ECCO SAFETY GROUP

ECCONet 3.0 Standard

May 16, 2017
Revision 1.3

Contents

1. Purpose

ECCO Safety Group offers vehicle safety systems built with a mixture of components from the various ESG product lines. This gives customers more flexibility in designing their systems.

Communication is an important opportunity to make safety products that are more intelligent, utilizing sensors from all over the vehicle. ECCONet 3.0 allows ESG products to cooperate as peers, without a central controller and without specific foreknowledge of peer functionality.

A single communication wire shared by many diverse products might seem like a recipe for a complicated system. The purpose of ECCONet is to facilitate product interoperability, and to providing a simple, easy-to-follow protocol with standard libraries in order to accelerate time to market.

2. Overview

ECCONet 3.0 is a holistic solution for binding the operations of separate safety devices into a singular functional system.

In order to clarify this, envision each connected safety device as part of a single running computer program. Each device runs on its own thread but shares common global variables with the others.

Figure 1 shows a set of safety components on the can bus. ECCONet plays two separate roles in unifying the system. First, it standardizes the logic, names, and value ranges for the variables that represent individual device inputs, outputs and features. Second, it seamlessly binds such variables across the network to act as if all devices were running on the same hardware platform.

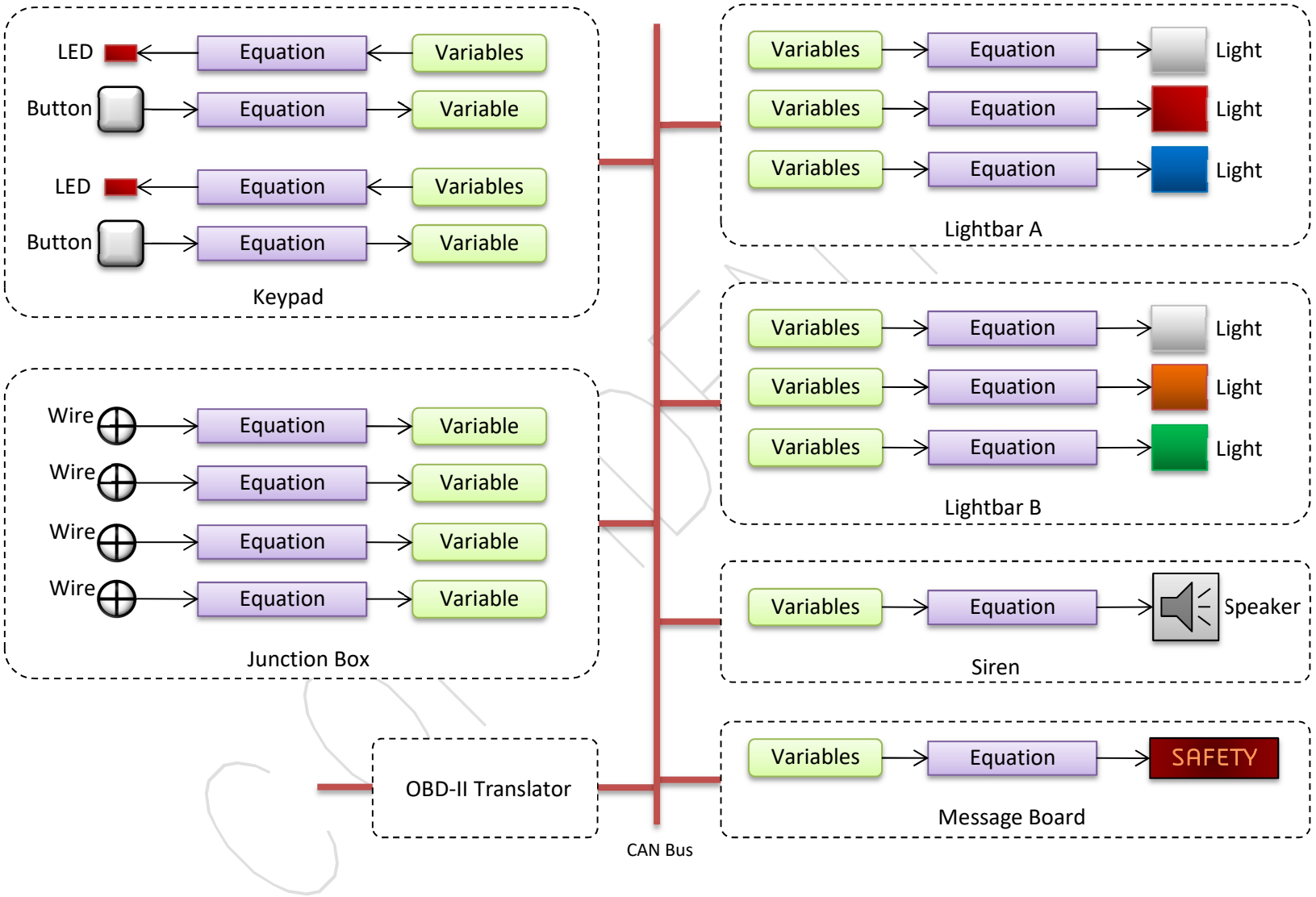

Figure 1 ESG Safety Equipment System

2.1 Variables

Variables represent the components and features of the system. Variables can also represent intermediate values that are useful to simplify functional equations.

2.1.1 Scope

ECCONet variables have either local or global scope. Local variables are used at the discretion of the product designer and are not visible to other devices on the network. The programming terms *private* and *public* are synonymous with local and global, respectively.

2.1.2 Classification

Variables are classified into three general types:

2.1.2.1 Input Status

Input status variables reflect the state of system inputs such as single wires and keypad buttons.

2.1.2.2 Output Status

Output status variables reflect the state of system outputs such as lights, sirens and message displays. Button indicator LEDs are also classified as outputs.

2.1.2.3 Command

Command variables bypass behavioral equations and directly control device outputs and token sequencers.

2.1.3 Name Enumeration vs. Strings

For network efficiency, variable names are conveyed as 16-bit unsigned enumerations instead of strings. ECCO Safety Group maintains a list of such enumerations. There are approximately 25,000 enumerations available.

2.1.4 Classification Prefixes

Many system features are related to all three variable classifications. The named lights are a good example. In order to simplify the enumeration table, the upper 3 bits of each variable enumeration indicate the classification. See Figure 2.

