CMRS\_2560 Theory of Operation

Problem Statement

The Carquinez Model Railroad Society desires the following:

* A modular signaling system that can be developed in stages over a number of years
* A signaling system that is responsive to switch positions and train movements
* A dispatching system that receives feedback from the railroad to show actual position of switches
* Improved reliability of the layout infrastructure with a minimal impact on DCC system performance

Introduction

The CMRS\_2560 system has been designed to be a modular, integrated switch control, train detection, and signaling system for the Carquinez Model Railroad Society. The CMRS\_2560 uses Ethernet enabled microcontrollers for detecting switches and trains and controlling signal indications and switch positions. The system also provides feedback to JMRI via its SimpleServer interface so that microcontrollers can share state information across Ethernet and also send feedback to JMRI to keep dispatcher controls in sync.

JMRI’s SimpleServer is a critical component of the overall system; all CMRS\_2560 devices must be able to connect to the SimpleServer to not only send state information on devices that each microcontroller controls, but also to register to receive updates on state variables of interest. JMRI’s SimpleServer provides filtering to minimize the amount of data that is retransmitted to clients.

Hardware Description

Each CMRS\_2560 “station” consists of an Arduino microcontroller and a set of configurable devices that are attached to the Arduino. The Arduino selected for this application is the Mega 2560, an uprated version of the Arduino Uno R3.[[1]](#footnote-1) The Mega 2560 was selected because of the software complexity to run the “station.” The Mega 2560 has 256k of program memory, which is Read Only Memory (ROM), and 8k of Random Access Memory (RAM) for the dynamic variables. There is also 4k of system programmable ROM where configuration and persistent state data for the “station” is kept. A Serial Monitor interface using a USB cable is available for debugging – interfacing with a Serial Monitor app within the Arduino IDE.

Arduino hardware architecture allows for the addition of “shields” to the system. Shields are daughter boards that provide additional I/O or communications functionality to the Arduino. Most CMRS\_2560 “stations” have two shields added to the Mega 2560 controller board. The first shield is a “Grove” shield that breaks out certain I/O pins from the Arduino with a modular 4 pin Grove connector (see below). The second shield is a W5100 Ethernet shield, which provides both physical Ethernet connectivity to the station, but also a microSD card interface that is used as part of the “station” configuration process.

The Grove shield provides a connectorized breakout of the Inter-IC Communications (I2C) serial bus from the Arduino that is used to communicate to all of the modular boards. This breakout is much more cost effective than using a screw terminal breakout shield (there are more than 50 digital and analog I/O pins on a Mega 2560) and far more reliable than simply sticking a wire into edge sockets of the Mega 2560.

