmit Shutdown Command, abort communication; version error
  - Valid Propeller Version = 1 (8-bit value, received LSB-first)
- **Transmit Command** [Image P7, Image P8 - Image P9]
  - 32-bits; often Shutdown (ID only), LoadRun (RAM only) or ProgramRun (RAM & EEPROM); see Table G1
  - If transmitted Command is Shutdown (aka Identify), success; terminate communication
- **Transmit Size of Application in longs** [Image P9]
  - 32-bits; this value should come from word 9:8 of the application image, shifted right by 2 bits
- **Transmit Application Image** [Image P10]
  - Stream of 8-bit bytes
- **Wait for Acknowledge (RAM Checksum)** [Image P11]
  - Up to 250 ms of: transmit 2-bit Calibration Pulse, check for 1-bit Ack (0) or Nak (1)
  - If no response before timeout or Negative Acknowledge (Nak) received, abort communication; transmission error or RAM verify error, respectively
- **If Command is LoadRun** [Image P11]
  - Success; terminate communication
- **Else, Command is ProgramShutdown or ProgramRun** [Image P12]
  - **Wait for EEPROM Program** [Image P13]
    - Up to 5 s of: transmit 2-bit Calibration Pulse, check for 1-bit Ack (0) or Nak (1)
    - If no response before timeout or Negative Acknowledge (Nak) received, abort communication; EEPROM program error
  - **Wait for EEPROM Verify** [Image P13]
    - Up to 2 s of: transmit 2-bit Calibration Pulse, check for 1-bit Ack (0) or Nak (1))
    - If no response before timeout or Negative Acknowledge (Nak) received, abort communication; EEPROM verify error
  - **Success; terminate communication**

The following images are actual logic analyzer captures of Protocol Proper. The example system consists of a “host” Propeller, running PropellerLoader.spin, that is connected to a Propeller to-be-programmed.

Identification Sequence

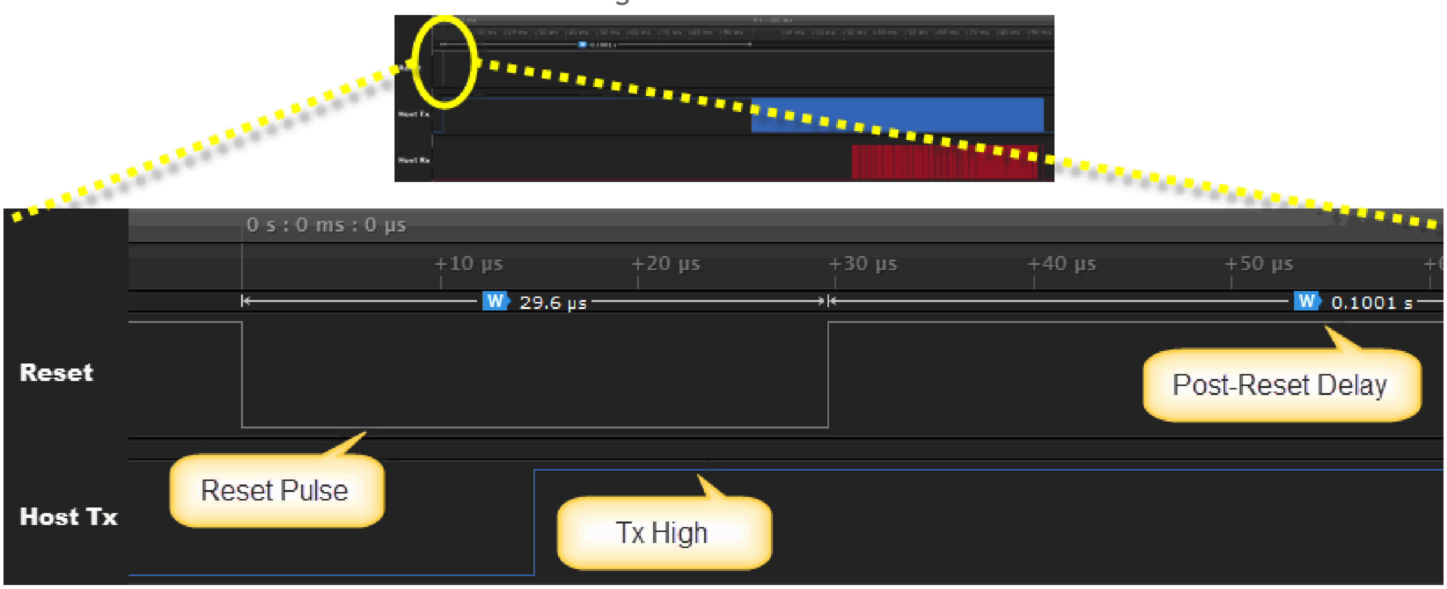
The first set of images demonstrates the identification process (Reset through Command) which covers the first half of the Propeller Proper Algorithm.

Image P1 - Full Identification Sequence



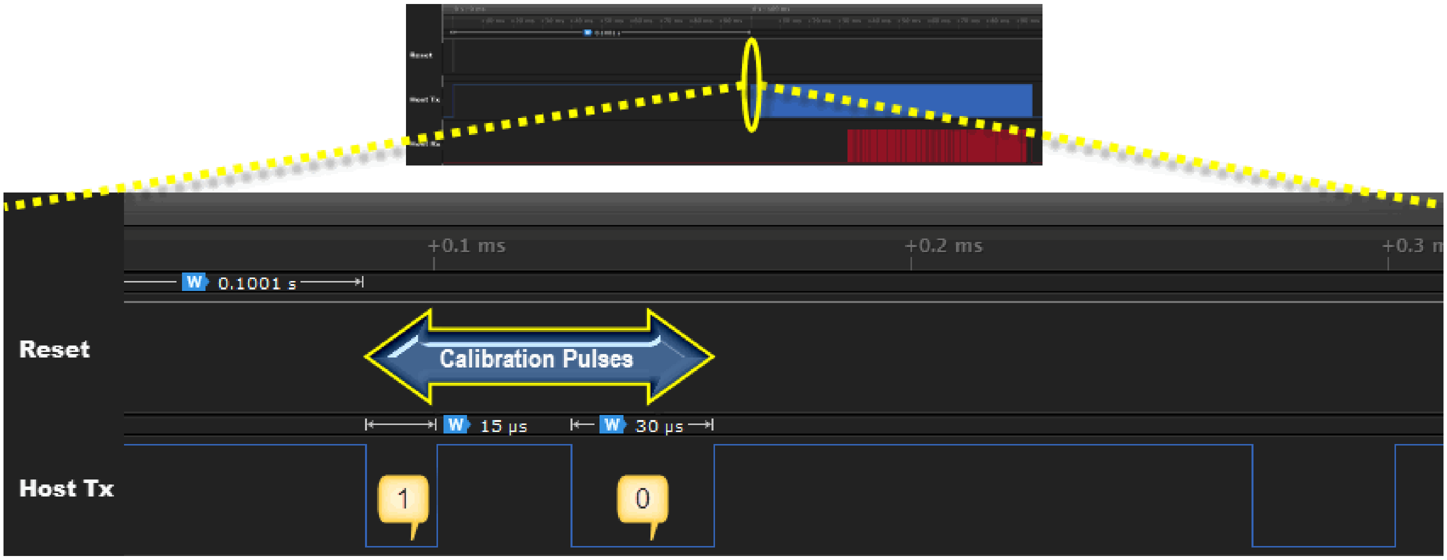
This image shows a full identification sequence with Protocol Proper. Driven signals rest high; Host Rx isn't driven by the Propeller until it recognizes the host. Note the approximate 100 ms post-reset delay (after Reset's low pulse) before host transmission begins (Host Tx). The following are zoomed-in views of this image.

Image P2 - Reset Pulse



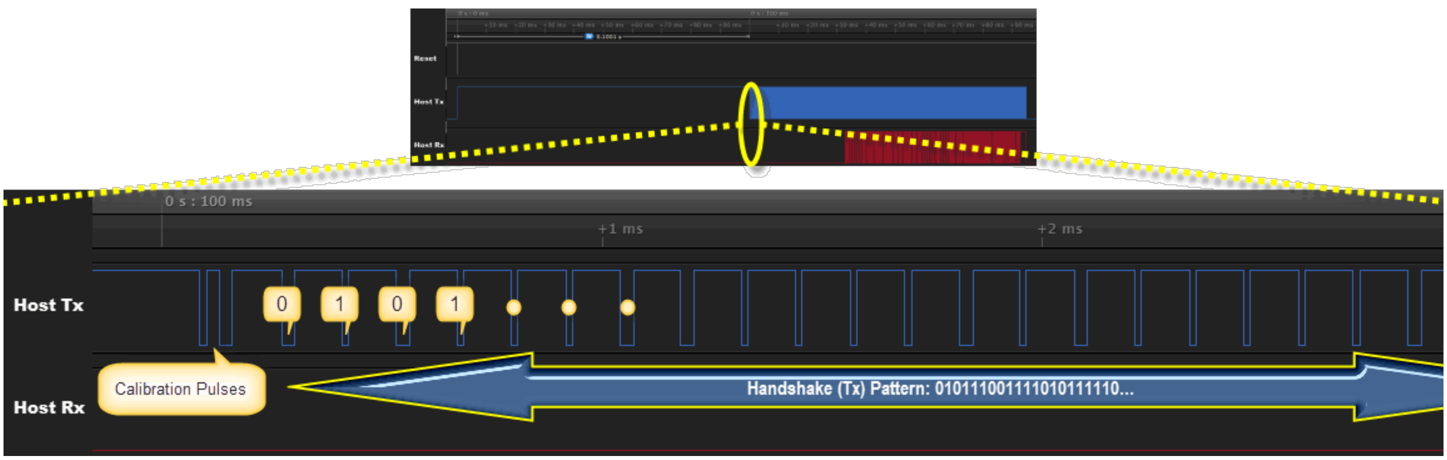
The Host-driven Reset Pulse (when connected to the Propeller's RESn pin) causes the Propeller to stop any activity and perform a boot sequence. The Reset Pulse is shown here followed by an approximate 100 ms delay before Host Tx communication begins. The Host Tx signal (connected to the Propeller's Rx pin) must “rest” high before the Propeller wakes up on the rising edge of the Reset signal.