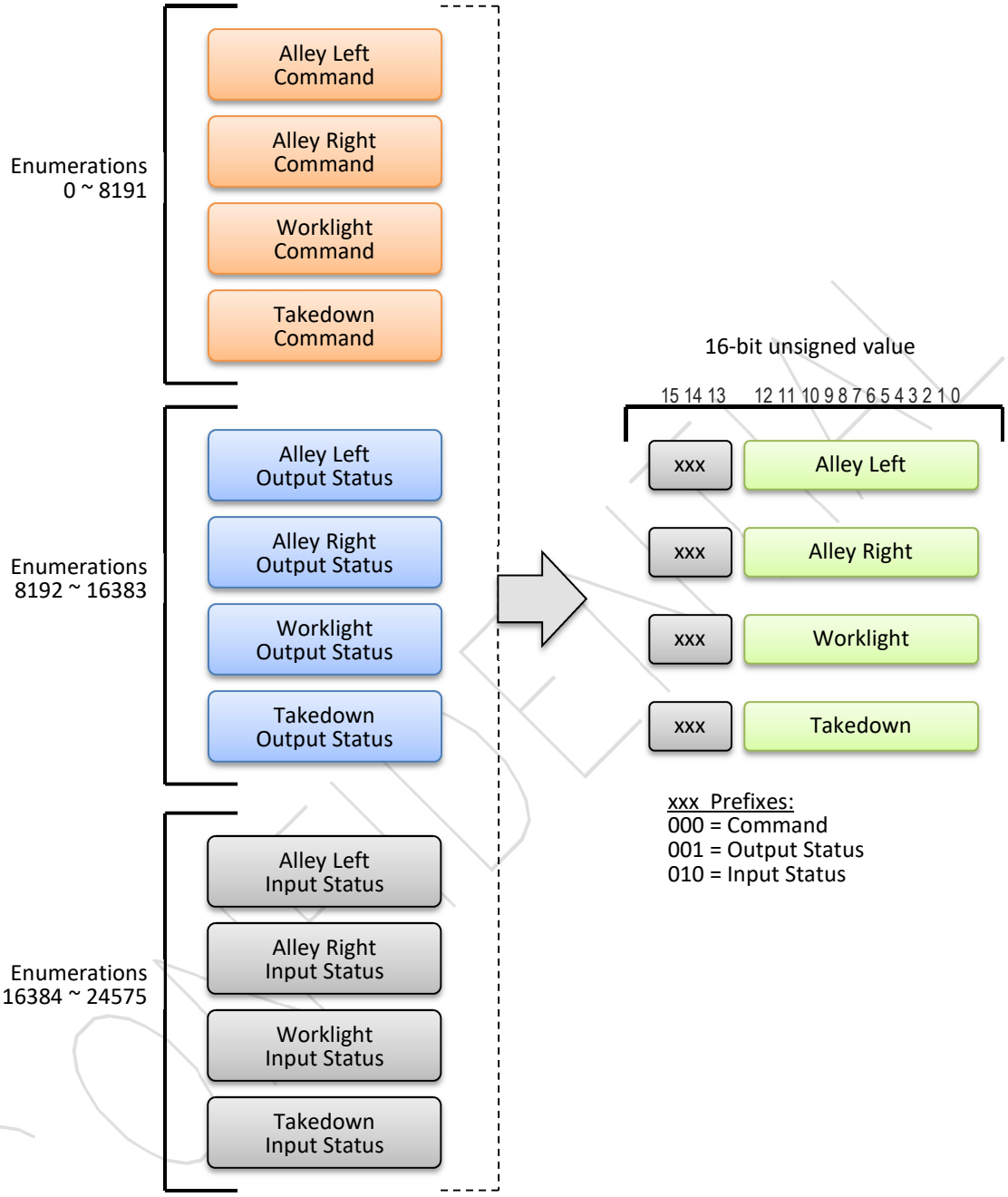

Figure 2 Variable Classification Prefixes

2.1.5 Enumeration Regions

Variable enumerations are divided up into regions. See Figure 3.

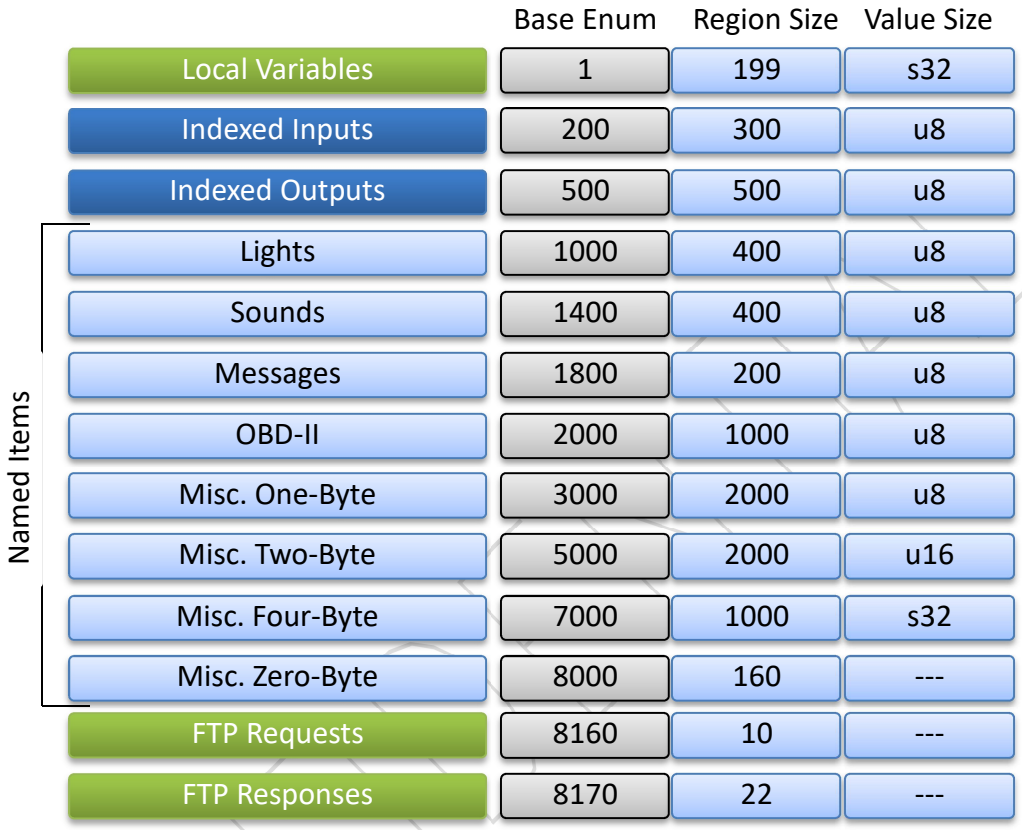

Figure 3 Value Name Enumeration Regions

2.1.5.1 Private Variables

Region 1~199 is reserved for private local variables, and may be used for any purpose. Local variables are 32-bit signed integers.

2.1.5.2 Public Indexed Input Array

Region 200~499 is reserved for public indexed arrays of non-descript 8-bit inputs. An example would be a graphics-based input device with arrays of user controls.

Device firmware maps the enumerations to specific inputs. Although 8 bits are specified, the typical input is a Boolean value.

2.1.5.3 Public Indexed Output Array

Region 500~999 is reserved for public indexed arrays of non-descript 8-bit outputs. An example would be an array of light heads used for patterns.

Device firmware maps the keys to specific outputs. These outputs must be responsive to commands tokens.

Smart light heads that have a component failure warning should issue output status tokens with the corresponding enumerations and an 8-bit failure code. Such tokens should only be issued when a failure occurs, and each no more often than once per second.

2.1.5.4 Named Items

The remaining enumeration space is allocated to named items and FTP requests.

2.1.6 Values

2.1.6.1 Common Ranges

The common value ranges for relative light intensity and acoustic intensity is 0~100.

2.1.6.2 Sizes

For network efficiency, the enumeration regions are assigned a value size of 0, 1, 2, or 4 bytes. Internally all variables are signed 32-bit values, however the designer should keep in mind the actual value size when performing calculations.

2.1.7 Guidance On Enumerated Variables

Product designers should follow these four conventions when enumerating variables:

a. Designers should use existing enumerations when applicable.
b. Designers should request new enumerations for new features, rather than make substitutions.
c. If there is no requirement to name the individual elements in an array of components or patterns, then designers should use the existing indexed enumerations.
d. Designers should group keys in numeric succession when possible to enhance token compression.

✍ *The initial variable database is considered incomplete until all product stakeholders have reviewed and requested their required enumerations.*

2.2 Patterns

A unique variable name enumeration is used for each token sequencer, *not the patterns themselves.* While this might seem inconsistent with the other types of arrays, keep in mind that the *component* being represented by the variable is a sequencer, not a pattern table.

ESG maintains a separate enumeration for patterns. So for example, *TokenSequencer0* is a variable name, and the desired pattern enumeration is its value. In this way, the pattern to be sequenced can be the result of a calculation.

2.2.1 Enumeration Regions

The pattern enumeration is divided into regions for lightbars, Safety Director, and so on. The first 25% of each region is reserved for *nondescript indexed* patterns that are typically created by GUI.