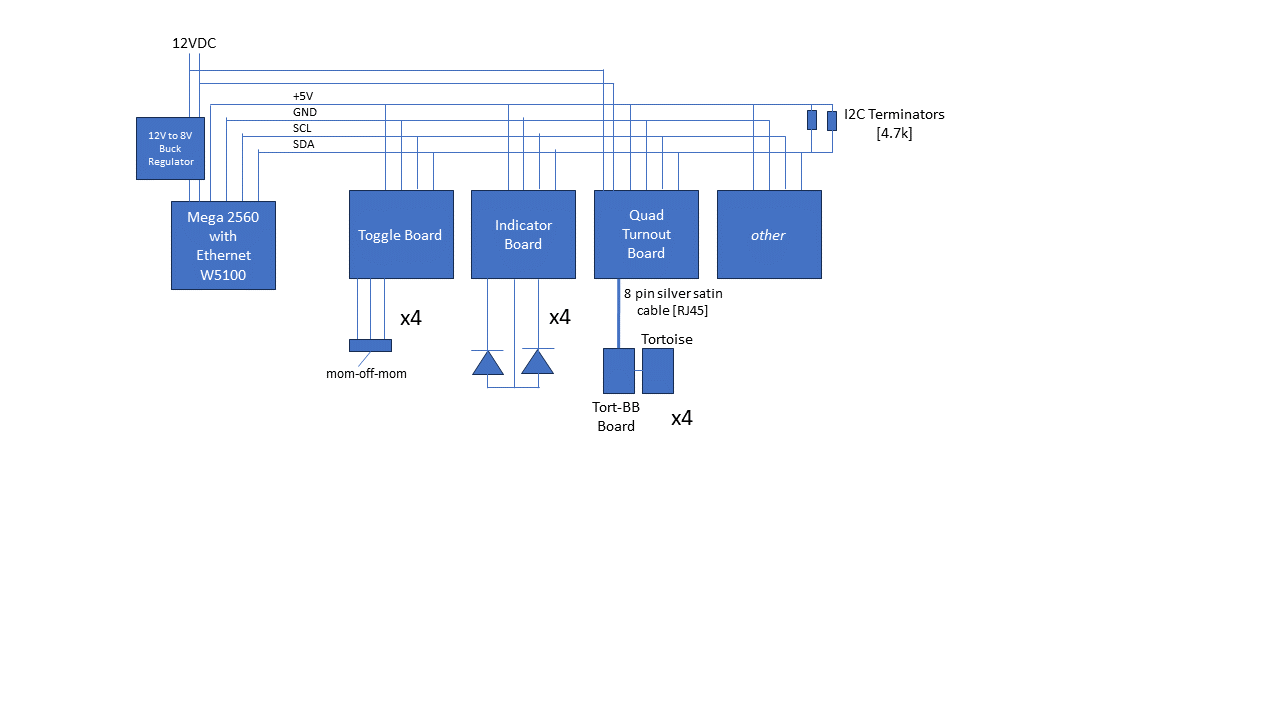


Figure 1: Basic Architecture of the CMRS\_2560 with Arduino, I2C Bus, and custom hardware with typical interface to hardware on the layout.

I2C Hardware Bus

The I2C bus is a bidirectional data bus that was originally designed for serial communications between IC’s across a printed circuit board. Arduino and other microcontroller manufacturers have provided I2C as a short distance serial bus that provides access to a plethora of I/O, sensor, and control IC’s that are integrated with other hardware to provide small sensor and controls for robotics and other motion automation.

The I2C bus for the CMRS\_2560 application consists of a clock line (SCL) and a data line (SDA), along with 5V power and a common return. Rather than run the I2C clock at the default 1MHz, for the CMRS\_2560 the clock is software selected to be 100kHz. Running the clock at a slower speed should provide more reliable operation across a longer I2C bus. Using shielded wiring, bus lengths over 6 feet long have been tested on the bench; however, these bus lengths have not been tested installed at the club to date.

Ethernet Controller

The W5100 Ethernet board is a relatively low-end Ethernet controller, but the communications protocol for all of the devices is simple Telnet connection that sends data on state updates and an occasional sync message and receives a subset of those messages back from the JMRI SimpleServer.

A physical Ethernet board was selected over a WiFi enabled Arduino or shield for connection reliability and to minimize interference and load to the club WiFi network. Though not installed at this time, there is a plan to use three or four Cisco switches as concentrators around the Layout Room that will be exclusively for the CMRS\_2560 system. The Internet network has been selected to be an “unrouteable network space” so as to minimize the possibility of the network being attacked.

To date, the CMRS\_2560 systems that have been deployed do not require any feedback from other parts of the network and are running without the network connected. The software has been designed to allow the CMRS\_2560 “station” to continue to run and allow for local operation of switches even if the network connection is missing or the connection to the JMRI Simple Server is unavailable. In these cases, missing feedback may cause unexpected signal aspects, but shouldn’t impact switch control, except for the rare situation where a “station” may have a remote panel mirroring the station behavior via the network connection.[[2]](#footnote-2)

Interface Boards

Connected to the I2C bus are the CMRS Interface Boards. These are a combination of custom built boards and commercially available I2C boards. Four types of I2C interface chips are used: the Texas Instruments PCF8574[[3]](#footnote-3) and PCF8574A[[4]](#footnote-4) Remote 8-Bit I/O Expander for I2C Bus, the Microchip Technologies MCP23017 16-Bit I/O Expander with Serial Interface[[5]](#footnote-5), and a Liquid Crystal display controller (model TBD).