Image P3 - Calibration Pulses



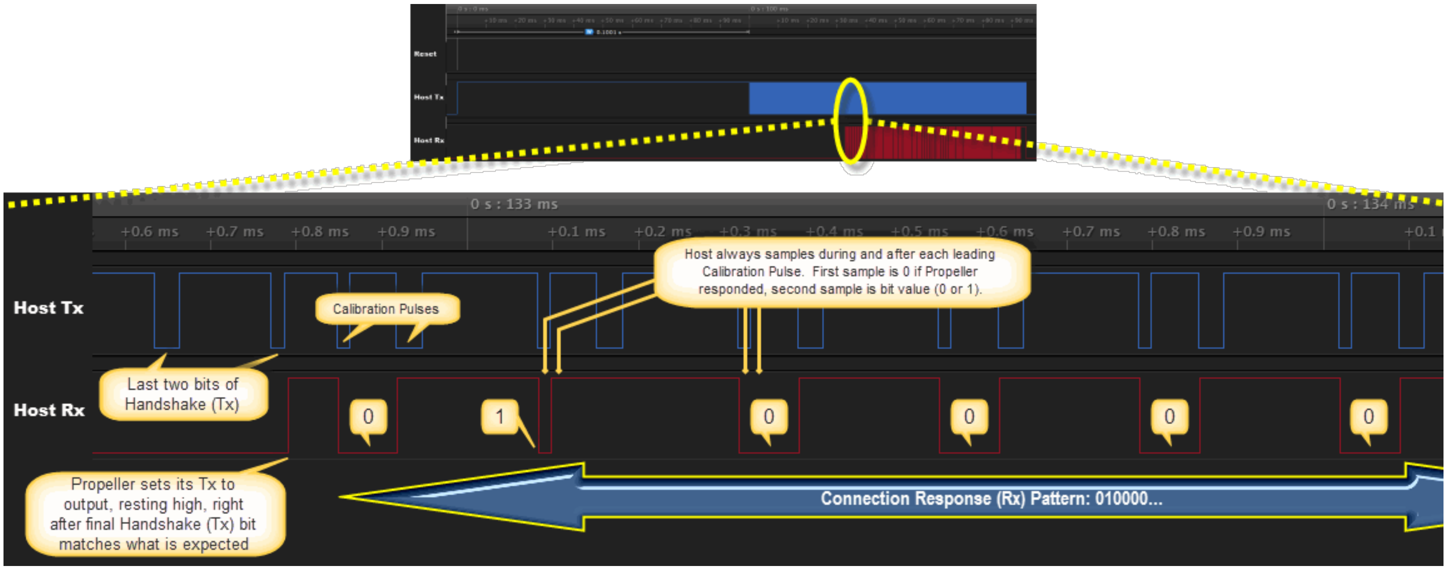
Approximately 100 ms after the rising edge of the Reset Pulse, Host Tx communication begins with Calibration Pulses. This is the host's way of saying, "This is how I talk." The 2-bit calibration sequence consists of a low pulse (of width t), a spacer (high) pulse, and a second low pulse (of width 2t) followed by a spacer (high).

Image P4 - Handshake Pattern



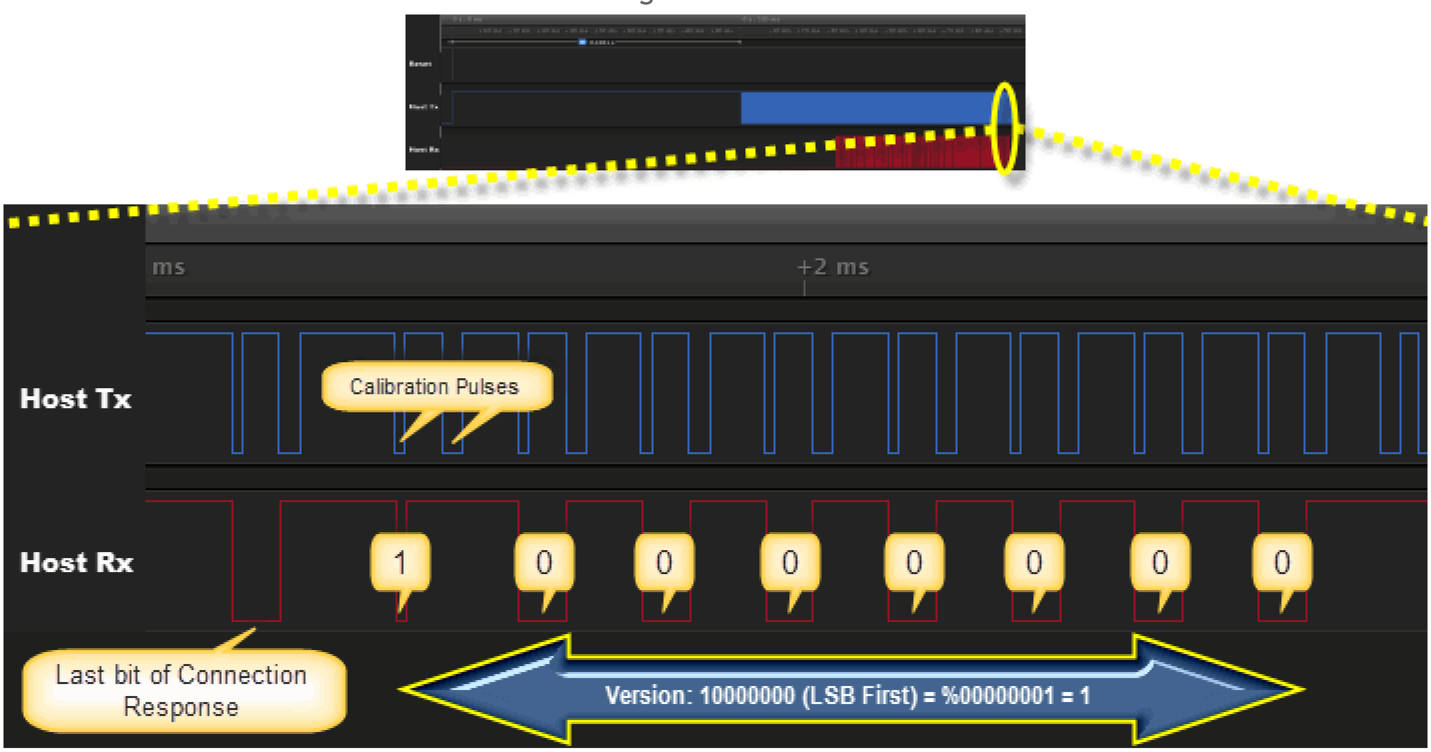
The Handshake pattern (the host-transmitted pattern of 250 bits) immediately follows the Calibration Pulses. This is the host's way of saying, "I'm a qualified Propeller host trying to speak to a Propeller." Pulse widths of the 250-bit pattern closely match the timing indicated by the Calibration Pulses. The contents of this pattern is described in [Appendix A](#).

Image P5 - Connection Response



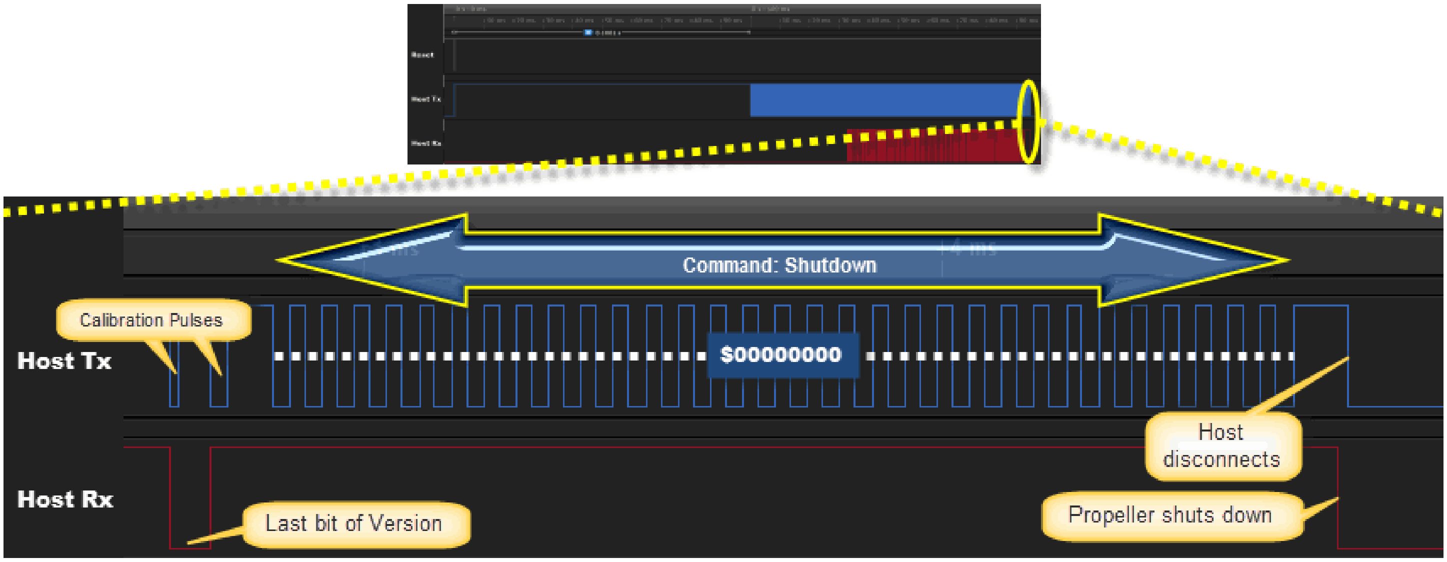
The Connection Response (a Propeller-transmitted pattern of 250 bits) occurs right after the Handshake. It is the Propeller's way of saying, "I'm a Propeller that hears and acknowledges the qualifications of the host." After the last bit of the Handshake, the Propeller transmits one bit of the Connection Response for each Calibration Pulse pair it receives from the host. The host transmits 250 Calibration Pulse pairs in order to receive all Connection Response bits. The host always samples the Propeller's transmission (Host Rx) during and right after each leading Calibration Pulse bit; the first sample is always 0 if the Propeller responded at all and the second sample is 0 or 1, the actual bit value of the Connection bit. NOTE: The Propeller doesn't set its Tx line (connected to Host Rx) to output (resting high) until after the last correct Handshake pattern bit is received. The contents of this Connection Response pattern is described in [Appendix A](#).

Image P6 - Version



The Version is an 8-bit value transmitted by the Propeller immediately after the Connection Response; 1 bit (LSB-first) for every Calibration Pulse pair the host transmits.

Image P7 - Command (Shutdown)

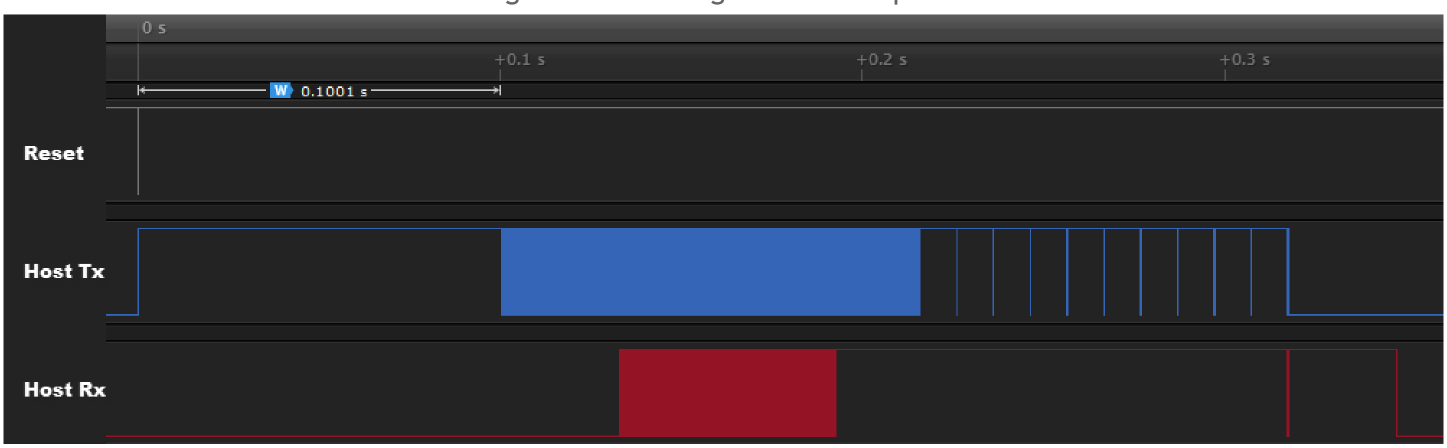


After Version is received, the host transmits the Command (the intent of the communication); in this case, Shutdown, aka Identify. The Command is a 32-bit value (LSB-first). If the Command were other than Shutdown, communication would continue as shown in the images below.