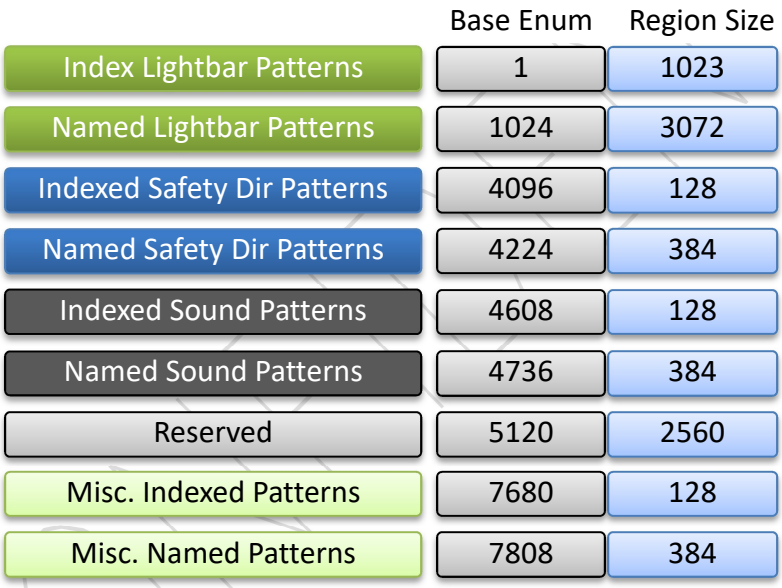

Figure 4 Pattern Enumeration Regions

2.2.2 Enumeration Zero

Pattern enumeration zero (0) is reserved to stop a token sequencer at a given address.

2.2.3 Building Pattern Tables

Figure 5 shows the ECCONet standardized pattern table format. *Notes*:

1. All pattern tables start with a 16-bit value indicating the number of patterns.
2. A pattern start with repeats = 0 runs indefinitely.
3. The first pattern step must be the All-Off, or a normal timed step with same function.
4. Pattern step tokens may be variables of 0~4 bytes.
5. Pattern step token compression is optional (sequencers use the network codec).

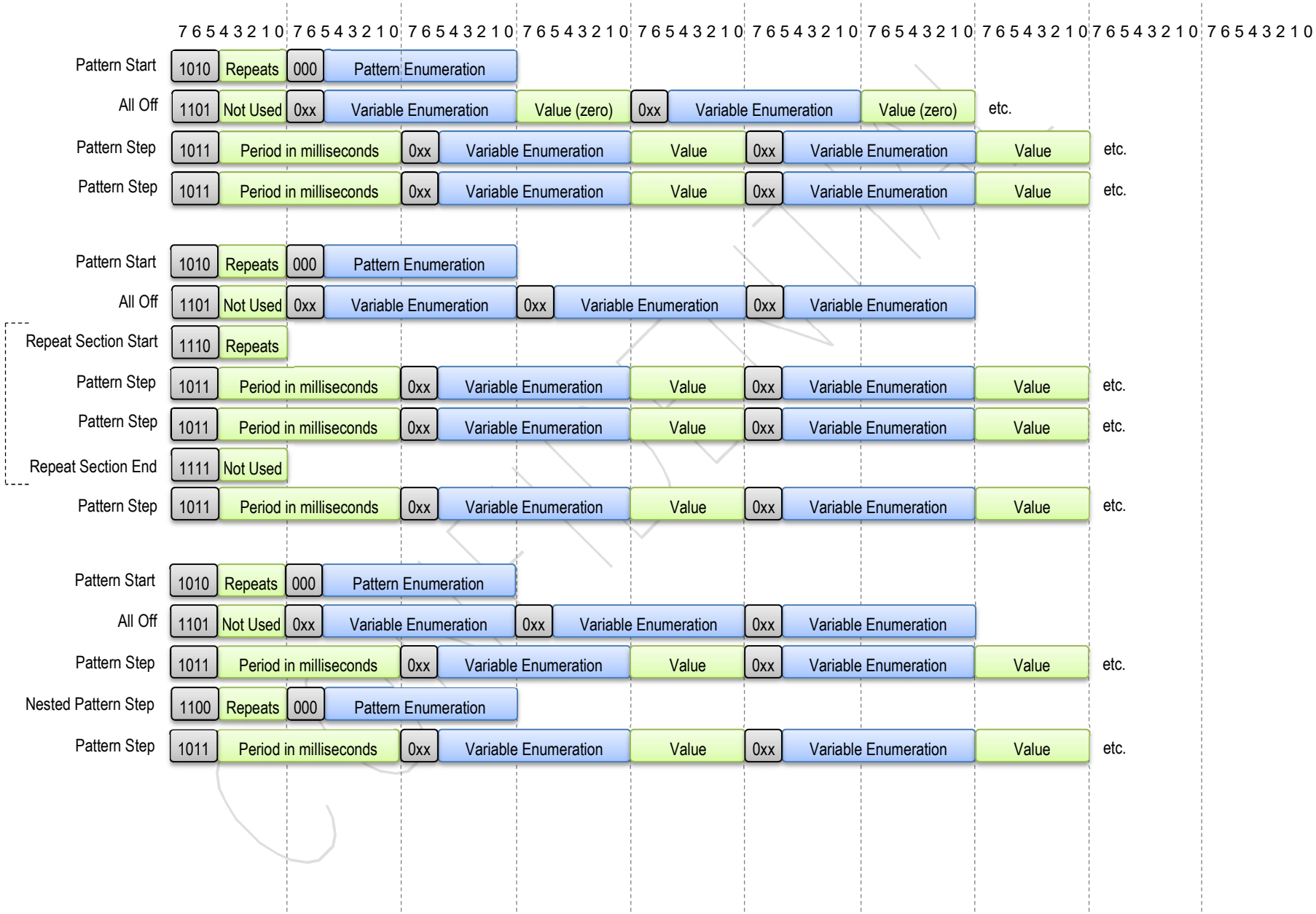

Figure 5 Pattern Table Format

2.3 Equations

The equations shown in Figure 1 define the behavior of the system. Equations may be used to logically combine several inputs into a single pattern output, to scale output intensity based on a system state, to convert a momentary button into a toggle, and so on.

2.3.1 General Format

The equations are plain-text and use the same operators and syntax as the C-language, with a few notable differences:

1. All equations start with the '$' character and end with the ';' character.
2. The *left-hand expression* with one or more variables is evaluated and the result is stored in a single variable in the *right-hand expression*.
3. Variables are expressed as '[nnn]' where nnn is the variable enumeration.

As an example, this equation controls an Alley Left light:

$((0 != [17390]) && (0 != [0x4dae])) * 100 = [9198];-c@

Where: 17390 is KeyLightAlleyLeft enumeration | InputStatus prefix
0x4dae is KeySystemPowerState enumeration | InputStatus prefix
9199 is KeyLightAlleyLeftt enumeration | OutputStatus prefix

2.3.2 Variable Address Option

The variables of an equation may be qualified with a device address using the -a option.

For equation inputs, specifying the network address is useful if static addresses have been programmed for certain devices. If the address is specified, then the variable is only bound to the corresponding output variable of the device with the specified address.

Static device addresses should only be used in certain cases where there is no other logical means to provide the desired system behavior.

For equation outputs, specifying the address is useful to indicate which token sequencer is to be used (sequencers are discussed further on).

As an example, this equation controls a Token Sequencer 0:

$[12345] = [13192-a121];-c@

Where: 13192 is the KeyTokenSequencerPattern enumeration | OutputStatus prefix
-a121 is the token sequencer 0 local address

2.3.3 Time-Logic Unit

Each expressed equation is part of a time-logic unit, which has a few extra features that are primarily intended for keypads and other input devices.