There are several boards currently in operation or development:

* 4 Channel Turnout Controller (MCP23017 controller)
* Toggle Switch Board – 4 inputs (PCF8574 controller)
* Indicator Board – 8 outputs (PCF8574A controller)
* Detector Input Board – 8 inputs (PCF8574A (?) controller)
* Signal Driver Board – 4 signals (MCP23017 controller)
* 4x4 Keypad (PCF8574 controller commercial board)
* 16 channel Relay Board Controller (MCP23017 controller commercial board)
* Liquid Crystal – 20x4 display (TBD controller commercial board integrated)

I2C Interface Specification

The addressing scheme for I2C follows the specification in the follow diagram:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | A7 (MSB) | A6 | A5 | A4 | A3 | A2 | A1 | A0 (LSB) |
| I2C add | L | D | D | D | A2 | A1 | A0 | R/W |

The “address” specified in the Arduino software interface are the middle 6 bits, 6 to 1. Bits 6 through 4 are usually masked permanently by the I/O device. For the PCF8574 and MCP23017, DDD is masked to 100; for the PCF8574A, DDD is masked to 111. The three programmable bits, A2, A1, and A0, are pins on the device and are set by a DIP toggle switch on the board in question. Thus, the accessible address space for the CMRS\_2560 devices are:

|  |  |  |
| --- | --- | --- |
| Device | Hex Address (A6-A1) | Decimal Address (A6-A1) |
| PCF8574 | 20 – 27 | 32 – 39 |
| PCF8574A | 38 – 3F | 56 – 63 |
| MCP23017 | 20 – 27 | 32 – 39 |
| Liquid Crystal[[6]](#footnote-6) | 27 | 39 |

Thus, for the selection of devices in this project, there are only 16 available addresses on the I2C bus. The current allocation of addresses is thus:

|  |  |  |
| --- | --- | --- |
| Device Type | Max in Project | Address allocated in Software |
| Toggle Switch Board | 2 | 32 – 33 |
| Sensor Board | 2 | 34 – 35 |
| Quad Turnout Board | 4 | 36 – 39 |
| Indicator Board | 2 | 56 – 57 |
| Signal Board | 2 | 38 – 39 |
| Relay Board | 1 | 39 |
| Liquid Crystal | 1 | 39 |

Software Description

The Arduino Mega 2560 software is developed with the Arduino IDE tool. The language used is C++, though most of the memory intensive classes are not implemented in the language subset. In fact, using the least amount of memory for the application is almost always the developer’s first concern. The compiler in the Arduino IDE provides the statistics on the memory usage of the application. A language sensitive editor for C++, such as Eclipse, is also recommended to be able to navigate and edit the application more efficiently. Finally, the applications used for the CMRS\_2560 project are kept under version control at <https://github.com/rdlehmer/arduino/> including this documentation.