Program RAM Sequence

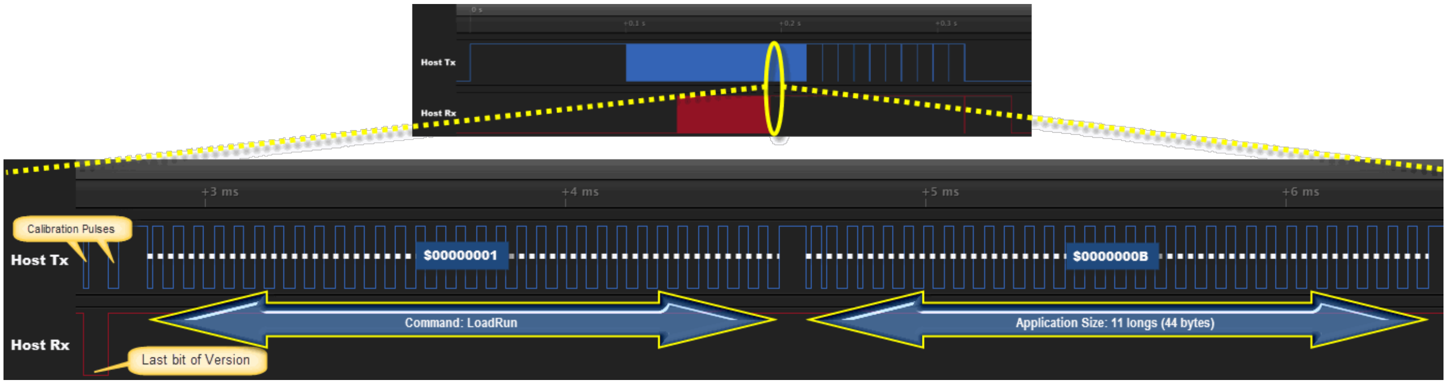
The next set of images expands on the sequence shown above by demonstrating a Program RAM sequence, where the LoadRun command is used and an actual Propeller Application is transmitted. All communication prior to the LoadRun command is exactly the same as before.

Image P8 - Full Program RAM Sequence



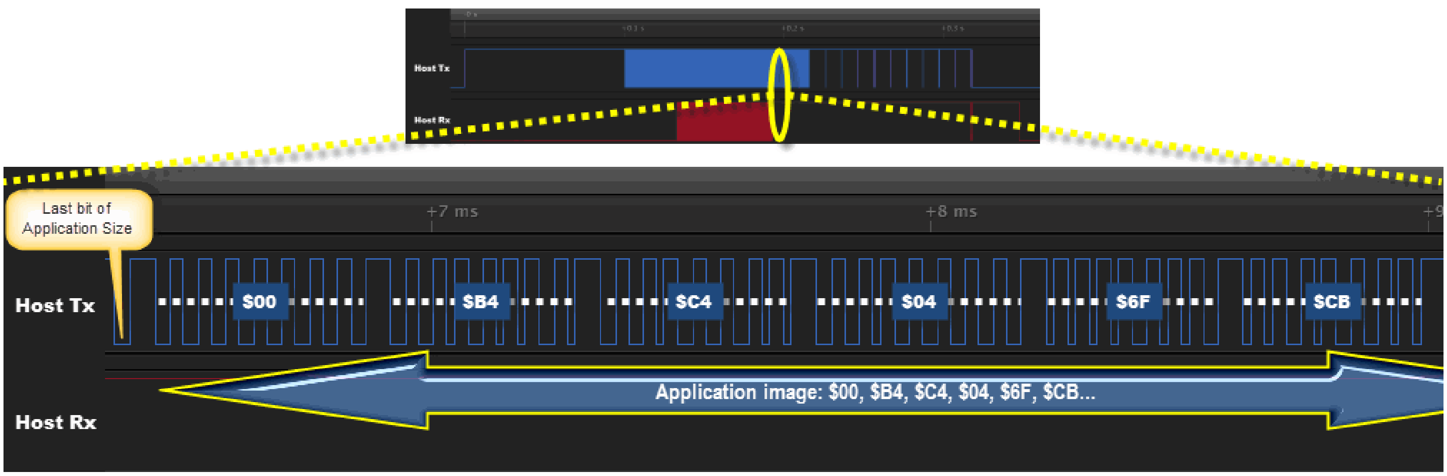
This is an entire Program RAM sequence with Protocol Proper (LoadRun Command). The leading signals, up until the LoadRun command is transmitted, is exactly the same as in the Identification sequence shown in prior images. The following are zoomed-in views of this image.

Image P9 - Command (LoadRun) and Application Size



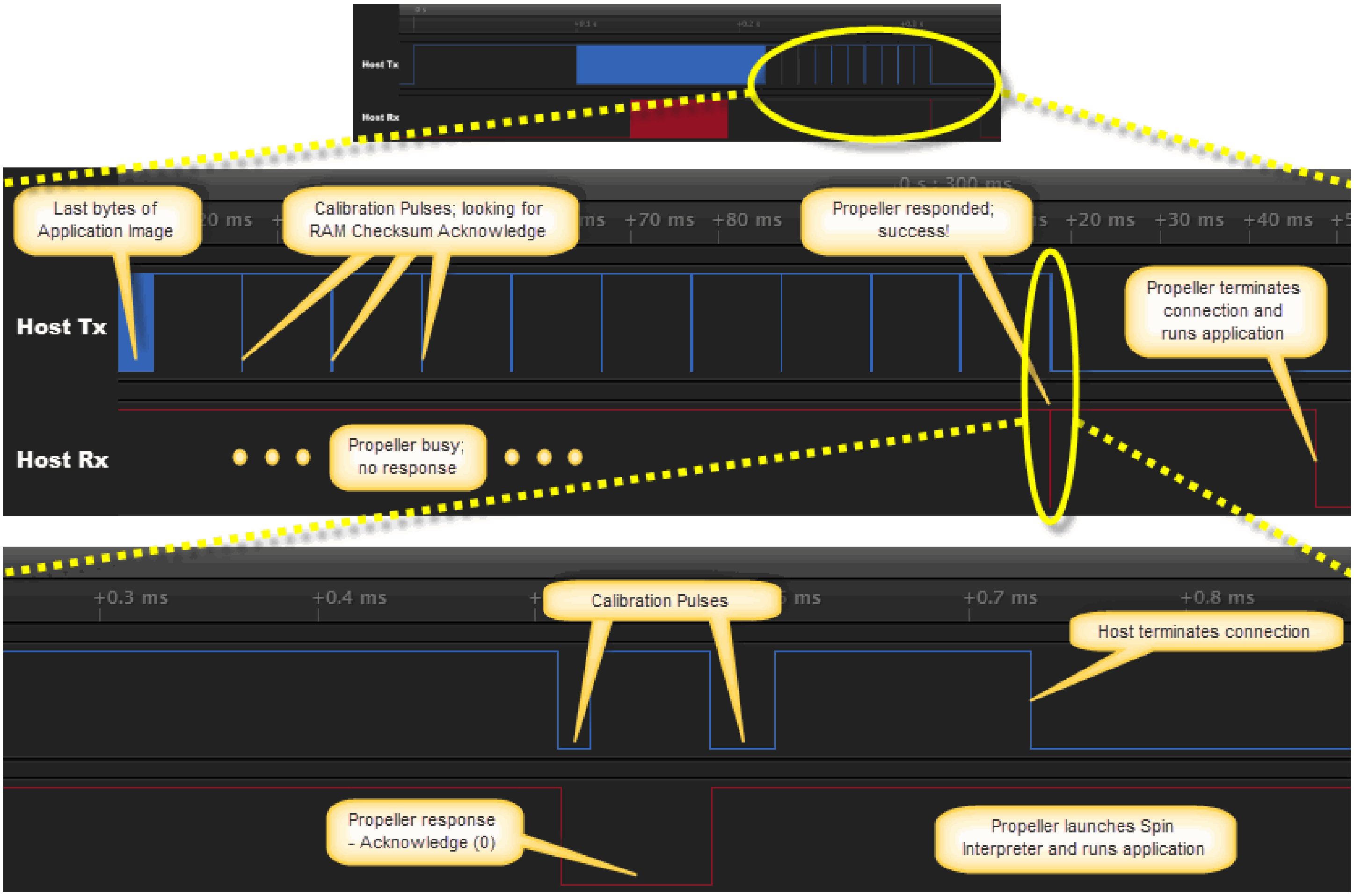
After Version is received, the host transmits the LoadRun command followed by the Application Size. Each are 32-bit values transmitted LSB-first. The actual Propeller Application image is expected to follow.

Image P10 - Application Image



After the Application Size is transmitted, the host transmits each byte of the Application Image. Each 8-bit value is transmitted LSB-first. The small Propeller Application used in this example consists of 44 bytes, transmitted using the same low-pulse method as before: `00 B4 C4 04 6F CB 10 00 2C 00 34 00 18 00 38 00 1C 00 02 00 08 00 00 00 37 03 3D D6 1C 37 03 3D D4 47 35 C0 3F 91 EC 23 04 73 32 00`. You can use this same application data during protocol development; when successfully downloaded, it will make the Propeller toggle I/O pin P16 high or low every second (assuming the Propeller has a 5 MHz crystal connected).

Image P11 - Acknowledge (RAM Checksum)

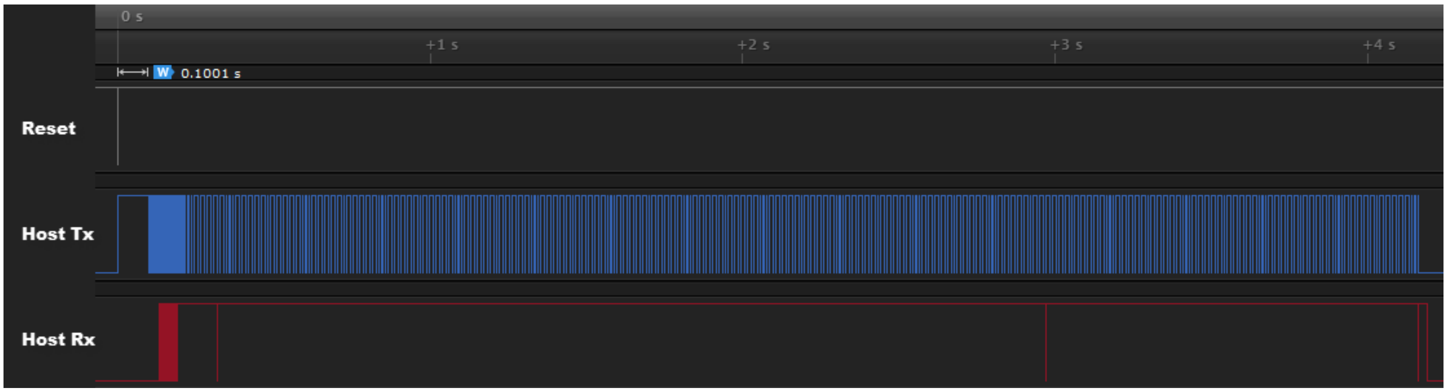


After the last bytes of the Application Image are transmitted, the host polls for a RAM Checksum Acknowledgement by periodically transmitting Calibration Pulses and checking for a Propeller response. Polling may continue for up to 250 ms with delays between polls of > 10 ms and < 100 ms recommended. The Propeller responds with an Ack (0) if the received Application Image has been verified in RAM (via RAM Checksum) or with a Nak (1) if RAM verification failed. If Ack is received and the Command is LoadRun (as in this example), this is the end of the download process. If Ack is received and the Command is ProgramRun or ProgramShutdown, communication continues as in the following images.