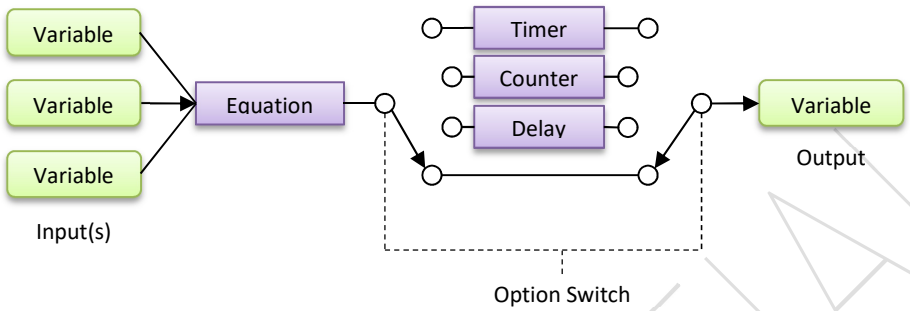

Figure 6 Time-Logic Unit

2.3.4 Output Options

Output options allow a designer to specify the time-logic unit configuration, and when an equations output token should be sent to the application and system. While it might seem obvious to send a token when the result changes, a designer may choose to only have a token output on an output value increase, and no output token for intermediate variables.

2.3.4.1 Logic Options

-am**xxx** Activity monitor. A Boolean output, high when an input token is received more often than once per **xxxx** milliseconds, else low. Example: $[1234]=[4567];-am2000

-rctr**xxx** Rising-edge up-counter. When the equation result changes from zero to non-zero, increment the counter. If the counter reaches **xxxx**, the reset the counter to zero. Example: $[1234]=[4567];-rctr4

-fctr**xxx** Falling-edge up-counter. Same as rising edge up-counter, but increments when the equation result changes from non-zero zero to zero.

-rt Rising-edge toggle. When the equation result changes from zero to non-zero, toggle the Boolean output. Example: $[1234]=[4567];-rt

-ft Falling-edge toggle. Same as rising-edge toggle, but toggles when the equation result changes from non-zero to zero.

-r[**xxx**]skt Rising-edge skip toggle. When the equation result changes from zero to non-zero, request that the specified [**xxx**] time-logic output variable skip the next toggle or up-count. Example: $[1234]=[4567];-r[8899]skt

-r[**xxx**]skt Falling-edge skip toggle. Same as the rising-edge skip toggle, but makes the skip-toggle request when the equation result changes from non-zero to zero.

-rc[**xxx**] Rising-edge time-logic output variable clear. When the equation result changes from zero to non-zero, clears the specified [**xxx**] time-logic unit output variable. Example: $[1234]=[4567];-rc[8899]

-fc[**xxx**] Falling-edge time-logic output variable clear. When the equation result changes from non-zero to zero, clears the specified [**xxx**] time-logic unit output variable.

-rd**xxx** Rising-edge delay. A Boolean output that changes from 0 to 1 the specified time [**xxx**] after the equation result changes from zero to non-zero. When the equation result changes from non-zero to zero, the Boolean output changes from 0 to 1 immediately. Example: $[1234]=[4567];-rd[8899]

-fc**xxx** Falling-edge delay. Same as the rising-edge delay, but delays the falling edge transition.

2.3.4.2 Token Send Options

The output option @ indicates that when the output changes as specified below, a token should be sent to the application and to the network. Although these options include rising and falling by a specified amount, a better solution for rapidly-changing sensors is to use feedback hysteresis in the equation itself.

-c@ Send a token on output change.

-r@ Send a token on output rising >= 1.

-r@ Send a token on output falling >= 1.

-r**xxx**@ Send a token on output rising >= **xxx**.

-f**xxx**@ Send a token on output falling >= **xxx**.

2.3.5 All System Components Use Equations

The full extent of the equation format is designed to support the needs of *both* input and output devices. Providing a consistent logic platform on both ends of the wire is important for successful product interoperability.

However, custom light bar programming via GUI will most often require only a small subset of the available features. Keypad equation programming will likely be done at the time the product firmware is written, and may never be reprogrammed other than via firmware updates.

2.4 Public Variable Synchronization

2.4.1 Synchronization Rules

2.4.1.1 Left-hand Expression Variables

Left-hand expression public variables are synchronized to the same-named right-hand expression variables from other devices. Private variables are not synchronized.

2.4.1.2 Right-hand Expression Variables

Right-hand expression public variables that are categorized as *input status* are synchronized to the same-named right-hand expression variables from other devices. This facilitates the synchronization of user controls.

2.4.2 Tokens

ECCONet 3.0 tokens encapsulate variables for transport within each operating system and across the network. The token contains a key-value pair, which is simply the enumerated variable name and value, where the *key* is the enumeration. Internally, the token key is a 16-bit unsigned integer, and the token value is a 32-bit signed integer. The pair value size is large enough to hold all of the region value sizes listed above.

The token also contains an address field. When used within the application, the address field is the address of system resources, such as token sequencers and the equation processor. When used across the network, incoming tokens hold the address of the sender; outgoing tokens hold the address of the recipient. In this way, the application can know whether a token was created either by a local resource or an external device.

2.4.3 Event Order

ECCONet libraries track the order of *input status* events so that in the case of conflicting input variables, the most recently-changed variable prevails.

2.5 System Logic Conventions

2.5.1 Virtual User Controls

In order to keep logic simple and consistent, virtual user controls mimic their root physical form to convey events and status, and stay in sync with other same-key controls.

Consider Figure 7, where a touch-screen button and flip-flop together emulate an old-school physical light switch. The button and light will not be on the same PCB, but instead will be connected via CAN. We must now decide on which PCB to locate the flip flop.

Unless the CAN connection is between the flip-flop and the light, periodic status updates are of no value. In other words, we can immediately broadcast state changes A, B or C, but can only periodically convey state D. For a light PCB that lost messages or even temporarily lost power, state D would be good a good thing to know.

Moreover, consider the case where the same button controls a second light on yet another CAN node. Having just one flip-flop ensures that both lights will be in phase.

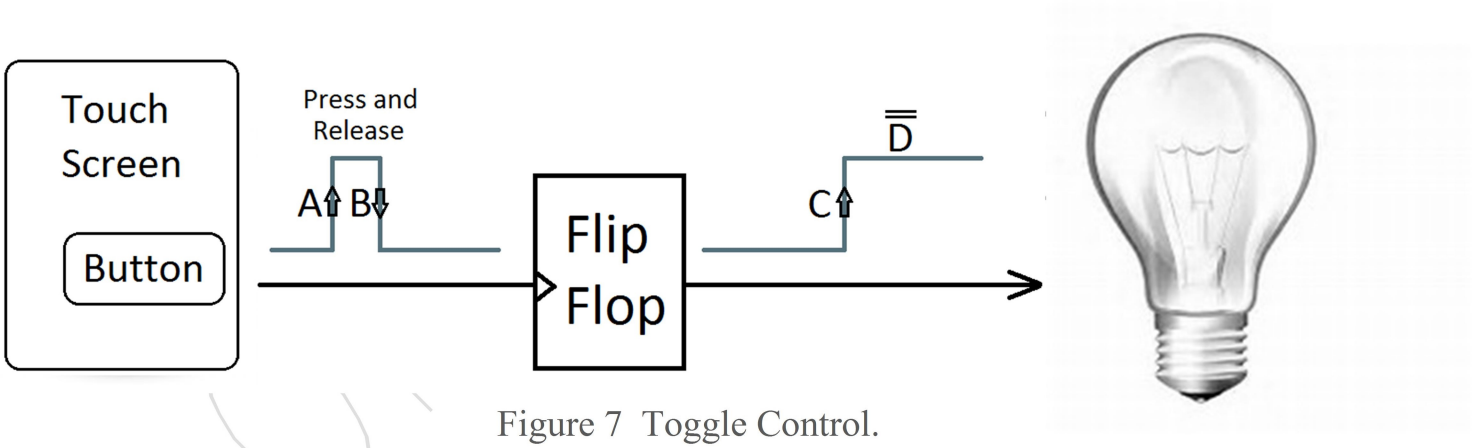

Figure 7 Toggle Control.