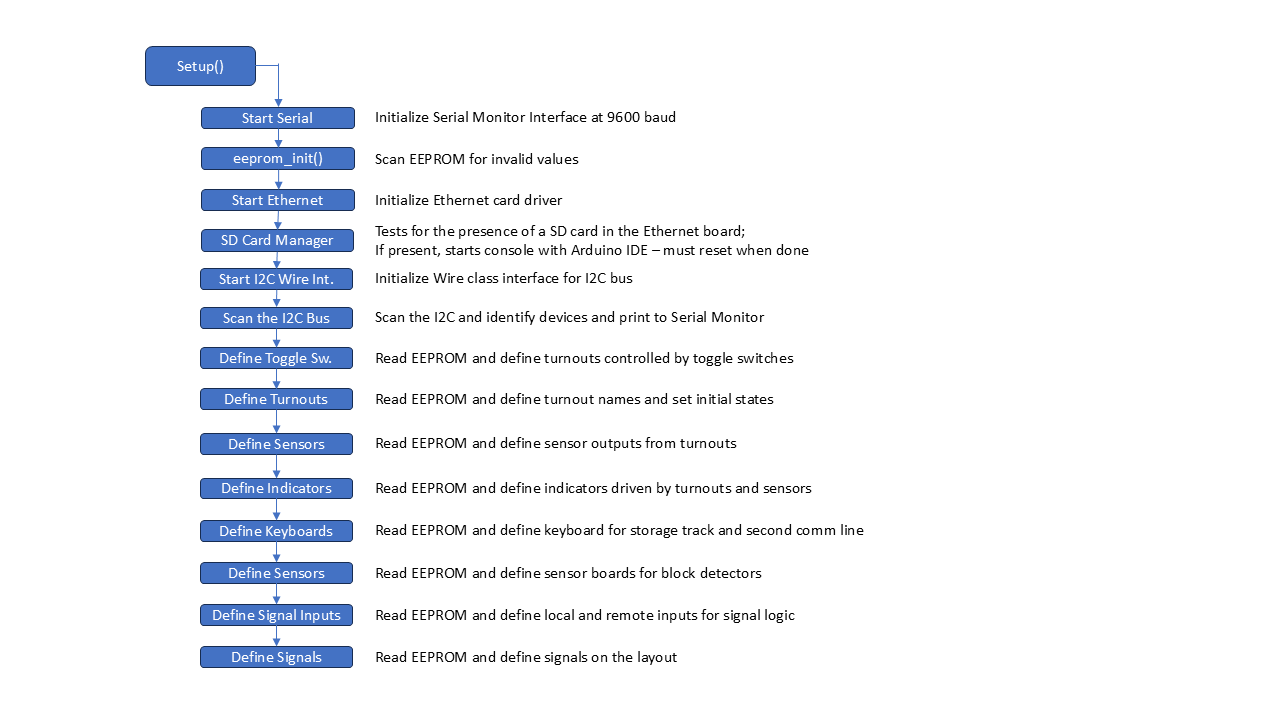


Figure 2 – Flowchart of the CMRS\_2560 setup() function.

The Arduino itself understands only two “entry” points, calling setup() once at the beginning of execution and then calling loop() repeatedly until physically reset or powered off. Figure 1 shows the flowchart of the setup() function for the CMRS\_2560 project. The setup() routine initializes the interfaces (Serial for the monitor console, the Ethernet driver, and the I2C driver). It also tests for the presence of a microSD card inserted in the Ethernet shield.

If the microSD card is present, this is used as a signal to enter the device configuration console using the Serial Monitor interface in the Arduino IDE. See Appendix A for more information on the configuration console commands.

**NOTE: Once configured, the microSD needs to be removed (or at least ejected) and the Arduino needs to be reset. Inconsistent behavior has been observed if the Arduino is allowed to execute after the configuration console is used.**

The 2k EEPROM in the Mega 2560 is a critical element of the functionality of the device. Important configuration information about the “station,” including Ethernet addresses, the number and type of I2C modules, and the specific function and behavior of each channel of those modules are defined by the configuration tool, and the contents of the EEPROM can be written to a file on the microSD card, as well as having the data read back into the EEPROM. The default value of each byte of the EEPROM is FF (255) and generally the code considers 255 to be the same as a 0, but the eeprom\_init() function does scan the EEPROM and sets cells that are 255 to 0.

Once the Wire class is initialized in the software, the scan\_i2c() function is called and the valid address space of the I2C bus is scanned and any responding I2C addresses are reported to the serial monitor console. This is completely informational and does not cross-check or validate the addresses with the configuration that is stored in EEPROM.

Appendix A CMRS\_2560 Console Commands

The following commands are available once the CMRS\_2560 is in configuration mode (powered up with a microSD card installed in the Ethernet shield).

**NOTE: The CMRS\_2560 software on startup scans the EEPROM addresses from 0016 to 2047 and any byte that is set to FF (255) is automatically updated to 00. In most cases, FF can be interpreted as a valid value that can cause unexpected behavior. Rather than guard against FF in every case, all EEPROM bytes above the MAC and Internet fields (where FF is possibly a valid value) are updated if they are equal to FF.**

Show Configuration

*conf*

This prints the configuration of the hardware. This includes the following data:  
 MAC address xx-xx-xx-xx-xx-xx  
 IP address xxx.xxx.xxx.xxx  
 Server address xxx.xxx.xxx.xxx  
 Number of toggles/sensors/quad turnouts/indicators/signals/dual turnouts/relay  
 Quad Turnout Configuration  
 Quad Turnout Sensor Configuration  
 Dual Turnout Configuration  
 Sensor Board Configuration  
 Signal Input Configuration  
 Signal Logic Configuration  
 Indicator Configuration  
 Signal Configuration

Examine

*ex aaaa dddd  
 ex/h aaaa dddd  
 ex/s aaaa dddd*

The *ex* command prints out data from the ROM in various formats. By default, data is shown in decimal; with the */h* argument, the data is shown in hex; and with the */s* argument, the data is shown as a string. Data is shown starting from the address *aaaa* and *dddd* bytes are shown. If the argument *dddd* is missing, one byte is shown. In hex or decimal mode, eight bytes are shown per line.