Program EEPROM Sequence

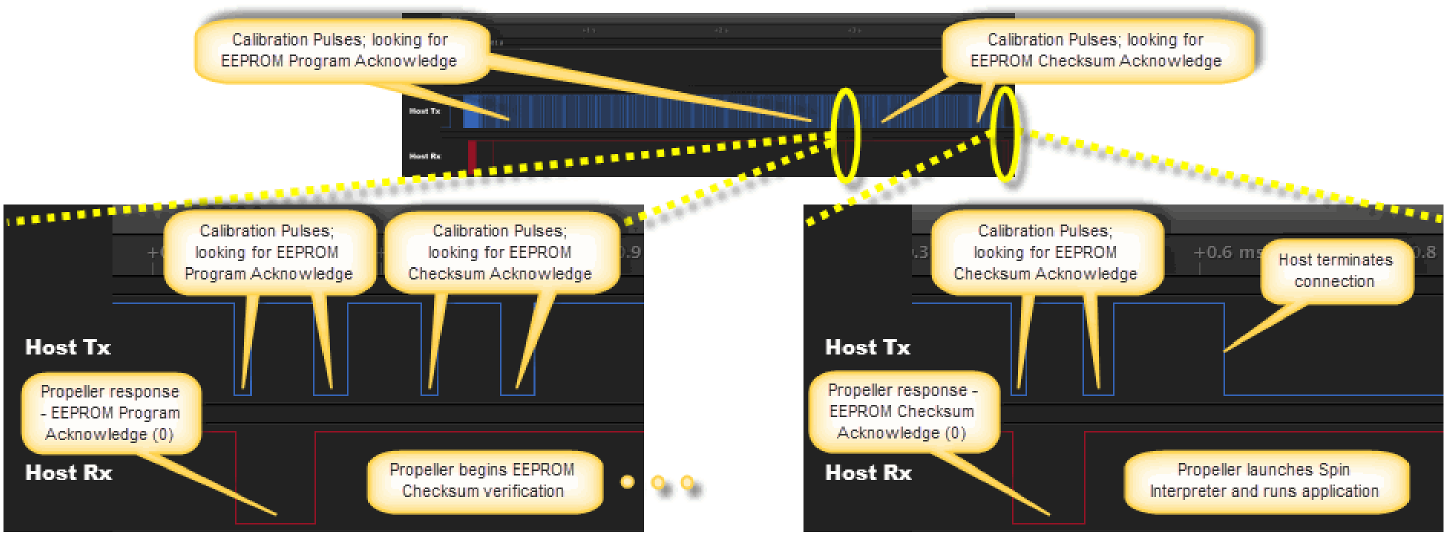
The final set of Protocol Proper images, below, expands on the previous by demonstrating a Program EEPROM sequence, where the ProgramRun command is used instead. All communication is exactly like above except the Command is ProgramRun and two more polling sequences occur after the positive Acknowledge (RAM Checksum) is received.

Image P12 - Full Program EEPROM Sequence



This is an entire Program EEPROM sequence with Protocol Proper (ProgramRun Command). Communication is exactly like in previous images, except for the ProgramRun command and the extra polling sequences. The following is a zoomed-in view of this image.

Image P13 - EEPROM Program and Checksum Acknowledge



Assuming the Command is ProgramRun or ProgramShutdown, after the Propeller acknowledges the RAM Checksum, it immediately starts the EEPROM programming and verification processes. The host polls for an EEPROM Program Acknowledgement by periodically transmitting Calibration Pulses and checking for a Propeller response. Polling may continue for up to 5 s with delays between polls of > 10 ms and < 100 ms recommended. The Propeller responds with an Ack (0) if EEPROM Programming is done, or with a Nak (1) if EEPROM programming failed. If Ack is received, the host once again polls (this time for EEPROM checksum verification) similar to before, but only for up to 2 s. Once a final Ack is received, that marks the end of the download process. NOTE: If the Command is ProgramShutdown, the Propeller terminates the connection swiftly after final acknowledgement, rather than launching the Spin Interpreter and application.

Protocol RS-232

This implementation is fit for UART and USB Virtual COM Port based connections from the host to the Propeller. Developers are urged to read through and understand Protocol Proper, even if they only intend to implement Protocol RS-232, since full understanding of the intent can not be gained otherwise.

Protocol RS-232 takes the nature of Protocol Proper and carefully translates it to fit the constraints of the RS-232 protocol. The resulting signals are, from the Propeller's perspective, indistinguishable from that of Protocol Proper except in terms of timing. The same exact General Principles apply here as well.

Except for cases where special techniques are applied to overcome behaviors of typical host systems, it can be said that Protocol RS-232 is simply a translation wrapper around Protocol Proper.

Metrics

- **Baud Rate** - 38,400 to 230,400 bps (115,200 recommended).
- **Data Bits** - 8
- **Parity** - None
- **Stop Bits** - 1
- **Flow Control** - Off
- **Delay Between Bytes** - ≤ 90 ms

Protocol RS-232 Algorithm

The following describes the steps for Protocol RS-232 from the perspective of the host. The images afterwards contain additional vital information using actual annotated examples.

When implementing this protocol from scratch, try attempting only an identification operation (the first half of the algorithm) then the final steps (in the second half) will generally be easier to implement.

- **Reset the Propeller** [Image RS1, Image RS2]
  - Set DTR/RTS (which drives Propeller RESn low) > 10 µs, then clear DTR/RTS
- **Wait for Post-Reset Delay** [Image RS1, Image RS2]
  - 90 to 100 ms recommended
- **Transmit Calibration Pulses** [Image RS3]
  - 2-bits encoded in 1 byte ($F9; %11111001)
- **Transmit Handshake Pattern** [Image RS4]
  - 250-bits encoded into multiple bytes; see Appendix A - Handshake and Connection
- **Receive Connection Response** [Image RS5]
  - 250 iterations of: transmit 2-bit Calibration Pulse as 1 byte ($F9), receive 1-bit Connection Response Bit as byte ($FE or $FF); see Appendix A
  - If Propeller does not respond to any Calibration Pulse, or Connection Response bit is invalid, abort communication; connection error
- **Receive Version** [Image RS6]
  - 8 iterations of: transmit 2-bit Calibration Pulse as 1 byte ($F9), receive 1-bit Version Pulse as byte ($FE or $FF)
  - If Propeller does not respond to any Calibration Pulse, or if Version is bad, transmit Shutdown Command, abort communication; version error
  - Valid Propeller Version = 1 (8-bit value, received LSB-first)
- **Transmit Command** [Image RS7]

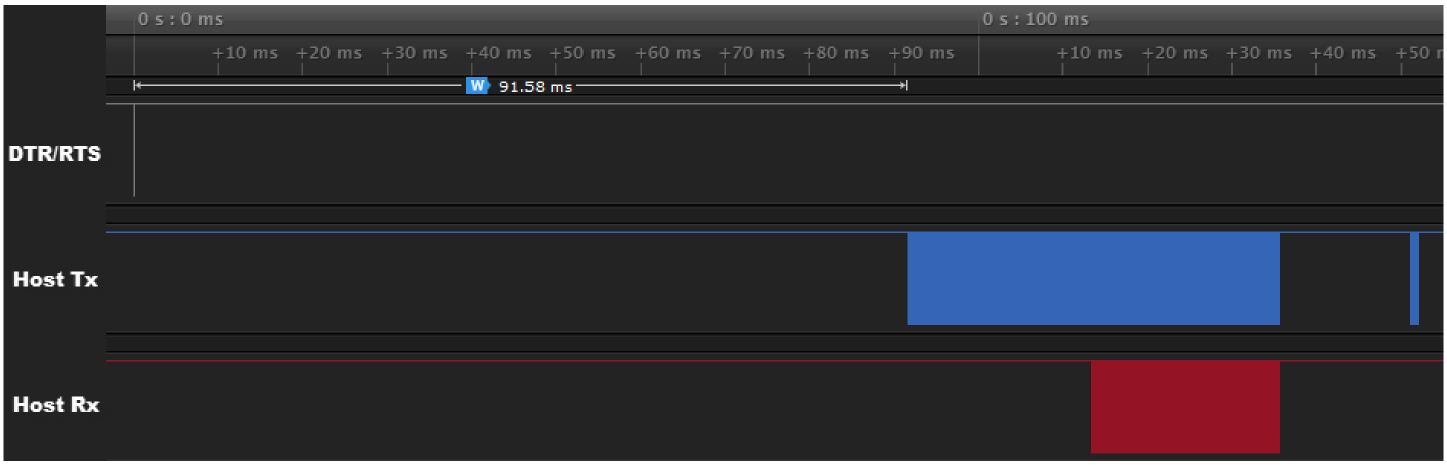
- 32-bits as multiple bytes; often Shutdown (ID), LoadRun (RAM) or ProgramRun (EEPROM); see Table G1
  - If transmitted Command is Shutdown (aka Identify), success; terminate communication
- **Transmit Size of Application in longs []**
  - 32-bits as multiple bytes; value should come from word 9:8 of the application image, shifted right by 2 bits
- **Transmit Application Image []**
  - Stream of bits encoded into multiple bytes
- **Wait for Acknowledge (RAM Checksum) []**
  - Up to 250 ms of: transmit 2-bit Calibration Pulse as 1 byte ($F9), check for Ack ($FE) or Nak ($FF)
  - If no response before timeout or Negative Acknowledge (Nak) received, abort communication; transmission error or RAM verify error, respectively
- **If Command is LoadRun []**
  - Success; terminate communication
- **Else, Command is ProgramShutdown or ProgramRun []**
  - **Wait for EEPROM Program []**
    - Up to 5 s of: transmit 2-bit Calibration Pulse as 1 byte ($F9), check for Ack ($FE) or Nak ($FF)
    - If no response before timeout or Negative Acknowledge (Nak) received, abort communication; EEPROM program error
  - **Wait for EEPROM Verify []**
    - Up to 2 s of: transmit 2-bit Calibration Pulse as 1 byte ($F9), check for Ack ($FE) or Nak ($FF))
    - If no response before timeout or Negative Acknowledge (Nak) received, abort communication; EEPROM verify error
  - **Success; terminate communication**

The following images are actual logic analyzer captures of Protocol RS-232. The example system consists of a "host" computer whose USB-based serial converter device is connected to a Propeller. The host's RS-232 baud rate is set to 115.2 k. The signals are illustrated from the host's perspective but were captured from the Propeller side of the connection.