The user *expects* the button to work like an old-school switch, and just as reliably. A small delay would probably be forgiven (where message redundancy covered up a corrupt message error) but the user would not be happy to have to hit the button twice to get the light to come on.

In contrast, a *momentary* control's state is only preserved by user intent. When the control's press rate is faster than the periodic status update rate, the user would probably forgive a missed press or two without an afterthought. For example, a user is advancing through available flash patterns.

2.5.2 Virtual User Control Synchronization

Unlike physical controls, virtual user controls have the advantage of programmatic state changes. When two virtual controls exist in the system with the same token key identifier, a user changing one control will change the state of the other as well.

> ✍ *A control whose state is changed programmatically in response to an event or status broadcast does not re-broadcast such a change.*

2.6 User Control Indicators

System designers choose for themselves how to represent a user control state, either with a LED indicator or on graphic displays with different state images.

In the case of a single LED per user control, designers must decide whether the LED represents the state of the control, or the *state of the output device*. This choice is subjective, as explained next.

Suppose for example that a system has been programmed to use two inputs to enable an alley light; a toggle button input *and* a door not-closed switch. Should the button indicator be lit when the feature is on and ready, or only when the alley light is actually on? What would the user want?

In another example, the system has two identical light bars. The system designer wants the button's LED to provide *conclusive feedback* that the light bars are indeed running a pattern. However, should the indicator be lit if just one light bar is responding?

Such closed-loop/open-loop decisions are left to the system designer. ECCONet simply ensures that all devices are capable of either topology.

3. ECCONet 3.0 C Library and C# Communication Stack

3.1 Overview

A standard C99 library is provided for ECCONet implementation, and is shown in Figure 8. The library includes the following modules:

1. A protocol message composer with CAN frame output buffer
2. A protocol message decomposer with CAN frame input buffer
3. A token compressor/decompressor (Codec) for messages and light patterns
4. An FTP Server and FTP Client
5. A Flash drive file system
6. An equation processor
7. Four token pattern sequencers with ability to sequence any kind of token, including nested patterns

The library provides continuity among the 3.0 devices and is a tool for accelerated product implementation.

3.1.1 Interface

The library is loosely-coupled to the application via a constant table of method callbacks, and has no direct interface to the device hardware. The application first resets the library, and then clocks the library with the current system time in milliseconds. The library requires a stack that is at least 512 bytes deep.

3.1.2 Non-library Devices

Nothing would prevent a non-library device or partial-library device from being a good ECCONet 3.0 citizen, provided the logic and communication protocols detailed in this document are adhered to. The library provides a straight-through path from application to CAN port.

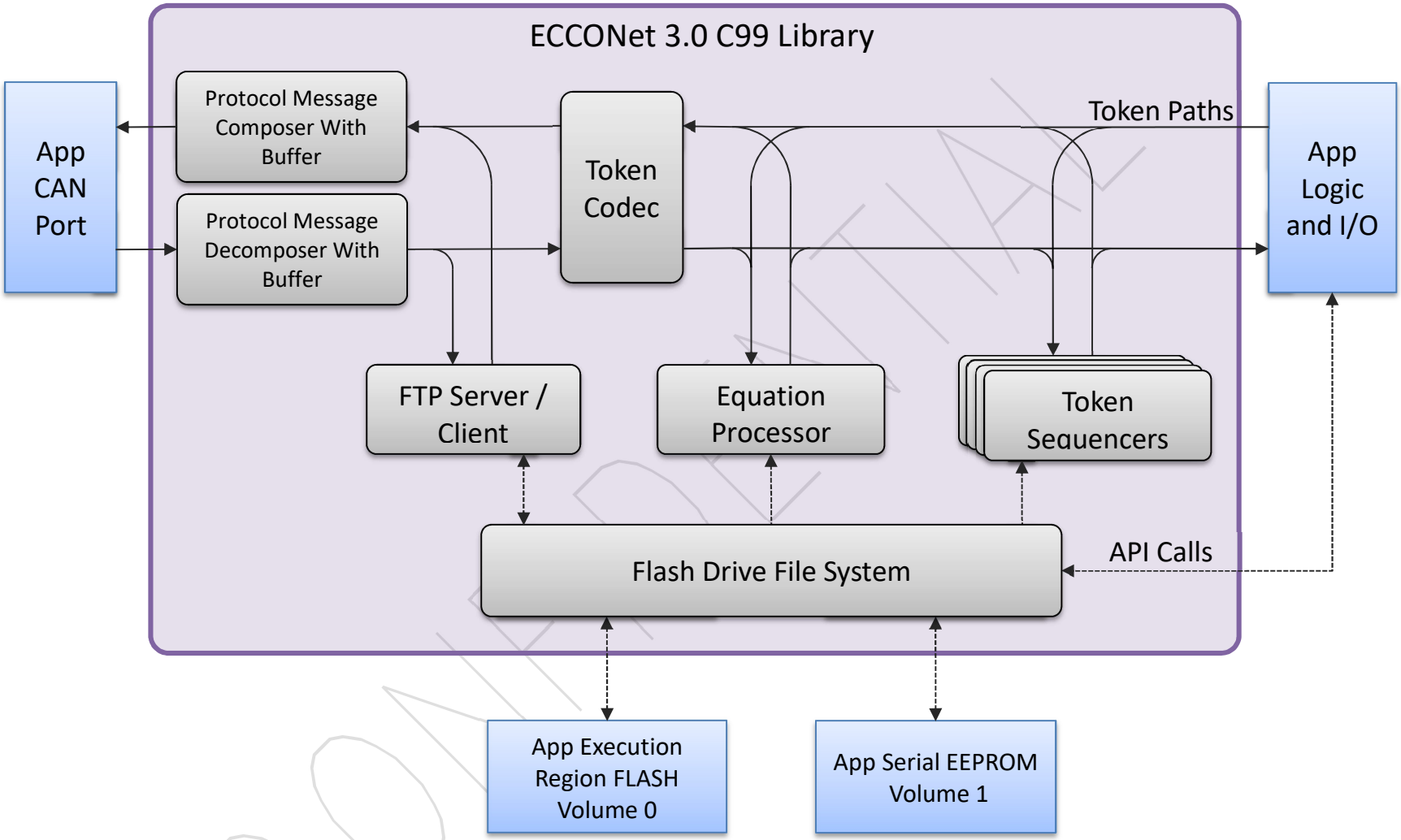

Figure 8 ECCONet 3.0 C99 Library And Application Interface

3.2 ECCONet C Library Equation Processor

The Equation Processor complies with the equation specifications on page 12. The processor continuously updates all equation outputs. In addition to the equation-specified token output options, the processor synchronizes the public variables as specified on page 15 approximately once per second.

3.2.1 File Names and Local Network Addresses

The processor takes an 8.3 text file named "equation.txt" in flash drive volume 0 as input. The processor outputs to local address 125.

3.2.2 Equation Processor Resources

The equation processor can track 50 variables, and for each equation calculation has operator and operand stacks of 20 each. If required, these capacities may be increased based on the hardware platform memory capacity.

3.3 ECCONet C Library Codec

The Codec compresses and decompresses message tokens per the communication specification herein. Token compression is desirable for the network traffic, and is desirable for minimizing token pattern code space.

The Codec is used by the token sequencers in case the token pattern tables were compressed at compile time. In the cases where indexed outputs are in the pattern table, compression is optimal. The Series 12+ could realize a 10:1 compression ratio.

3.4 ECCONet C Library Token Sequencers

The ECCONet C library contains an array of 4 token sequencers that comply with the pattern specifications on page 10.

3.4.1 File Names and Local Network Addresses

Each sequencer runs independently using its own 8.3 named token pattern file, "patterns.tb0", "patterns.tb1", "patterns.tb2" and "patterns.tb3" located on flash drive volume 0. The sequencers 0~3 output to local address 121 through 124, respectively.

