draft

EADIN Lite Network Protocol User Manual

Software versions 2.x.x

By Eliot Aretskin-Hariton,

eliot.d.aretskin-hariton at nasa.gov

Aircraft engines today uses analog signaling to control all aspects of the engine. These signals require dedicated copper lines for each sensor and actuator which increases the weight and complexity of the engine system. Analog systems have been preferred in the past because they are robust, the underlying technologies that support distributed engine control were not ready, and the FAA certification process is well understood. The progression of moderately high-temperature (250­0 C) electronics, supports alternative digital networks architectures. EADIN Lite is being used as a test network at the NASA Glenn Research Center, to understand the limitations of digital control networks for aircraft engines. EADIN Lite is a custom protocol for half-duplex networks what was designed based on the preliminary Engine Area Distributed Interconnect Network (EADIN) specification as described by the Distributed Engine Control Working Group (DECWG). This manual will describe the functionality of the EADIN Lite network and design decisions.

Table of Contents

[2 Overview: 2](#_Toc444083033)

[3 Design Specifications: 2](#_Toc444083034)

[4 Function Calls 3](#_Toc444083035)

[4.1 Public Functions 3](#_Toc444083036)

[4.2 Private Functions 4](#_Toc444083037)

[4.2.1 CRC Functions 4](#_Toc444083038)

[5 Example Usage 5](#_Toc444083039)

[6 Error Messages 7](#_Toc444083040)

[7 Variables: 8](#_Toc444083041)

[8 General Operation: 9](#_Toc444083042)

EADIN Lite Open Source License:

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at:

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

# Overview

The EADIN Lite network was constructed to replicate the basic functionality of the EADIN network as described by the DECWG. The desire was to be able to replicate the basic functionality of the proposed engine control network so that we could investigate its capabilities. This is essential to enable us to answer questions like, how much bandwidth is required for the engine control network. There is little consensus at the moment about items like this. Additionally, we have a desire to create a model of how an engine control network will behave. We do this by creating a real EADIN network, and then using statistical data to model the control network in software. This will provide us with the capabilities of running aircraft engine simulations with a control network included in faster than real time. Additionally, since the EADIN Lite network operates in real time, it can be used in conjunction with the Hardware-In-the-Loop simulation at NASA GRC to operate in hard-real time with hardware sensors and actuators attached.

# Design Specifications

Specifications for the construction of this network is based on two documents released by DECWG[[1]](#footnote-1),[[2]](#footnote-2). From these documents we drew our initial design specifications which allowed us to create our network. These specifications are as follows:

1. 18 bytes sent / received for every message
2. 10-17 miliseconds per minor frame
3. Max clock speed based on current high-temperature electronics, 20MHz
4. Network Hardware: RS-485 Half duplex (simplex) connections that support multi-drop and 10Mb/s
5. 14 bit Break Field
6. 10 bit sync field
7. A multi byte data field that contains destination node information and a 10 bit Identifier (ENQ/ACK/NAK) field
8. CRC-15 algorithm using the 0xC599 polynomial which is similar to the CAN algorithm

There are many more specifications in these two documents that go into the minutia about how to program the nodes, how to talk to multiple nodes at the same time, how to read specific memory locations of a remote node. We implement only the most basic constraints, and use them to construct our control network which should be capable of roughly simulating the engine control network of the future. The network is not capable of loading software on the microcontrollers. That operation needs to be done before running this network protocol.

# Function Calls

The EADIN Lite library v2.0.0 is built using function calls from the Arduino.h libraries. Additionally, the library uses an object oriented approach to make reading and writing to the network simpler for the user. The following are the valid function calls the user can make. Default values are specified for function inputs in eadin.h and eadin.cpp.