Identification Sequence

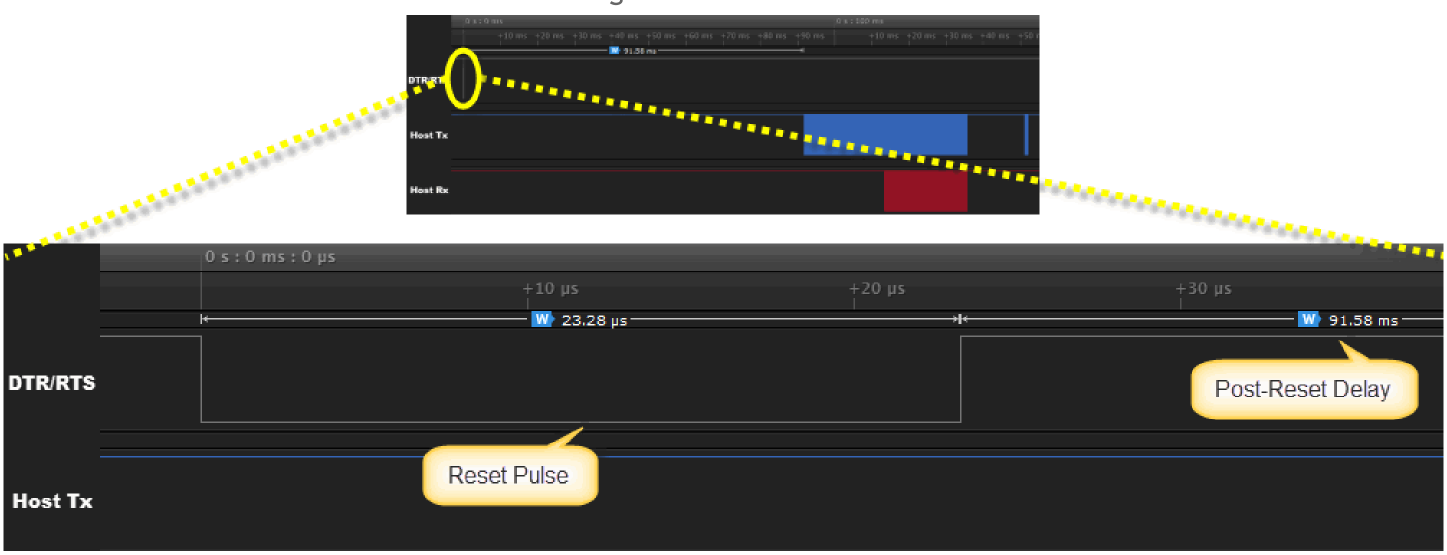
The first set of images demonstrates the identification process (Reset through Command) which covers the first half of the Propeller RS-232 Algorithm.

Image RS1 - Full Identification Sequence



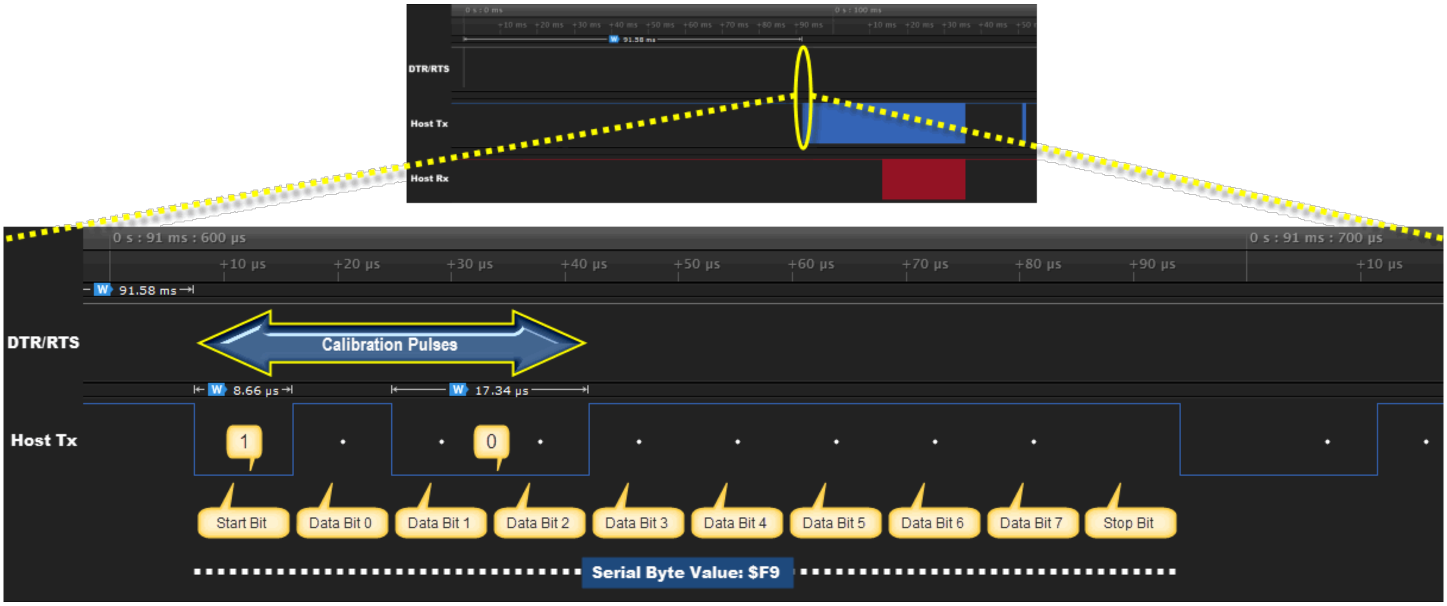
This image shows a full identification sequence with Protocol RS-232. Signals rest high. Note the approximate 90 ms post-reset delay (after DTR/RTS's low pulse) before host transmission begins (Host Tx). The following are zoomed-in views of this image.

Image RS2 - Reset Pulse



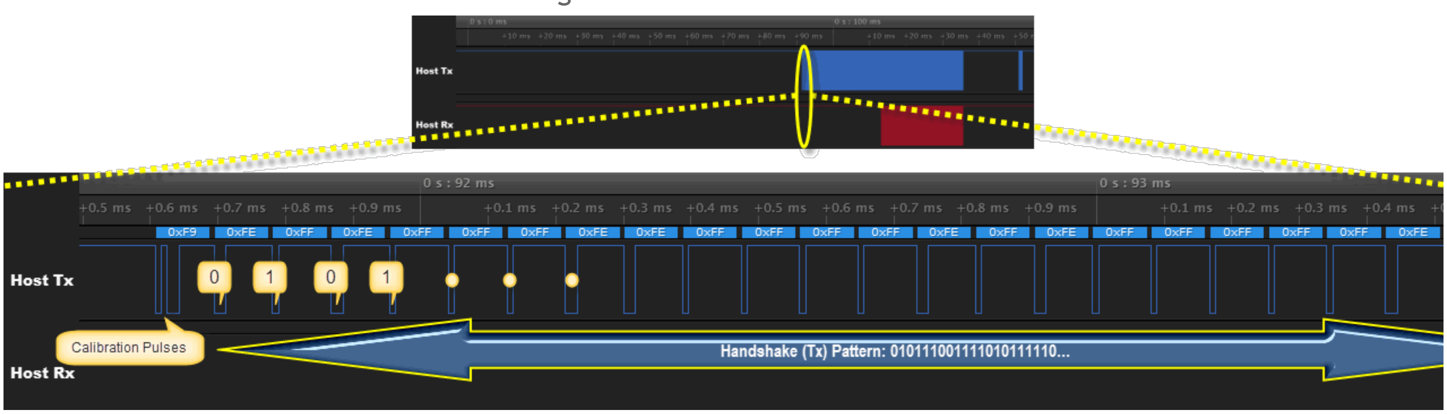
The host-driven Reset Pulse on DTR/RTS (when connected to the Propeller's RESn pin) causes the Propeller to stop any activity and perform a boot sequence. The Reset Pulse is shown here followed by an approximate 90 ms delay before Host Tx communication begins.

Image RS3 - Calibration Pulses



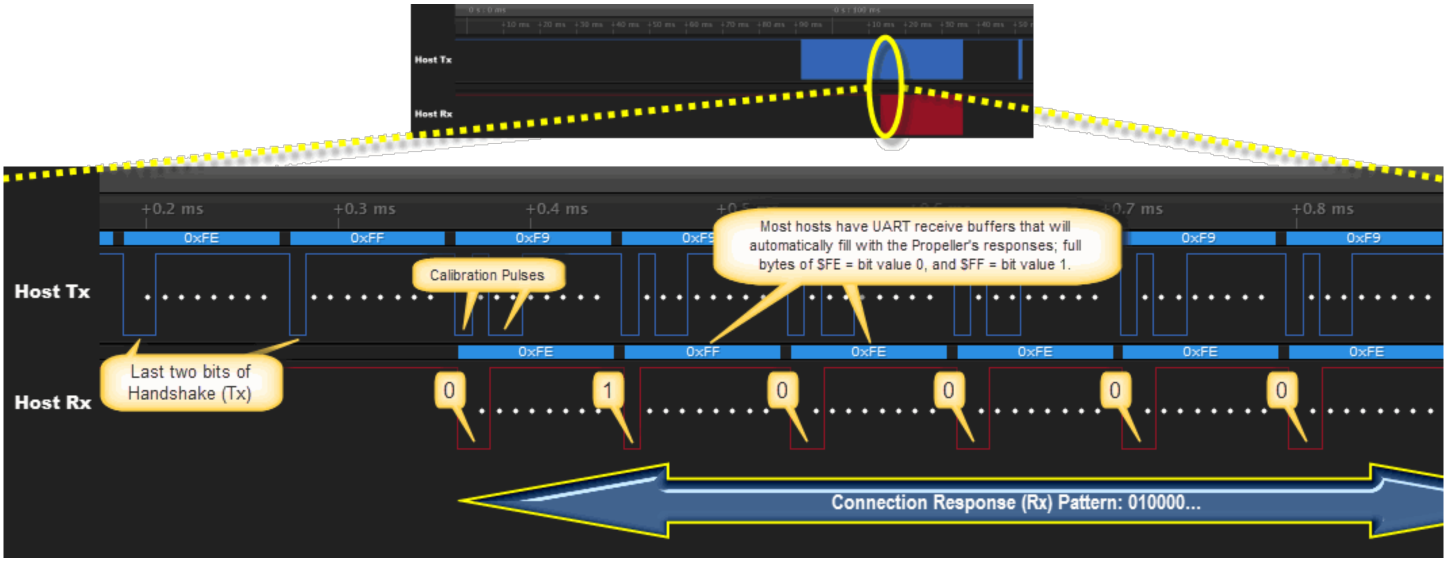
Approximately 90 ms after the rising edge of the Reset Pulse (DTR/RTS), Host Tx communication begins with Calibration Pulses. This is the host's way of saying, "This is how I talk." The same 2-bit calibration sequence described by [Protocol Proper](#) is actually communicated by Protocol RS-232 as a serial byte value of $F9 (%11111001), at 115.2 kbaud in this case. Note that the effective signal looks very similar to [Protocol Proper's Calibration Pulses](#) and, with the [General Principles](#) in mind, translates to *exactly* the same data; a binary 1 and a binary 0 communicated as a low pulse (of width t), a spacer (high), and a second low pulse (of width 2t) followed by a spacer (high). Also note that the serial start bit itself (a low pulse signaling the beginning of a serial byte) is always a data bit as interpreted by Protocol Proper. Similarly, the stop bit itself (a high pulse marking the end of a serial byte) is always a spacer as interpreted by Protocol Proper.