3.4.2 Embedded Pattern Cascading

Sequencers 1~3 provide their lower-ordered neighbor with the list of All-Off tokens for which the neighbor should not sequence. In other works, sequencer 0 is prevented from sequencing the tokens currently being sequenced by sequencer 1, as specified by sequencer 1's All-Off pattern step entry. This feature allows an array of lights to display an overall pattern while displaying an embedded pattern using just a portion of the lights.

3.5 ECCONet C Library FTP Server and FTP Client

The ECCONet C library contains both an FTP server and client. The ECCONet file transfer protocol is proprietary, and optimized for the CAN bus. Appendix C shows the FTP protocol request and reply format.

The FTP server and client are both capable of getting file info, reading, and writing 8.3 named files in the device flash memory. The client portion allows a smart controller to read the files of a lightbar, providing the capability for an adaptable field control GUI.

During the time when an FTP server or client is connected, the CAN receiver of the message decomposer filters only those messages related to the file transfer. Both the client and server have a no-response timeout, and the decomposer filter has a timeout as well.

3.6 ECCONet C Library Flash Drive File System

The ECCONet C library contains a FLASH drive file system with up to 3 separate drive volumes. Behavioral files and pattern files are always stored in the application execution memory region volume 0.

The flash drive file system makes API calls back into the application for raw reads and writes, and to get the volume for a given file name.

The file system attempts to level-wear the flash memory. Files that are re-written are marked as erased, and the flash continues to fill top-down and bottom-up. When the flash is full, the file system defragments the volume and erases unused space.

The application designer may wish to automatically cache flash writes for a few hundred milliseconds, or until a sector boundary is crossed. This caching would prevent unneeded flash wear during defragmentation.

3.7 ECCONet C Library Protocol Message Composer and Decomposer

The ECCONet C library contains both a protocol message composer and decomposer, each with separate buffers. The message composer has a 40-frame buffer, and the decomposer has a 20 frame asynchronous buffer and 80-frame synchronous buffer.

The decomposer sorts out-of-order packets and interleaved packet streams from multiple sources, and reproduces coherent messages. The decomposer then routes the messages to the FTP server, the FTP client, or the Codec.

The composer and decomposer in tandem cooperate to manage the system event index. The event index is used to ensure that the most recent input event prevails when there are conflicting input statuses.

3.8 Command Token Pass-Through

Per the ECCONet 3.0 specification, the C library passes command tokens directly to the application. Such tokens are used to directly control system resources, to request hardware and firmware revision, to place the system in firmware update mode, and so on. In order to be ECCONet 3.0 compliant, the application must process such tokens and execute the given commands. The list of commands is included in the variable enumeration.

3.9 ECCONet 3.0 #C Communications Stack

A standard C# communications stack is provided for ECCONet GUI implementation that works in conjunction with the ECCONet C99 library. The stack implements the FTP client protocol and USB-to-CAN interface. The stack provides an API for reading and writing files to devices at known addresses.

The stack also provides a list of online devices and their CAN addresses. This feature is dependent on the device firmware application being responsive to requests for device descriptions and firmware revisions. This feature is independent of the ECCONet 3.0 C99 library.

Using the FTP client, the GUI can write files for behavioral logic, flash patterns, static addresses, and any other files for which the device has room. Eventually such files may include GUI hints for a new generation of system controllers.

4. Network Protocol

In order to keep ECCONet simple and extensible, all nodes adhere to this *protocol* and *convention*:

- All nodes set their own virtual address unless programmed to be static.
- All nodes broadcast any new input event, and periodically broadcast their status.
- All nodes track the new event order, and input events increment the order.
- All nodes normally only use events and status to actuate outputs and patterns.
- All virtual user controls mimic their root physical form to convey events and status, and sync with other same-key controls.
- All nodes convey information with key-value pairs, using keys that are globally unique and maintained by ESG, and values with common regions.
- All devices use message compression when communicating via CAN.

Each of these tenets is explained in more detail further on.

4.1 Network Terminology

In this document, a *device* is a node on the network. Devices can be generally thought of as input devices, output devices, or hybrids (such as translators to other networks).

Input device refers to dedicated keypads, touch-screen keypads, wire inputs such as the J-Box, and the like. *Output device* refers to lightbars, beacons, message signs, sirens, speakers, wire outputs such as the J-Box, and the like.

4.2 Physical CAN Bus

The ECCONet CAN bus is based on ISO 11898-2 standard. The standard defines up to 40 m total bus length, up to 120 nodes, and up to 1 Mbps data rate. ECCONet has its roots in HazCAN and thus operates at 125 kbps with extended frame format. The electrical and timing parameters are listed in Appendix A.

A physical CAN + Power wiring connector is currently not specified for ECCONet. Standardizing on just one or two standard connectors would benefit the device manufactures, the installers and third-party OEMs.

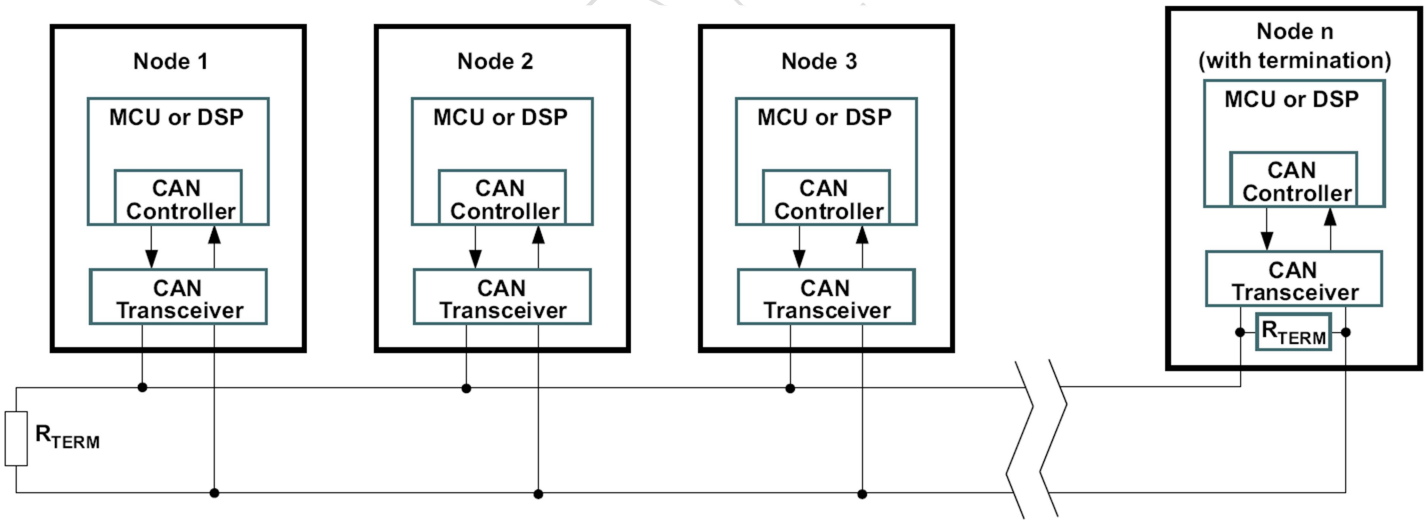

Figure 9 Typical CAN Bus

4.3 Message Routing

ECCONet nodes most frequently use one-to-many broadcasts, and very infrequently use one-to-one messages.

4.3.1 One-to-Many Broadcasts

One-to-many messages are used to broadcast events, status messages and commands. In

order to improve system reliability, redundancy is used. As described further on, all devices broadcast a new event, and periodically broadcast their status.

System designers *may* choose to augment redundancy by closing the control loop; only changing an indicator upon change of an associated output. There are caveats with that strategy, however, which are discussed on page 16.

In addition, ESG can provide installers with a *statistical ping tool* to ensure good wiring in potentially noisy environments.