## Public Functions

|  |  |  |
| --- | --- | --- |
| Usage | Example | Notes |
| EADIN OBJ(my\_ID, qCRC, poly) | EADIN TTL(0x00,false,0xC529);  This creates an object of type EADIN with node ID of 0x00 (master) and use the CRCSlow() algorithm with the polynomial 0xC529. | Used to create the object during setup phase. |
| EADIN TTL(0x01); | Typical usage accepting default values for everything but my\_ID. |
| OBJ.begin(\*rw\_port, baud, RTS) | TTL.begin(&Serial1,9600,4);  Opens the hardware serial port Serial1 at 9600 baud and uses digital I/O port #4 for the Read / Transmit switch (RTS). | Used to assign basic values of the object, and open the serial ports. The function also automatically adds a delay of ~3 seconds before loading up the serial ports because some microcontroller boards have reprogramming issues if there is no delay at the start of the setup code. |
| OBJ.write(data[], dest, cmd\_only) | TTL.write(my\_dat, 0x01, false);  Write an array of 8 bytes contained in my\_dat to node 0x01 and request a response. | Write the data array to a remote node with the option to request no response. Restrictions exist.[[3]](#footnote-3) |
|  | TTL.write(my\_dat,0x01);  Write array of data to node 0x01. | Typical usage for a master contacting a slave. |
|  | TTL.write(my\_dat);  Write data to the master node. | Typical usage for a slave responding to the master. |
| flag = OBJ.read(rx\_data[]) | TTL.read(received\_data);  Read data from the master node or from the last slave you contacted (if you’re the master). | Used by nodes to read data from other nodes with restrictions.[[4]](#footnote-4) Master and slave call is the same. Flag passed back from object describes success or failure mode of reading attempt. (see error messages below) |
| OBJ.end() | TTL.end(); | Closes the serial connection and stops broadcasting. |

## Private Functions

A discussion on some of the private functions contained in the library.

Message\_start()

This function searches the serial stream for the start byte sequence (typ. 0x00 0x01). Once it finds that sequence, it returns a 1 for success or it returns 0 (zero) if it can’t find the start byte sequence within the specified number of attempts. Tries are automatically calculated by the program based on the length of the message, in an effort to create a function that is dependent on time rather than attempted read cycles.

CLEANSERIAL()

This function is used by the microcontroller to wipe the information in the serial buffer. It does this by starting and stopping the serial connection. The implementation of this function is specific to the Arduinos and should be modified if you wish to use this function with other microcontrollers.

SETPREAMBLE(), SETHEADDER(), SETDATA(), SETCRC()

All these functions are used to put together pieces of the message[] array for later broadcast.

SERIALBROADCAST()

This function writes out the completed message[] array to the serial port.

### CRC Functions

CRCSlow()

This algorithm performs a cyclic redundancy check (CRC) on 13 bytes of the default 18 byte message. The preamble and the footer are the parts of the message automatically excluded from the CRC. The goal of a CRC is to ensure data validity by proving that the data has not changed while in transit. The proof is constructed through a mathematical operation that is done when the message is constructed and broadcast, and then again at the receiving node when the message is read. Historically, CRCs simply added up all the ones and formed a byte with this result. However, if two bits are flipped in this type of scheme, the CRC yielded the same result. This is called an undetectable CRC error. Today we use a more complex CRC algorithm that is less prone to failures. The basics methods of constructing this function are detailed in Ross Williams’ guide[[5]](#footnote-5). See below for an algorithm comparison with CRCFast().

CRCFast()

This algorithm uses the CRC16Fast\_table() as a lookup table and performs a different CRC process than CRCSlow() with the same goal of providing data validation. It is faster than the CRCSlow() algorithm but provides a less robust validation that is more susceptible to undetected CRC errors. The basics of implementing a CRCFast() function and lookup table can also be found in Ross Williams’ guide.3

CRC16Fast\_table()