Image RS4 - Handshake Pattern



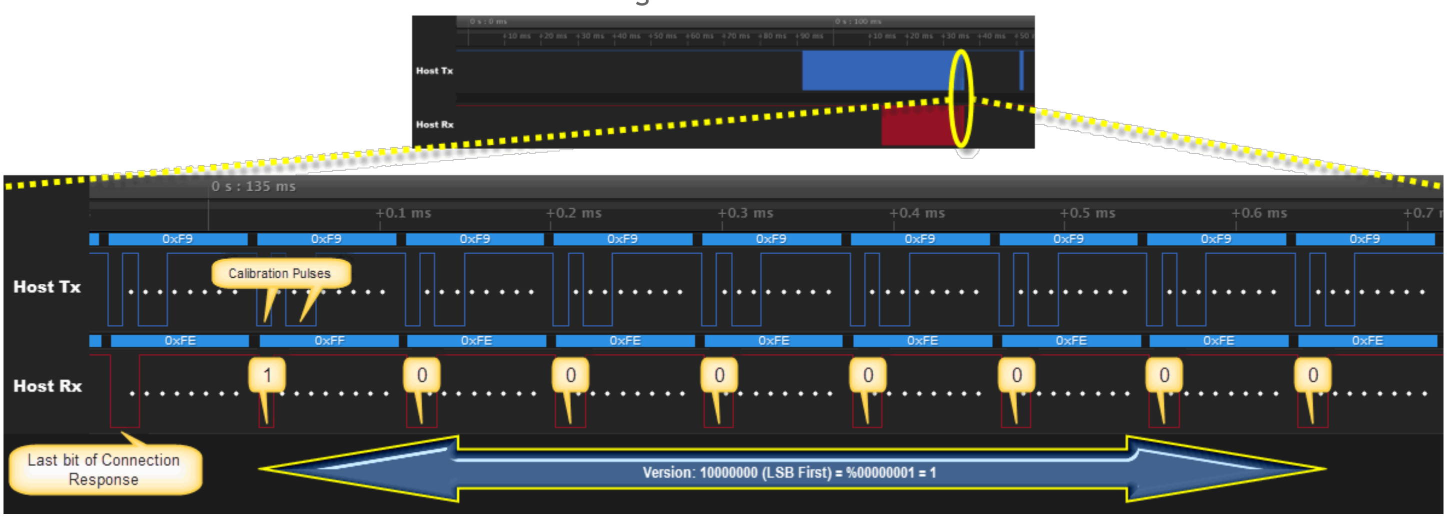
The Handshake pattern (the host-transmitted pattern of 250 bits) immediately follows the Calibration Pulses. This is the host's way of saying, "I'm a qualified Propeller host trying to speak to a Propeller." As with the Calibration Pulses, the 250-bit pattern matches that of Protocol Proper (looking at low pulses only). In this example of Protocol RS-232, binary 0 is expressed with serial byte value $FE (%11111110) and a binary 1 as serial byte value $FF (%11111111). Remember, the start bit of each serial byte is a data bit as interpreted by Protocol Proper. This serial stream translates to the same data as, and looks similar to, Protocol Proper's [Handshake](#). The contents of the Handshake pattern is described in [Appendix A](#).

Image RS5 - Connection Response



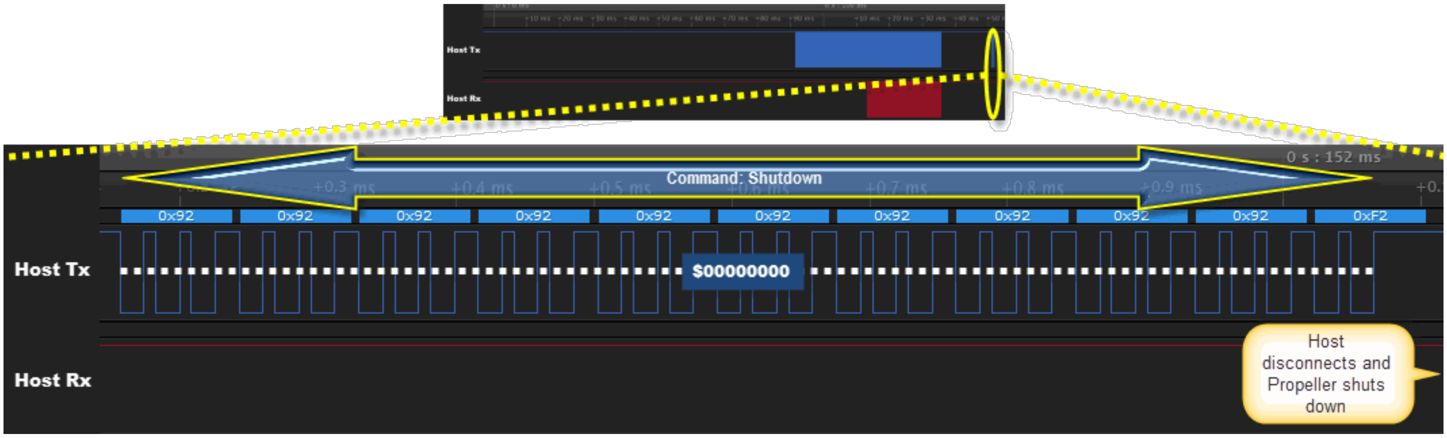
The Connection Response (a Propeller-transmitted pattern of 250 bits) occurs right after the Handshake. It is the Propeller's way of saying, "I'm a Propeller that hears and acknowledges the qualifications of the host." After the last bit of the Handshake, the Propeller transmits one bit of the Connection Response for each Calibration Pulse pair it receives from the host. The host transmits 250 Calibration Pulse pairs (as serial bytes $F9) in order to receive all Connection Response bits; 1 bit per response byte. The host usually has a UART receive buffer that automatically fills with the Propeller's responses; full bytes of $FE (meaning bit value 0) and $FF (meaning bit value 1). NOTE: The Propeller doesn't set its Tx line (connected to Host Rx) to output until after the last correct Handshake bit is received. Some circuitry mistakenly treats this initially-floating signal as actual bytes received– this necessitates careful flushing of the host's receive buffer to properly parse the Connection Response. The contents of this Connection Response pattern is described in [Appendix A](#).

Image RS6 - Version



The Version is an 8-bit value transmitted by the Propeller immediately after the Connection Response; 1 bit per byte (LSB-first) for every Calibration Pulse pair (byte) the host transmits. The host should parse the LSBs of the 8 response bytes into a single 8-bit value to validate the Version response.

Image RS7 - Command (Shutdown)



After Version is received, the host transmits the Command (the intent of the communication); in this case, Shutdown, aka Identify. The Command is a 32-bit value (LSB-first) encoded into multiple bytes (11 in this case). If the Command were other than Shutdown, communication would continue as shown in the images below.

Appendix A - Handshake and Connection

The Handshake and Connection pattern (aka Handshake Tx and Handshake Rx) is a pseudo-random sequence of 500-bits that the host and Propeller use to identify and validate each other. The bit pattern is derived from an 8-bit Linear Feedback Shift Register (LFSR) tap that generates 255 different 8-bit values (pseudo-randomly) before repeating (on the 256th iteration). The host and the Propeller use the same LFSR tap value to independently generate the same bit sequence, and each relies on the nature of the repetition boundary as part of the sequence.

The bit pattern itself can easily be auto-generated at run time (algorithmically) or can be pre-generated and read from memory at run-time. Both methods are described below.

Auto-generated Values

This algorithm assumes that a globally-scoped, byte-sized variable (named LFSR) is pre-set to the initial seed value of 80 ($50), the ASCII value of character ‘P’. The LFSR tap bits are 7, 5, 4, and 1. Upon every iteration, the current values of the LFSR variable’s tap bits are exclusive-or’d together, the current 8-bit LFSR value is shifted left 1 bit, and the 1-bit result of the aforementioned exclusive-or’d tap set is stored into the LFSR’s bit 0.

Algorithm for function IterateLFSR:

1. Set Result to: LFSR
2. Set LFSR to: **(**LFSR **<<** 1**)** **|** **(((**LFSR **>>** 7**)** **^** **(**LFSR **>>** 5**)** **^** **(**LFSR **>>** 4**)** **^** **(**LFSR **>>** 1**))** **&** 1**)**

NOTE: **&** is bitwise AND, **^** is bitwise XOR, **|** is bitwise OR, **<<** is bitwise shift left, and **>>** is bitwise shift right

Each call of the function *IterateLFSR* returns the current value of LFSR and then updates LFSR to the next 8-bit value in the pseudo-random number sequence. Though the LFSR variable holds an 8-bit value, only the least significant bit (LSB; bit 0) of each generated value is transmitted or received as part of the Handshake or Connection stream. NOTE: For simplicity, the *IterateLFSR* function can be made to return only the current LSB, instead of the entire LFSR value.

Pre-generated Values

The *IterateLFSR* function noted above generates the 255 values shown here, in order. Further iterations beyond the 255th call of *IterateLFSR* repeat this exact sequence again.

LFSR Output (255 8-bit Hex Values)

```
50 A1 42 85 0B 17 2E 5C B9 73 E7 CF 9E 3D 7A F5 EB D7 AF 5F BE 7C F8 F1 E3 C7 8E 1C 39 72 E5 CA
94 28 51 A3 47 8F 1E 3C 78 F0 E1 C2 84 09 12 24 49 92 25 4B 97 2F 5E BC 79 F2 E4 C8 91 22 44 88
11 23 46 8D 1B 36 6D DB B7 6E DC B8 71 E2 C5 8B 16 2C 59 B3 66 CC 99 32 65 CB 96 2D 5B B6 6C D9
B2 64 C9 93 27 4E 9D 3A 75 EA D5 AA 55 AB 57 AE 5D BB 76 ED DA B5 6B D6 AD 5A B4 69 D3 A7 4F 9F
3F 7F FF FE FC F9 F3 E6 CD 9B 37 6F DE BD 7B F7 EE DD BA 74 E8 D0 A0 40 80 01 02 05 0A 15 2B 56
AC 58 B1 63 C6 8C 19 33 67 CE 9C 38 70 E0 C0 81 03 07 0F 1F 3E 7D FA F4 E9 D2 A5 4A 95 2A 54 A9
52 A4 48 90 20 41 82 04 08 10 21 43 87 0E 1D 3B 77 EF DF BF 7E FD FB F6 EC D8 B0 61 C3 86 0C 18
31 62 C4 89 13 26 4C 98 30 60 C1 83 06 0D 1A 34 68 D1 A2 45 8A 14 29 53 A6 4D 9A 35 6A D4 A8
```

The least significant bits of the sequence above, assembled as a 255-bit stream, looks like the following. As should be expected, beyond the 255th bit, the exact sequence repeats again.

LSB of LFSR Output (255-bit stream)

```
0101110011110101111110001110010100011110000100100101111001000 1000
1101101110001011001100101101100100111010101011101101011010011111
1110011011110111010000000101011000110011100000011111010010101001
000001000011101111111011000011000100110000001101000101001101010 00
```

Either one of the above value sequences can be stored and retrieved from memory to process the Handshake and Connection phase of communication.

Usage During Communication

When protocol communication begins, both the host and the Propeller prepare for their own copy of the Handshake and Connection sequence, whether it be auto-generated or pre-generated.

During the Handshake phase, the host transmits the LSB of each of the first 250 values in the LFSR sequence. The Propeller follows along, comparing those received bits to its own sequence, and if they all match, the Propeller switches to the Connection phase. In the Connection phase, the two sides swap roles; the Propeller transmits the LSB of each of the next 250 values in the sequence (ie: starting at value 251) and the host follows along, comparing those to the rest if its own sequence. The pattern of the two 250-bit sequences intentionally overlaps the repetition boundary by 5 bits. If both sides see that the streams match what was expected, each considers the other side validated and qualified to continue the conversation.

This is illustrated in the Protocol Proper Algorithm and the data can be seen when you compare the first bits of the sequences (shown above) to the patterns in Image P4 - Handshake Pattern and Image P5 - Connection Response as well as Image RS4 - Handshake Pattern and Image RS5 - Connection Response. Keep in mind, the sequence repeats after 255 bits, but the Handshake phase only outputs the first 250 bits. The Connection phase (which also outputs 250 bits) continues where the other left off… first using the last 5 bits of the remaining sequence and then repeating the first 245 bits of the same sequence.