4.3.2 One-to-One Messages

One-to-one messages are used to send a command to a single device and to respond to a query for specific device information.

4.3.3 One-to-One Connections

One-to-one connections are used to transfer files to and from the system devices.

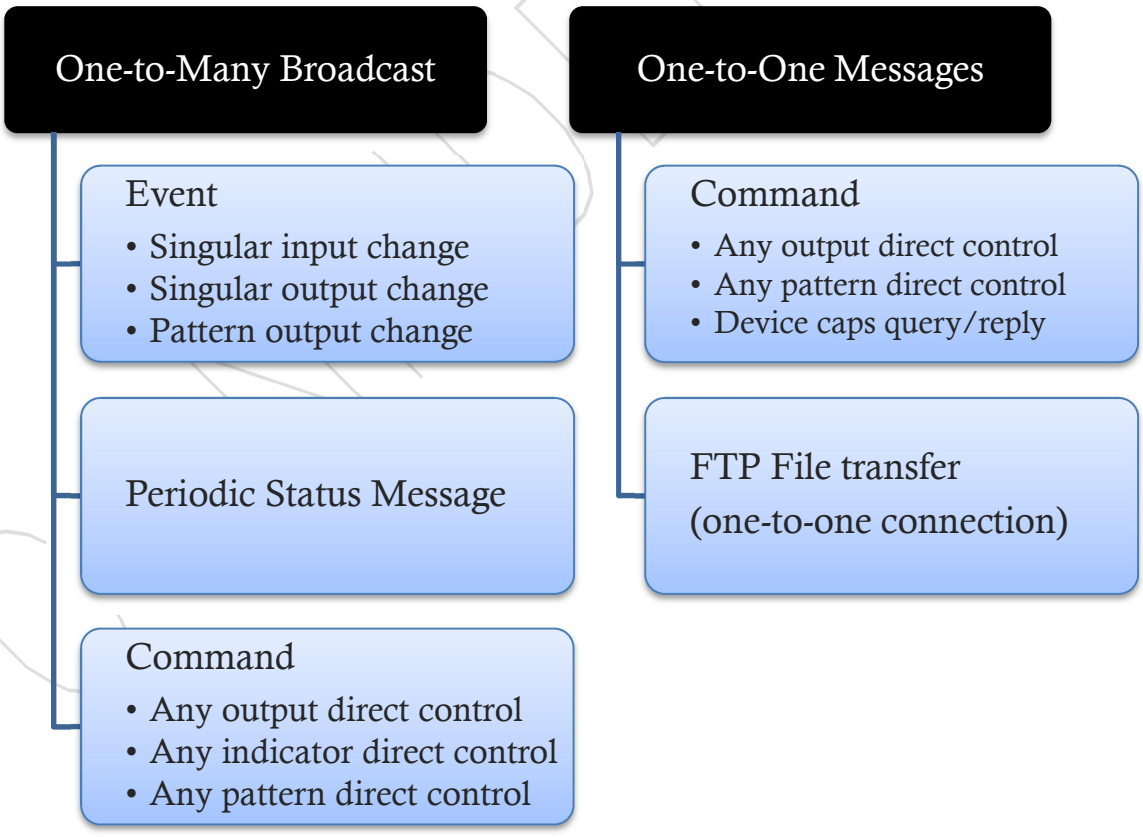

Figure 10 Message Routing and Types

For Ethernet and Wi-Fi transports, the ECCONet 3.0 one-to-many and one-to-one protocols are UDP and TCP, respectively.

4.4 Message Types

ECCONet messages are generally one of three types:

4.4.1 Event Broadcasts

All devices *immediately* broadcast a change in any of these applicable states:

a. Overall state (system good, or error code)
b. Discreet wire inputs
c. User controls
d. Sensors, including those for sensing hardware error conditions
e. Audio output devices
f. Individually-controlled lights, such as takedown
g. Message displays
h. Pattern sequencers (sequencer state as opposed to sequenced items)

4.4.2 Periodic Status Broadcasts

All devices *periodically* broadcast all of the applicable states from the event broadcast list above. In order to desynchronize broadcast traffic, the broadcast rate is 1 / (1s + (bus address – 60) * 1ms).

4.4.3 Device Commands

Device commands provide direct unconditional control of a device's components, such as message signs, sirens, individual lights, token pattern sequencers and indicators. Device commands are useful for production and test, and for 3rd party OEMs.

Device commands are also used to query a device for model and control information. Such information may be used, for example, to populate the buttons on a future touch-screen controller.

✍ *During normal ECCONet operation, only event and status messages are used to actuate output components and pattern sequencers.*

4.5 ECCONet Message Protocol

The CAN transport provides just 8 maximum payload bytes per packet. The ISO-TP protocol (ISO 15765-2) is commonly layered on CAN to transport messages longer than 8 bytes. However, ISO-TP is not a broadcast protocol and further reduces the payload to seven or fewer bytes per packet.

The ECCONet message protocol was developed to be capable of both broadcast and one-to-one messages, and to include token compression and event management. Figure 11shows the CAN ID field, and the data portion of the protocol is show in Appendix C.

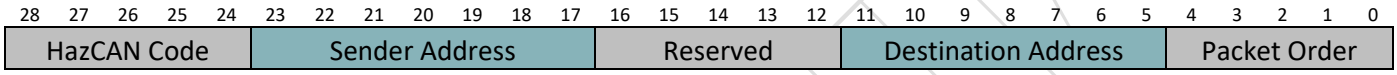

Figure 11 ECCONet Message Protocol CAN ID Field

4.5.1 HazCAN Code

ECCONet message protocol use HazCAN code 0x0D for the last (or only) packet of a message, and code 0x0C for all other packets of the same message.

4.5.2 Packet Order

ECCONet message protocol use a 5-bit rollover up-counter to track packet sequence. Any message with a broken packet sequence is ignored.

4.5.3 Message Checksum

ECCONet message protocol uses a CRC-16-IBM reversed checksum as the last two bytes of any message larger than one CAN frame. Any message with an invalid checksum is ignored.

4.6 ECCONet FTP Protocol

ECCONet uses a proprietary FTP protocol for one-to-one file transfers. The details of the FTP protocol are shown in Appendix C.

4.7 Event Order

All devices keep a local copy of a system-wide 8-bit rollover up-counter *event order variable*, which is included as the first byte of any ECCONet 3.0 message. When a device

senses an input change event, it increments its local copy and then broadcasts the event along with the new order value.

A device will update its local copy when the incoming event order is newer, and will ignore the message when the incoming event order is older than its local copy. Upon reset or boot, a device will set the event order to zero, which is a non-valid value, and wait 200ms before sending a regular event or status message.

> ✍ *The ECCONet 3.0 C99 library manages the event order variable.*
> *Note that only events from input devices (with the input status prefix token key) cause the event order variable to be incremented.*

4.8 Address Assignment

All devices self-assign a virtual bus address in the range of 1~120. Default and alternate addresses are derived from the device's UID. The derivation, along with assignments for reserved addresses 0 and 121~127, is shown in Appendix B.

Upon power-up or reset, a device broadcasts an *address reservation* message to reserve its default address. If a second device has already claimed the address, it responds immediately with an *address in use* broadcast (AIU). If the first device receives a corresponding AIU within 100ms of address reservation, it picks the next alternate address and restarts its address assignment. If the first device does not receive the AIU, then it claims the address and broadcasts its own AIU.

As a failsafe, a device will restart its address assignment if it ever receives a message from its own address.

> ✍ *ECCONet virtual addresses do not convey any metadata. With 120 available addresses and 14 devices on the bus, the likelihood that one of the devices will need to choose an alternate address is 55%.*

4.9 Device Discovery

ECCONet requires no special method for device discovery. All online devices transmit their status every 1120ms maximum. Any device may query another discovered device for specific device information.