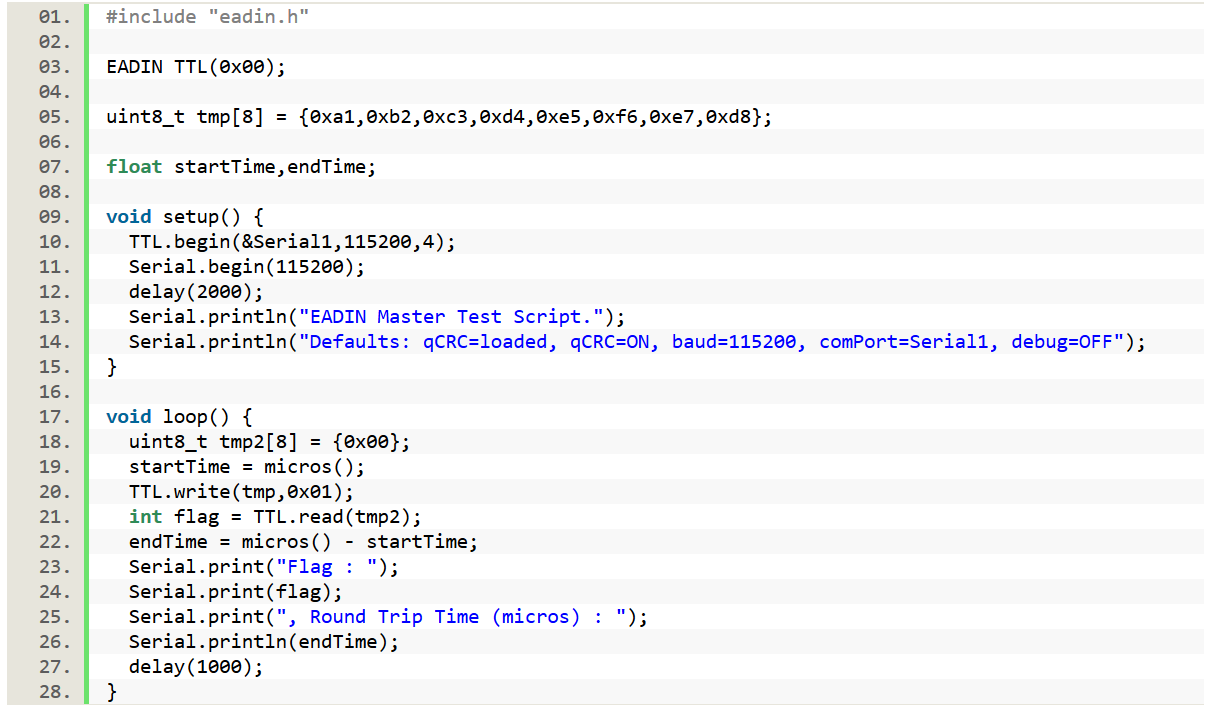
This algorithm computes a lookup table which is used by the CRCFast() function. This table is static and only needs to be computed once per session. The table is a function of the polynomial value specified during object creation. Additionally, the arrays CRCtable1 and CRCtable2 are used to store the output of this function. It is advantageous to load the pre-computed table at bootup because the table should not change and it will make the boot sequence faster. Additionally, there is the possibility for an error during code execution to cause a discrepancy in the table which would decrease the capability of the node to communicate with other nodes.

Algorithms Compared

The CRCSlow() function requires on average 8.5 microseconds per byte on the Arduino Yun. However, the CRCFast function requires only 2.5 microseconds per byte. This difference is significant for hard-real-time systems where data that arrives too late is discarded. However, the CRCSlow function is more robust than the CRCFast function. In single bit flip tests, the CRCFast algorithm failed to detect errors on the order of 1e-7 whereas CRCSlow found not a single error in 2e-8 tests. Thus while CRCSlow() is three times slower than CRCFast(), we estimate it is twenty times more robust for data validation.

# Example Usage

The following examples can also be found in the examples folder of the github download release v2.0.0.0 . The code for the master node is shown below[[6]](#footnote-6). A description of its function follows.