Deposit

*dep aaaa xxx  
 dep/h aaaa hh  
 dep/s aaaa CCCCCCC  
 dep/a aaaa xxx.xxx.xxx.xxx  
 dep/m aaaa hh-hh-hh-hh-hh-hh*

The *dep* command has various arguments to set the data is a ROM address or multiple adjacent addresses. The address *aaaa* is always in decimal, irrespective of which argument is selected.

Without any arguments, *dep* sets the address *aaaa* with the decimal value *xxx* (value limited to 0 to 255). With the */h* argument, the address *aaaa* is set with the hex value *hh* (00 to FF). With the */s* argument, the character string *CCCCCCC* is copied into ROM starting with the address *aaaa*. A null is appended to the end of the string *CCCCCCC* and addresses after the null terminator are not modified. With the */a* argument, an IP address defined by *xxx.xxx.xxx.xxx* is copied into 4 bytes of ROM starting with the address *aaaa*. Finally, with the */m* argument, the hex sextuple *hh-hh-hh-hh-hh-hh* is copied into 6 bytes of ROM starting with the address *aaaa*.

Write

*write <filename>*

The *write* command writes all of the ROM data to a *<filename>* as pairs of numbers. The first number is the decimal address of the ROM memory data and the second number is the decimal value in the ROM address. This is done for each and every byte.

Read

*read <filename>*

The *read* command opens the file *<filename>* (no extension is assumed) and reads the data from the file. Data shall be in the order of ROM address followed by a decimal value of the contents to be inserted. If the file *<filename>* does not exist, an error is produced.

Quit

*quit*

The *quit* command terminates the configuration mode. The user should remove the microSD card and press the Reset button on the Arduino. **WARNING: Unexpected behavior can be expected if the hardware is allowed to execute beyond typing the quit command.** The *quit* command is not required to complete configuration; the microSD card can be removed (simply ejected from the slot) and the Reset button pressed to restore the hardware to normal operation and test any modifications to configuration.

Appendix B JMRI Simple Server Messaging

Each CMRS\_2560 “station” needs to be a Telnet client of the JMRI Simple Server running on port 2048 on the system running the JMRI for the dispatching function. Please consult on the JMRI documentation at [www.jmri.org](http://www.jmri.org) for additional configuration information, such as having multiple JMRI instances communicating through the JMRI Simple Server.

These messages are currently supported by the CMRS\_2560 through JMRI:

Turnout controls

Message: TURNOUT NTxxxx { THROWN | CLOSED }

The turnout name label “NTxxxx” in the JMRI world identifies the device as an NCE (“N”) controlled Turnout (“T”) with up to 4 characters for the accessory number reserved. To be precise, the CMRS\_2560 shouldn’t have a “N” prefix, but experiments with JMRI have shown that using some other prefix than NT for CMRS\_2560 driven turnouts prevents the turnouts from being controlled either with NCE throttles or WiFi connected phones. So, as long as CMRS is operating in a hybrid configuration, NT prefixes for turnout name labels is the established standard.

Sensor reports

Message: SENSOR ISxxxx { INACTIVE | ACTIVE | UNKNOWN }

SENSOR messages will come from Tortoise contact monitoring as well as block detection circuits. In addition to being able to be used to drive local or remote turnout indications, they are also useful as logical inputs to signal logic. CMRS\_2560 will use the “IS” prefix as the standard for sensor identification.

Signalhead status

Message: SIGNALHEAD nnxxxx { GREEN | FLASHGREEN | YELLOW | FLASHYELLOW | RED | FLASHRED | LUNAR | FLASHLUNAR | DARK }

Message to send information about signal status to or from the CMRS\_2560 and JMRI.

Appendix C Class Definitions

After that, a hierarchy of classes for the interdependent hardware are initialized from the EEPROM data. The first class that is defined is CMRStoggles (along with the supporting class cmrs\_toggle).