Appendix B - PropellerLoader.spin

The Spin object, PropellerLoader.spin, implements Protocol Proper from one Propeller (the host) to another (being programmed). The source is provided below for review and the object file can also be downloaded from the [Propeller Object Exchange](#).

PropellerLoader.spin

```
''***************************************
''*  Propeller Loader v1.0              *
''*  Author: Chip Gracey                *
''*  Copyright (c) 2006 Parallax, Inc.  *
''*  See end of file for terms of use.  *
''***************************************

' v1.0 - 13 June 2006 - original version

''_____________________________________________________________________________
''
''This object lets a Propeller chip load up another Propeller chip in the same
''way the PC normally does.
''
''To do this, the program to be loaded into the other Propeller chip must be
''compiled using "F8" (be sure to enable "Show Hex") and then a "Save Binary
''File" must be done. This binary file must then be included so that it will be
''resident and its address can be conveyed to this object for loading.
''
''Say that the file was saved as "loadme.binary". Also, say that the Propeller
''which will be performing the load/program operation has its pins 0..2 tied to
''the other Propeller's pins RESn, P31, and P30, respectively. And we'll say
''we're working with Version 1 chips and you just want to load and execute the
''program. Your code would look something like this:
''
''
''OBJ loader : "PropellerLoader"
''
''DAT loadme file "loadme.binary"
''
''PUB LoadPropeller
''
''  loader.Connect(0, 1, 2, 1, loader#LoadRun, @loadme)
''
''
''This object drives the other Propeller's RESn line, so it is recommended that
''the other Propeller's BOEn pin be tied high and that its RESn pin be pulled
''to VSS with a 1M resistor to keep it on ice until showtime.
''_____________________________________________________________________________
''


CON

  #1, ErrorConnect, ErrorVersion, ErrorChecksum, ErrorProgram, ErrorVerify
  #0, Shutdown, LoadRun, ProgramShutdown, ProgramRun
```

```
VAR

  long P31, P30, LFSR, Ver, Echo


PUB Connect(PinRESn, PinP31, PinP30, Version, Command, CodePtr) : Error

  'set P31 and P30
  P31 := PinP31
  P30 := PinP30

  'RESn low
  outa[PinRESn] := 0
  dira[PinRESn] := 1

  'P31 high (our TX)
  outa[PinP31] := 1
  dira[PinP31] := 1

  'P30 input (our RX)
  dira[PinP30] := 0

  'RESn high
  outa[PinRESn] := 1

  'wait 100ms
  waitcnt(clkfreq / 10 + cnt)

  'Communicate (may abort with error code)
  if Error := \Communicate(Version, Command, CodePtr)
    dira[PinRESn] := 0

  'P31 float
  dira[PinP31] := 0


PRI Communicate(Version, Command, CodePtr) | ByteCount

  'output calibration pulses
  BitsOut(%01, 2)

  'send LFSR pattern
  LFSR := "P"
  repeat 250
    BitsOut(IterateLFSR, 1)

  'receive and verify LFSR pattern
  repeat 250
    if WaitBit(1) <> IterateLFSR
      abort ErrorConnect

  'receive chip version
  repeat 8
    Ver := WaitBit(1) << 7 + Ver >> 1

  'if version mismatch, shutdown and abort
  if Ver <> Version
    BitsOut(Shutdown, 32)
    abort ErrorVersion
```

```
  'send command
  BitsOut(Command, 32)

  'handle command details
  if Command

    'send long count
    ByteCount := byte[CodePtr][8] | byte[CodePtr][9] << 8
    BitsOut(ByteCount >> 2, 32)

    'send bytes
    repeat ByteCount
      BitsOut(byte[CodePtr++], 8)

    'allow 250ms for positive checksum response
    if WaitBit(25)
      abort ErrorChecksum

    'eeprom program command
    if Command > 1

      'allow 5s for positive program response
      if WaitBit(500)
        abort ErrorProgram

      'allow 2s for positive verify response
      if WaitBit(200)
        abort ErrorVerify


PRI IterateLFSR : Bit

  'get return bit
  Bit := LFSR & 1

  'iterate LFSR (8-bit, $B2 taps)
  LFSR := LFSR << 1 | (LFSR >> 7 ^ LFSR >> 5 ^ LFSR >> 4 ^ LFSR >> 1) & 1


PRI WaitBit(Hundredths) : Bit | PriorEcho

  repeat Hundredths

    'output 1t pulse
    BitsOut(1, 1)

    'sample bit and echo
    Bit := ina[P30]
    PriorEcho := Echo

    'output 2t pulse
    BitsOut(0, 1)

    'if echo was low, got bit
    if not PriorEcho
      return

    'wait 10ms
    waitcnt(clkfreq / 100 + cnt)
```

```
  'timeout, abort
  abort ErrorConnect


PRI BitsOut(Value, Bits)

  repeat Bits

    if Value & 1

      'output '1' (1t pulse)
      outa[P31] := 0
      Echo := ina[P30]
      outa[P31] := 1

    else

      'output '0' (2t pulse)
      outa[P31] := 0
      outa[P31] := 0
      Echo := ina[P30]
      Echo := ina[P30]
      outa[P31] := 1

    Value >>= 1

{{
```



```
}}
```

Appendix C - Propeller's ROM-Based Boot Loader

The following Propeller assembly code (PASM) is the actual source used to build the ROM-resident Boot Loader that starts up in the Propeller when it is powered up or reset. Developers that know PASM can gain important insight into Protocol Proper, and more, by studying this code. Note: During development, the codename for the Propeller was "PNut."

```
'       *****************************************
'       *                                       *
'       *       PNut Booter                     *
'       *                                       *
'       *       Version 0.1   12/10/2004        *
'       *                                       *
'       *       (C) 2004 Parallax, Inc.         *
'       *                                       *
'       *****************************************
'
'
' Entry
'
DAT                     org

                        test    mask_rx,ina     wc      'if rx high, check for host
        if_nc           jmp     #boot                   'else, boot from eeprom

                        call    #rx_bit                 'measure rx calibration pulses ($F9)
                        mov     threshold,delta         'and calculate threshold
                        call    #rx_bit                 '(any timeout results in eeprom boot)
                        add     threshold,delta
                        shr     threshold,#1

                        mov     count,#250              'ready to receive/verify 250 lfsr bits
:lfsrin                 call    #rx_bit                 'receive bit ($FE/$FF) into c
                        muxc    lfsr,#$100              'compare to lfsr lsb
                        test    lfsr,#$101      wc
        if_c            jmp     #boot                   'if mismatch, boot from eeprom
                        test    lfsr,#$B2       wc      'advance lfsr
                        rcl     lfsr,#1
                        djnz    count,#:lfsrin

                        or      outa,mask_tx            'host present, make tx high output
                        or      dira,mask_tx

                        mov     count,#250              'ready to transmit 250 lfsr bits
:lfsrout                test    lfsr,#$01       wz      'send lfsr bit ($FE/$FF)
                        call    #tx_bit
                        test    lfsr,#$B2       wc      'advance lfsr
                        rcl     lfsr,#1
                        djnz    count,#:lfsrout

                        rdbyte  bits,hFFF9FFFF          'get version byte at $FFFF
                        mov     count,#8                'send version byte
:version                test    bits,#$01       wz
                        call    #tx_bit
                        shr     bits,#1
                        djnz    count,#:version

                        call    #rx_long                'receive command
                        mov     command,rxdata

                        tjz     command,#shutdown       'if command 0, shutdown
                        cmp     command,#4      wc      'if command 4+, shutdown
        if_nc           jmp     #shutdown

                        call    #rx_long                'get long count
                        mov     count,rxdata

                        mov     address,#0              'get longs into ram
```

```
:longs                  call    #rx_long
                        wrlong  rxdata,address
                        add     address,#4
                        djnz    count,#:longs

                        mov     count,h8000             'zero remainder of ram
                        sub     count,address
                        shr     count,#2        wz
:zero   if_nz           wrlong  zero,address
        if_nz           add     address,#4
        if_nz           djnz    count,#:zero            '(count=0, address=$8000)

                        rdword  bits,#$0004+6           'get dbase address
                        sub     bits,#4                 'set pcurr to $FFF9
                        wrlong  hFFF9FFFF,bits
                        sub     bits,#4                 'set pbase flags
                        wrlong  hFFF9FFFF,bits

                        mov     bits,#0                 'compute ram checksum
:checksum               rdbyte  rxdata,count
                        add     bits,rxdata
                        add     count,#1
                        djnz    address,#:checksum
                        test    bits,#$FF       wz      'z=1 if checksum okay

                        call    #tx_bit_align           'send checksum okay/error

        if_nz           jmp     #shutdown               'if checksum error, shutdown

                        djnz    command,#program        'if command 2-3, program eeprom
                        jmp     #launch                 'else command 1, launch
'
'
' Program and verify eeprom from ram
'
program                 mov     smode,#1                'set mode in case error

                        mov     address,#0              'reset address
:page                   call    #ee_write               'send program command
                        mov     count,#$40              'page-program $40 bytes
:byte                   rdbyte  eedata,address          'get ram byte
                        call    #ee_transmit            'send ram byte
        if_c            jmp     #shutdown               'if no ack, shutdown
                        add     address,#1              'inc address
                        djnz    count,#:byte            'loop until page sent
                        call    #ee_stop                'initiate page-program cycle
                        cmp     address,h8000   wz      'check for address $8000
        if_nz           jmp     #:page                  'loop until done (z=1 after)

                        call    #tx_bit_align           'program done, send okay (z=1)

                        call    #ee_read                'send read command
:verify                 call    #ee_receive             'get eeprom byte
                        rdbyte  bits,address            'get ram byte
                        cmp     bits,eedata     wz      'compare bytes
        if_nz           jmp     #shutdown               'if verify error, shutdown (sends error)
                        add     address,#1              'inc address
                        djnz    count,#:verify          'loop until done
                        call    #ee_stop                'end read (z=1 from before)

                        call    #tx_bit_align           'verify done, send okay (z=1)

                        mov     smode,#0                'clear mode in case error

                        djnz    command,#launch         'if command 3, launch
                        jmp     #shutdown               'else command 2, shutdown
'
'
' Load ram from eeprom and launch
'
boot                    mov     smode,#0                'clear mode in case error

                        call    #ee_read                'send read command
```

```
:loop                   call    #ee_receive             'get eeprom byte
                        wrbyte  eedata,address          'write to ram
                        add     address,#1              'inc address
                        djnz    count,#:loop            'loop until done
                        call    #ee_stop                'end read (followed by launch)
'
'
' Launch program in ram
'
launch                  rdword  address,#$0004+2        'if pbase address invalid, shutdown
                        cmp     address,#$0010  wz
        if_nz           jmp     #shutdown

                        rdbyte  address,#$0004          'if xtal/pll enabled, start up now
                        and     address,#$F8            '..while remaining in rcfast mode
                        clkset  address

:delay                  djnz    time_xtal,#:delay       'allow 20ms @20MHz for xtal/pll to settle

                        rdbyte  address,#$0004          'switch to selected clock
                        clkset  address

                        coginit interpreter             'reboot cog with interpreter
'
'
' Shutdown
'
shutdown                mov     ee_jmp,#0               'deselect eeprom (replace jmp with nop)
:call                   call    #ee_stop                '(always returns)

                        cmp     smode,#0        wz      'if serial mode, send error (z=0)
        if_nz           mov     smode,#0                '(only do once)
        if_nz           mov     :call,#0                '(replace call with nop)
        if_nz           call    #tx_bit_align           '(may return to shutdown, no problem)

                        mov     dira,#0                 'cancel any outputs

                        mov     smode,#$02              'stop clock
                        clkset  smode                   '(suspend until reset)
'
'
'**************************************
'* I2C routines for 24x256/512 EEPROM *
'* assumes fastest RC timing - 20MHz  *
'*   SCL low time  =  8 inst, >1.6us  *
'*   SCL high time =  4 inst, >0.8us  *
'*   SCL period    = 12 inst, >2.4us  *
'**************************************
'
'
' Begin eeprom read
'
ee_read                 mov     address,#0              'reset address

                        call    #ee_write               'begin write (sets address)

                        mov     eedata,#$A1             'send read command
                        call    #ee_start
        if_c            jmp     #shutdown               'if no ack, shutdown

                        mov     count,h8000             'set count to $8000

ee_read_ret             ret
'
'
' Begin eeprom write
'
ee_write                call    #ee_wait                'wait for ack and begin write

                        mov     eedata,address          'send high address
                        shr     eedata,#8
                        call    #ee_transmit
        if_c            jmp     #shutdown               'if no ack, shutdown
```

```
                        mov     eedata,address          'send low address
                        call    #ee_transmit
        if_c            jmp     #shutdown               'if no ack, shutdown

ee_write_ret            ret
'
'
' Wait for eeprom ack
'
ee_wait                 mov     count,#400              '       400 attempts > 10ms @20MHz
:loop                   mov     eedata,#$A0             '1      send write command
                        call    #ee_start               '132+
        if_c            djnz    count,#:loop            '1      if no ack, loop until done

        if_c            jmp     #shutdown               '       if no ack, shutdown

ee_wait_ret             ret
'
'
' Start + transmit
'
ee_start                mov     bits,#9                 '1      ready 9 start attempts
:loop                   andn    outa,mask_scl           '1(!)   ready scl low
                        or      dira,mask_scl           '1!     scl low
                        nop                             '1
                        andn    dira,mask_sda           '1!     sda float
                        call    #delay5                 '5
                        or      outa,mask_scl           '1!     scl high
                        nop                             '1
                        test    mask_sda,ina    wc      'h?h    sample sda
        if_nc           djnz    bits,#:loop             '1,2    if sda not high, loop until done

        if_nc           jmp     #shutdown               '1      if sda still not high, shutdown

                        or      dira,mask_sda           '1!     sda low
'
'
' Transmit/receive
'
ee_transmit             shl     eedata,#1               '1      ready to transmit byte and receive ack
                        or      eedata,#%00000000_1     '1
                        jmp     #ee_tr                  '1

ee_receive              mov     eedata,#%11111111_0     '1      ready to receive byte and transmit ack

ee_tr                   mov     bits,#9                 '1      transmit/receive byte and ack
:loop                   test    eedata,#$100    wz      '1      get next sda output state
                        andn    outa,mask_scl           '1!     scl low
                        rcl     eedata,#1               '1      shift in prior sda input state
                        muxz    dira,mask_sda           '1!     sda low/float
                        call    #delay4                 '4
                        test    mask_sda,ina    wc      'h?h    sample sda
                        or      outa,mask_scl           '1!     scl high
                        nop                             '1
                        djnz    bits,#:loop             '1,2    if another bit, loop

                        and     eedata,#$FF             '1      isolate byte received
ee_receive_ret
ee_transmit_ret
ee_start_ret            ret                             '1      nc=ack
'
'
' Stop
'
ee_stop                 mov     bits,#9                 '1      ready 9 stop attempts
:loop                   andn    outa,mask_scl           '1!     scl low
                        nop                             '1
                        or      dira,mask_sda           '1!     sda low
                        call    #delay5                 '5
                        or      outa,mask_scl           '1!     scl high
                        call    #delay3                 '3
                        andn    dira,mask_sda           '1!     sda float
```

```
                call    #delay4                 '4
                test    mask_sda,ina    wc      'h?h    sample sda
        if_nc   djnz    bits,#:loop             '1,2    if sda not high, loop until done

ee_jmp  if_nc   jmp     #shutdown               '1      if sda still not high, shutdown

ee_stop_ret     ret                             '1
'
'
' Cycle delays
'
delay5          nop                             '1
delay4          nop                             '1
delay3          nop                             '1
delay2
delay2_ret
delay3_ret
delay4_ret
delay5_ret      ret                             '1
'
'
'*******************
'* Serial routines *
'*******************
'
'
' Transmit bit (nz)
' conveys incoming $F9 on rx to $FE/$FF on tx
'
tx_bit_align    mov     time,time_load          'reset time limit

:align          call    #rx_bit                 'align to next $F9
        if_c    jmp     #:align

tx_bit          mov     time,time_load          'reset time limit

:high           test    mask_rx,ina     wc      'wait while high
        if_c    djnz    time,#:high
        if_c    jmp     #shutdown               'if timeout, shutdown

                andn    outa,mask_tx            'tx low

:low            test    mask_rx,ina     wc      'wait while low
        if_nc   djnz    time,#:low
        if_nc   jmp     #shutdown               'if timeout, shutdown

                muxnz   outa,mask_tx            'tx low/high

:high2          test    mask_rx,ina     wc      'wait while high
        if_c    djnz    time,#:high2
        if_c    jmp     #shutdown               'if timeout, shutdown

                or      outa,mask_tx            'tx high

:low2           test    mask_rx,ina     wc      'wait while low
        if_nc   djnz    time,#:low2
        if_nc   jmp     #shutdown               'if timeout, shutdown

tx_bit_ret
tx_bit_align_ret        ret
'
'
' Receive long
'
rx_long         mov     time,time_load          'reset time limit

                mov     bits,#32                'ready for 32 bits
:loop           call    #rx_bit                 'input bit
                rcr     rxdata,#1               'shift into long
                djnz    bits,#:loop             'loop until done

rx_long_ret     ret
'
```