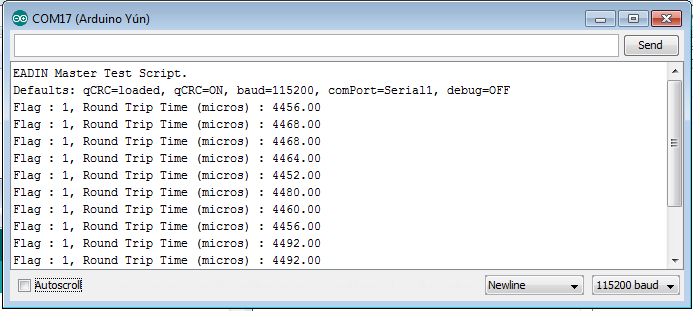
The code above initializes a node as the master node (#0). It then creates a data array tmp[], to store the actual command data that will be sent to the remote node. Some timing variables are created so that the user can see the speed of message communication. Two serial ports are hooked up to this node, one on hardware Serial1, and the second on Serial. The latter is used to print debug output. After setup, the master goes into a loop where it attempts to contact node #1. It writes the command data array tmp[] to node one and then waits for a response. If the response is received in time, the flag variable will be set as 1. Otherwise, there was some error in reading back the response from node #1. The data response from node #1 is saved in the tmp2[] data array. Finally, the result of the test is broadcast to the user over the Serial port.

The matching slave node code is as follows:



The node is initialized as slave node #1 along with an array tmp[] that holds status information to be sent to the master. Array tmp[2] is where command data from the master is stored. The node starts up the Serial1 port and begins listening for the master node in a loop. If a successful read occurs, flag becomes 1, once that happens, the node immediately writes the tmp[] status array back to node #0, the master node. The slave node then goes back into the loop of looking for data from the master.

Sample output from the master node is as follows:



This is showing that the master node received a reply every time it send out a request to the slave node (all flags == 1). Additionally, the amount of time it took from the master formulating the message to receiving a reply from the slave node was around 4460 micro seconds = 4.5 milliseconds. This is for a communication speed of 115200 baud on Serial1.

Note that the two top lines of the output are hard coded into the test script.

# Error Messages

Enabling ‘#define DEBUG’ in eadin.cpp will yield allot of information as you execute the program. By default, this output is sent over Serial port at 115200 baud. This can be changed by editing OBJ.begin() command in the eadin.cpp file.

OBJ.read() produces a flag as an output to describe if a message was successfully read or not. The meaning of each flag value is outlined below:

|  |  |
| --- | --- |
| OBJ.read() returns: | Meaning: |
| 0x00 | Illegal operation: Master is attempting to read message from itself. SOLUTION: Master must write a message to a valid target before it can read a message from that target. Please use OBJ.write() to write to a valid target before attempting to use OBJ.read() command. |
| 0x01 | Successful message read. Data has been stored in the specified byte array for use by the program. |
| 0x02 | Could not find start of message. This will happen often for slave nodes, as they are typically in a loop, attempting to listen to the master’s next command. |
| 0x03 | Incorrect sync string received or no sync string received. |
| 0x04 | Unable to read all the header data, data did not arrive within specified timeout |
| 0x05 | Master read message and request type was not ACK or Slave read message and request type was not ENQ or NAK. Incorrect request type. |
| 0x06 | The message received by the node is not addressed for this node. This will happen allot on bus and wireless networks because every node sees all the messages to other nodes. |
| 0x07 | Data is not from expected source node. Slaves cannot accept data from each other and they must only accept data from the master. Additionally, the Master will only accept data from the node specified in OBJ.read(). If a different nod responds, the master throws the data out. |
| 0x08 | Master failure mode only. Master received a message# and it did not match with expected message# that it sent to node. Node has replied to an old command by the master. |
| 0x09 | Unable to read all the message data & footer, data was not delivered within expected timeout. |
| 0x0a | Cyclic Redundancy Check (CRC) failed |