Class CMRStoggles

|  |  |
| --- | --- |
| **Method** | **Description** |
| CMRStoggles() | constructor |
| ~CMRStoggles() | destructor |
| void init() | initializes the Toggle boards on the I2C bus |
| void scan() | reads the Toggle boards on the I2C bus and computes the toggle state |
| int getToggle(arg) | returns the value of the toggle number referenced by the input arg |

Class cmrs\_toggle

|  |  |
| --- | --- |
| **Method** | **Description** |
| cmrs\_toggle() | constructor |
| ~cmrs\_toggle() | destructor |
| void set(arg) | sets the object with the value arg |
| byte get() | gets the object value |

There are four toggles defined per Toggle board. Each toggle is controlled by two inputs on the Toggle board. The cmrs\_toggle can have three states – 0 if neither input is active (contact closure), 1 if the THR contact is closed, and 2 if the CLO contact is closed. Toggles are read board by board, there is no map to read individual inputs.

The next class is CMRSturnouts, which is the interface to controlling the turnouts in the system. It contains a vector of cmrs\_turnout objects that do the work to control each turnout and has the implementation of the communications to JMRI when there is a state update.

Class CMRSturnouts

|  |  |
| --- | --- |
| **Method** | **Description** |
| CMRSturnouts() | Constructor |
| ~CMRSturnouts() | Destructor |
| void init() | scan EEPROM list of turnouts and initialize corresponding cmrs\_turnout object |
| byte getControl(arg) | return control (toggle) number for the turnout referenced by arg |
| void setControl(arg,val) | calls set(val) for the turnout referenced by arg |
| void setRemote(arg.val) | called when a remote TURNOUT message is received; calls sethw(val) to avoid sending an update back (and causing a feedback loop) and setSlavedControl(arg,val) to look and see if there is another turnout controlled by the same toggle |
| void sendTurnoutsStatus(arg) | calls sendUpdate() for the cmrs\_turnout object referenced by arg |
| void setSlavedControl(arg,val) | [private] scans cmrs\_turnout objects for additional turnouts that have the same control as the cmrs\_turnout object referened by arg – if another exists, set control to val. |

It’s important to remember that the remote messaging from JMRI is a source of commands as described in Appendix A, but that these commands can act as a feedback loop as they can make a roundtrip from the CMRS\_2560 and JMRI and back before the turnout actually moves and updates its state in the CMRS\_2560. So, no CMRS\_2560 should send a TURNOUT message for a turnout in response to receiving a TURNOUT message for the same switch label. However, it may generate a TURNOUT message for any “slaved” turnout, such as the other switch in a crossover pair that needs to move in response to the command to the other turnout.

Class cmrs\_turnout

|  |  |
| --- | --- |
| **Method** | **Description** |
| cmrs\_turnout() | constructor |
| ~cmrs\_turnout() | destructor |
| void init(i,j) | initializes the object with turnout channel i+1 and controlled by toggle j |
| void set(control) | calls sethw(control) and sendUpdate() if sethw() returns 1 |
| byte sethw(control) | Tests control and state and throws or closes turnout as necessary; returns 1 if there is change |
| byte getControl() | returns the control number (toggle) for the object |
| void sendUpdate() | If the network is available, sends a message to JMRI that the turnout command has changed state (see Appendix A) |
| void turnout\_throw() | [private] Throws the turnout assigned to object; updates EEPROM state |
| void turnout\_close() | [private] Closes the turnout assigned to object; updates EEPROM state |

For each turnout controller channel, all eight contacts from a Tortoise are connected to the board. Pins 1 and 8 are the drive power and the polarity of the drive power is reversed when the turnout needs to be driven to the other position. The CMRS\_2560 keeps the last state of the drive command in EEPROM so that turnouts don’t roll around as the system is powered up.

There are two sets of SPST contacts with a center common on each Tortoise. The set on pins 2-3-4 is reserved for switching DCC power for frog polarity control. The other set of pins 5-6-7 is monitored by input pins on the turnout controller board (two complementary inputs per turnout). Up to 16 of these inputs can be defined in CMRS\_2560 and are scanned at the same rate as all other inputs. If the contact is closed, the SENSOR is defined to be ACTIVE, otherwise it is INACTIVE.