4.10 Key-Value Tokens

As discussed on page 15, ECCONet 3.0 uses tokens to keep public variables in sync. A token is a transport wrapper around a *key-value pair,* and also includes fields for *address*.

As an example of a public variable, an amplifier volume knob is a *volume control* (key) that has a *current rotation* (value). A remote amplifier may input the variable into its equations and set the volume accordingly.

Figure 12 The Key-Value Pair

✍ *With a database of common keys and common values, we have an extensible way to share information.*

4.11 Compressed ECCONet

All ESG devices on the CAN bus transmit compressed token messages, and are capable of receiving both compressed and uncompressed messages. The need for compression arises from the relatively low bitrates and small payload sizes of the CAN transport.

Token compression details are shown in Appendix C. Effective compression requires an understanding of the nature of the tokens to be compressed.

4.11.1 Sequential Token Keys

For like groups of inputs or outputs, the ESG key database is preloaded with common sequential assignments. A *binary repeat* and an *analog repeat* token both work by specifying a base key and a number of repeats, and then increment the key for each subsequent value symbol. Thus, in order to achieve best compression, sequential keys should be used when possible.

4.11.2 Binary Repeats

A binary repeat uses binary flags to confer either a zero or a common value to the resulting uncompressed token. Note that the common value may be any type of value; a binary value or 1~4 byte value based on the token key.

4.11.3 Analog Repeats

An analog repeat applies individual scaler values to a series of sequential keys. This is useful for certain lighting pattern sequences and still provides good key compression.

System designers are strongly encouraged to keep compression in mind when adding new key ranges to the database.

4.11.4 Variable Token Value Width

The amount of payload space used to express a token value is dependent on the token key. ECCONet provides values of 0~4 bytes.

- *During normal operation, assuming a data rate of 125 kbps with extended frame format, 14 live nodes each sending 4-packet status messages once per second, basic bus utilization is 5.7%.*
- *The standard API for ECCONet performs the compression and decompression, alleviating designers from implementation details.*

5. Appendices

A. CAN Bus Electrical

Parameters:

a. Up to 40 m total bus length
b. Up to 120 nodes
c. Extended frame format B with 29-bit identifier
d. 125 kbps data rate
e. 8 quanta per bit, 6 before and 2 after sample point

Cable Parameters and Bus Termination:

The ISO 11898-2 standard specifies the interconnect to be a twisted-pair cable (shielded or unshielded) with 120-Ω characteristic impedance (ZO). Resistors equal to the characteristic impedance of the line should be used to terminate both ends of the cable to prevent signal reflections. Unterminated drop lines (stubs) connecting nodes to the bus should be kept as short as possible to minimize signal reflections. The termination may be on the cable or in a node, but if nodes may be removed from the bus the termination must be carefully placed.

B. Address Assignment

UID Bit:	0	0	0	0	0	127	126	125	124	123	122	121	120	119	…	6	5	4	3	2	1	0	
XOR:	0	1	1	0	1	0	0	0	1	1	0	1	0	0		0	1	1	0	1	0	0	Result:
Subtotal:	subresult 18 +							subresult 17 +								subresult 0 =							Device Address

Next alternate value rotates XOR right, ->

1	1	0	1	0	0	0

and processes subresults again.

After 6 rotations, process reverts to default but with each subsequent result incremented by 1,
and after 6 more rotations, process reverts to default but with each subsequent result incremented by 2, and so on.

The default device address is derived by taking the device UID in 7-bit chunks, XORing with a pattern, and performing a 7-bit summation. The next alternate address is derived in the same manner, but with the XOR value rotated left one bit. After 6 rotations, the process restarts with the default address but each result is now incremented by one. After 6 more rotations, the process starts again with the default address but each result is now incremented by two, and so on.

ECCONet Addresses:

Address 0: Broadcast

Address 1~120: ESG Devices

Address 121~124: Token Sequencers

Address 125: Equation Processor

Address 126: PC Access

Address 127: 3rd Party OEM Access

C. Token,Compression, Pattern and Network Data Formats

Token Prefixes			
Prefix			**Token Type**
0	0	0	Command
0	0	1	Output Status
0	1	0	Input Status
0	1	1	Binary Repeat
1	0	0	Analog Repeat
1	0	1	Pattern Sync

Prefix				Pattern Token Type
1	0	1	0	Pattern with Repeats
1	0	1	1	Pattern Step with Period
1	1	0	0	Pattern Step with Repeats of Nested Pattern
1	1	0	1	Pattern Step with All Off
1	1	1	0	Pattern Section Start with Repeats
1	1	1	1	Pattern Section End

Bit positions: 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0

Format								
Uncompressed Token	0 x x	Key	Value Byte 0 (opt.)	Value Byte 1 (opt.)	Value Byte 2 (opt.)	Value Byte 3 (opt.)		
Binary Repeat	0 1 1	Num Keys - 1	0 x x	Base Key	Common Value 0~4 Bytes	Binary Flags 0-7	Binary Flags 8-15	Binary Flags 16-23	Binary Flags 24-31
Analog Repeat	1 0 0	Num Keys - 1	0 x x	Base Key	Key 0 Value	Key 1 Value	Key 2 Value	Key 3 Value	Etc.
Sync Token	1 0 1	Unused	0 0 0	Pattern Enumeration				
Pattern with Repeats	1 0 1 0	Num Repeats	0 0 0	Pattern Enumeration				
Pattern Step with Perdiod	1 0 1 1	Period in milliseconds (1 to 4095)	0 x x	Key 0	Key 0 Value	0 x x	Key 1	Key 1 Value
Pattern Step with Nested	1 1 0 0	Num Repeats	0 0 0	Pattern Enumeration				
Pattern Step with All Off	1 1 0 1	Not Used	0 x x	Key 0	Zero Value	0 x x	Key 1	Zero Value
Pat Section Start	1 1 1 0	Num Repeats						
Pattern Section End	1 1 1 1	Not Used						

Direction	Message	Prefix	Key				
Request:	File Info / File Read Start	0 0 0	KeyRequestFileReadInfo	Null-Terminated File Name			
Response:	File Info / File Read Start	0 0 0	KeyResponseFileReadInfo	Null-Terminated File Name	File Data Size 4 bytes	File Data CRC 2 bytes	
Request:	File Read Segment	0 0 0	KeyRequestFileReadSegment	Segment Index 0..n 2 bytes			
Response:	File Read Segment	0 0 0	KeyResponseFileReadSegment	Segment Index 0..n 2 bytes	File Data 256 bytes max		
Request:	File Write Start	0 0 0	KeyRequestFileWriteInfo	Null-Terminated File Name	File Size 4 bytes	File CRC 2 bytes	Unlock Code 4 bytes
Response:	File Write Start	0 0 0	KeyResponseFileWriteInfo	Null-Terminated File Name			
Request:	File Write Segment	0 0 0	KeyRequestFileWriteSegment	Segment Index 0..n 2 bytes	File Data 256 bytes max		
Response:	File Write Segment	0 0 0	KeyFileWriteSegmentInfo	Segment Index 0..n 2 bytes			

Example Tokens

Indexed Input	0	x	x	Indexed Key	0/1		
Named Light	0	x	x	Named Output Key	Level 0~100		
Named Sound	0	x	x	Named Output Key	Level 0~100		
Indexed 8-bit Output	0	x	x	Indexed Key	Level 0~100		
Indexed 16-bit Output	0	x	x	Indexed Key	Level 0~100	Status	

Device GUID 0	0	x	x	Serial Number Key 0	Byte 0	Byte 1	Byte 2	Byte 3
Device GUID 1	0	x	x	Serial Number Key 1	Byte 4	Byte 5	Byte 6	Byte 7
Device GUID 2	0	x	x	Serial Number Key 2	Byte 8	Byte 9	Byte 10	Byte 11
Device GUID 3	0	x	x	Serial Number Key 3	Byte 12	Byte 13	Byte 14	Byte 15