# Variables

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Description | Type | Default Value (hex) |
| ENQ | Message is an enquiry of status which requires a response. Typically attached to a message by the master controller when it is requesting status of remote node. | Definition / substitution | 0x05 |
| ACK | Message is a response to an enquiry of status and does not require a response. Typically attached to a message | Definition / substitution | 0x04 |
| NAK | Message is a negative acknowledgement, and does not require a response. This is not typically | Definition / substitution | 0x15 |
| PREAMBLE\_L | The number of bytes in the preamble (start and Sync bytes). These bytes are not included in the CRC. | Definition / substitution | N/A |
| HEADDER\_L | The number of bytes in the header. These bytes are included in the CRC. | Definition / substitution | N/A |
| DATA\_L | The number of bytes in the data part of the message, user selectable. These bytes are included in the CRC. | Definition / substitution | N/A |
| FOOTER\_L | The number of bytes in the footer of the message. This is where the CRC is stored. These bytes cannot be included in the CRC calculations. | Definition / substitution | N/A |
| MESSAGE\_L | The overall length of the message which is calculated as the sum total of the preamble, header, data, and footer length. | Uint8\_t | N/A |
| DEBUG | Enables serial printing to help debug the program | True/False | false |
| POLYNOMIAL | The polynomial used by the CRC algorithm | Uint16\_t | 0xC599 |
| BREAK0 | Signifies the start of a message (first byte) | Uint8\_t | 0x00 |
| BREAK1 | Signifies the start of a message (second byte) | Uint8\_t | 0x01 |
| SYNC | Place holder for a byte which will be used to synchronize the clocks of the master and slave node. | Uint8\_t | 0x55 |
| CRCtable1 & CRCtable2 | This table of values is used by the CRCFast() algorithm to speed up the process of creating a CRC. This table is specific to the POLYNOMIAL value. If the POLYNOMIAL value is changed, the CRCtable must be recalculated with the CRC16FAST\_table() function. | Uint8\_t[256] | N/A |
| My\_ID | The identification information for this node. | Uint8\_t | N/A |
| RTS | The pin on the microcontroller used to signal that we are reading or transmitting a message. | Uint8\_t | 4 |
| Rw\_port | The port on the microcontroller that the control network is connected to | HardwareSerial\* | N/A |
| Rw\_speed | The baud rate of the serial communication on the microcontroller. | Unsigned long | N/A |
| Trash | Not in use | Uint8\_t[64] | N/A |
| qCRC | Selects the use of CRCFAST() as opposed to CRCSlow(). Commenting out #define enable\_CRCFast in eadin.cpp will reset qCRC to false, even if it is set as true in the object constructor e.g. EADIN TTL(0x00); | bool | true |
| DEBUG | This is a build option and this it can only be modified by going into eadin.cpp See Error Messages section above |  |  |
| cmd\_only | Setting this to ‘true’ signals that the slave node should not attempt to respond to the master, and that the master is expecting no message back from the slave node. This enables the master to write to individual nodes faster, but the master does not know if the messages were received or not because no response message is sent. | bool | false |
| waitTime & timeOutFactor | These are scaled delay factors that facilitate spacing out the reading and writing of messages. The master cannot expect a message to be immediately available from a slave so the master waits some amount of time, and then attempts to see if the slave’s message has arrived. These scaling factors are used for cmd\_only messages as well to ensure the outgoing buffer does not fill up which will cause messages to fall out and not be sent. | Float | Baud rate dependent |
| Other Temporary function variables | Several functions contain temporary variables that are used for a variety of function specific purposes | Various | Various |

1. Engine Area Distributed Interconnect Network (ADIN), January 31, 2012, Andy Berner, BAE Systems – Electronics & Integrated Solutions, DECWG Phase 1 Requirements Meeting [↑](#footnote-ref-1)
2. Engine Area Distributed Interconnect Network User’s Manual – Version 3.0, April 19, 2012, DECWG [↑](#footnote-ref-2)
3. Slaves are restricted to only write to the master node. [↑](#footnote-ref-3)
4. Slaves are restricted to only read from master node. [↑](#footnote-ref-4)
5. A Painless Guide to CRC Error Detection Algorithms, Ross N. Williams, (http://www.ross.net/crc/download/crc\_v3.txt) [↑](#footnote-ref-5)
6. Thank you to google syntaxhighlighter and [www.planetb.ca](http://www.planetb.ca) for creating pretty syntax highlighting tools [↑](#footnote-ref-6)