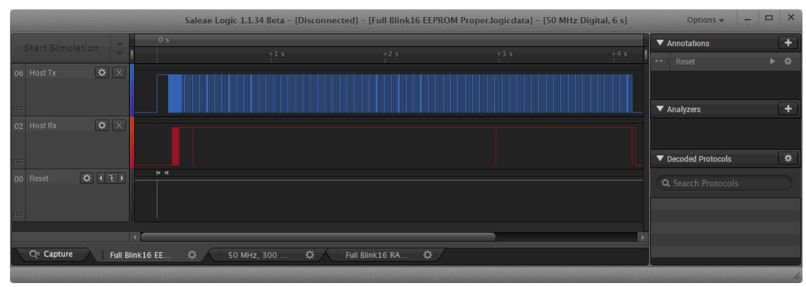
```
'
' Receive bit (c)
'
rx_bit                  test    mask_rx,ina     wc      'wait while rx high
        if_c            djnz    time,#rx_bit
        if_c            jmp     #boot                   'if timeout, boot from eeprom

                        mov     delta,time              'time while rx low

:loop                   test    mask_rx,ina     wc      'h?h    2 instructions/loop
        if_nc           djnz    time,#:loop             '1      400ns @20MHz...1us @8MHz
        if_nc           jmp     #boot                   'if timeout, boot from eeprom

                        sub     delta,time              'delta = rx low time in loops
                        cmp     delta,threshold wc      'resolve bit into c

rx_bit_ret              ret
'
'
' Constants
'
mask_rx                 long    $80000000
mask_tx                 long    $40000000
mask_sda                long    $20000000
mask_scl                long    $10000000
time                    long    150 * 20000 / 4 / 2     '150ms (@20MHz, 2 inst/loop)
time_load               long    100 * 20000 / 4 / 2     '100ms (@20MHz, 2 inst/loop)
time_xtal               long    20 * 20000 / 4 / 1      '20ms (@20MHz, 1 inst/loop)
lfsr                    long    "P"
zero                    long    0
smode                   long    0
hFFF9FFFF               long    $FFF9FFFF
h8000                   long    $8000
interpreter             long    $0001 << 18 + $3C01 << 4 + %0000
'
'
' Variables
'
command                 res     1
address                 res     1
count                   res     1
bits                    res     1
eedata                  res     1
rxdata                  res     1
delta                   res     1
threshold               res     1
```

Appendix D - Logic Analyzer

A critical tool for properly implementing protocols is a logic analyzer or oscilloscope to see what's actually going on at the signal level, in a digital and analog sense.

Parallax recommends the very compact and affordable [Saleae Logic] series of analyzers that provide both digital and analog measurement. These small hardware devices plug into a computer's USB port for analyzing signals using dedicated software. Capture, zoom, protocol analysis, and many other features make this a great tool; a vital part of an embedded system designer's toolkit.

The Saleae Logic 8 model was used to create all the logic capture images in this document.



Scratch Pad (To be moved or deleted when document is finished).

Encoded LFSR (8-bit Bin) [Tx Handshake Pattern for Protocol RS-232]

```
FE FF FE FF FF FF FE FE FF FF FF FF FE FF FE FF FF FF FF FF FE FE FE FF FF
FF FE FE FF FE FF FE FE FE FF FF FF FF FE FE FE FE FF FE FE FF FE FE FF FE
FF FF FF FF FE FE FF FE FE FE FF FE FE FE FF FF FE FF FF FE FF FF FF FE FE
FE FF FE FF FF FE FE FF FF FE FE FF FE FF FF FE FF FF FE FE FF FE FE FF FF
FF FE FF FE FF FE FF FE FF FF FF FE FF FF FE FF FE FF FF FE FF FE FE FF FF
FF FF FF FF FF FF FE FE FF FF FE FF FF FF FF FE FF FF FF FE FF FE FE FE FE
FE FE FE FF FE FF FE FF FF FE FE FE FF FF FE FE FF FF FF FE FE FE FE FE FE
FF FF FF FF FF FE FF FE FE FF FE FF FE FF FE FE FF FE FE FE FE FE FF FE FE
FE FE FF FF FF FE FF FF FF FF FF FF FE FF FF FE FE FE FE FF FF FE FE FE FF
FE FE FF FF FE FE FE FE FE FF FF FE FF FE FE FE FF FE FF FE FE FF FF FE FF
```

Encoded LFSR (8-bit Bin) [Rx Connection Pattern for Protocol RS-232]

```
FE FF FE FE FE FE FF FE FF FF FF FE FE FF FF FF FF FE FF FE FF FF FF FF FF
FE FE FE FF FF FF FE FE FF FE FF FE FE FE FF FF FF FF FE FE FE FE FF FE FE
FF FE FE FF FE FF FF FF FF FE FE FF FE FE FE FF FE FE FE FF FF FE FF FF FE
FF FF FF FE FE FE FF FE FF FF FE FE FF FF FE FE FF FE FF FF FE FF FF FE FE
FF FE FE FF FF FF FE FF FE FF FE FF FE FF FF FF FE FF FF FE FF FE FF FF FE
FF FE FE FF FF FF FF FF FF FF FF FE FE FF FF FE FF FF FF FF FE FF FF FF FE
FF FE FE FE FE FE FE FE FF FE FF FE FF FF FE FE FE FF FF FE FE FF FF FF FE
FE FE FE FE FE FF FF FF FF FF FE FF FE FE FF FE FF FE FF FE FE FF FE FE FE
FE FE FF FE FE FE FE FF FF FF FE FF FF FF FF FF FF FE FF FF FE FE FE FE FF
FF FE FE FE FF FE FE FF FF FE FE FE FE FE FF FF FE FF FE FE FE FF FE FF FE
```