Class CMRSquadSensors

|  |  |
| --- | --- |
| **Method** | **Description** |
| CMRSquadSensors() | Constructor |
| ~CMRSquadSensors() | Destructor |
| void init() | Initializes all sensor (two contacts per Tortoise) hardware for input |
| void scan() | Scans through list of quad Turnout sensors to monitor (in EEPROM); any sensor objects that change trigger a call to sendUpdate() |
| int getQuadSensor(arg) | Returns the value of the quadSensor object referenced by arg |
| void sendUpdate(arg) | If the network is available, sends a SENSOR message to JMRI (see Appendix A) |
| void sendQuadSensorsStatus(arg) | Calls sendUpdate(arg) – used to send periodic status messages to JMRI |

In addition to these sensor inputs being used to communicate status to JMRI and other CMRS\_2560 devices (important for the later implementation of the signaling system), they can also be used to drive the turnout indicators. Indicators can be programmed to be driven by the a sensor input or a toggle state if the sensor input is inoperative (bad Tortoise contacts are all too regular).

Class CMRSindicators

|  |  |
| --- | --- |
| **Method** | **Description** |
| CMRSindicators() | Constructor |
| ~CMRSindicators() | Destructor |
| void init() | Initialize the indicator boards |
| void set(arg,val) | Call cmrs\_indicator.set(val) referenced by arg with value val |

Class cmrs\_indicator

|  |  |
| --- | --- |
| **Method** | **Description** |
| cmrs\_indicator() | Constructor |
| ~cmrs\_indicator() | Destructor |
| void init(arg) | Set the channel number of the indicator |
| void set(arg) | Set the state of the indicator |
| byte get() | Get the state of the indicator |

Class CMRSpower

|  |  |
| --- | --- |
| **Method** | **Description** |
| CMRSpower() | Constructor |
| ~CMRSpower() | Destructor |
| void init() | Set the pin function to drive the 16 channel power relay board |
| void set(arg, val) | Set the state of the arg pin with value val |

Class CMRSsensors

|  |  |
| --- | --- |
| **Method** | **Description** |
| CMRSsensors() | Constructor |
| ~CMRSsensors() | Destructor |
| void init() | Initializes all sensor hardware for input |
| void scan() | Scans through the sensor inputs |
| int getSensor(arg) | Returns the value of the Sensor referenced by arg |
| void sendUpdate(arg) | If the network is available, sends a SENSOR message to JMRI (see Appendix A) |
| void sendSensorsStatus(arg) | Calls sendUpdate(arg) – used to send periodic status messages to JMRI |

Class CMRSsignalInputs

|  |  |
| --- | --- |
| **Method** | **Description** |
| CMRSsignalInputs() | Constructor |
| ~CMRSsignalInputs() | Destructor |
| void init() | Initializes all input objects to “unknown” state |
| void scan() | Not implemented |
| int get(arg) | Returns the value of the Input referenced by arg |
| void set(arg, val) | Sets the Input referenced by arg with value val |
| void sendSignalInputsStatus(arg) | Send periodic status messages to JMRI |

Class cmrs\_signal

|  |  |
| --- | --- |
| **Method** | **Description** |
| cmrs\_signal() | Constructor |
| ~cmrs\_signal() | Destructor |
| void init(arg) | Set the default signal aspect and any check if a leading signal is set |
| void set(arg) | Set the state of the signal in hardware |
| byte getLeadingState() | Get the state of the leading signal defined for this signal |
| byte get() | Get the state of the signal |
| void sendUpdate() | If the network is available, sends a SIGNALHEAD message for the state of the signal |
| void sendLeadingUpdate() | If the network is available, sends a SIGNALHEAD AAAxxxx UNKNOWN if this method is called. |

Class CMRSsignals

|  |  |
| --- | --- |
| **Method** | **Description** |
| CMRSsignals() | Constructor |
| ~CMRSsignals() | Destructor |
| void init() | Initialize the signal boards |
| void set(arg,val) | Call cmrs\_signal.set(val) referenced by arg with value val |
| byte get(arg) | Get the signal value for the cmrs\_signal referenced by arg |
| byte getLeadingState(arg) | Get the leading signal state value for the cmrs\_signal referenced by arg |
| void advanceSignal(arg) | Evaluate the aspect of the cmrs\_signal referenced by arg based on the aspect the Leading Signal defined for the signal |
| void sendSignalsUpdate(arg) | Calls cmrs\_signal.sendUpdate() for referenced arg and if the state of the Leading Signal is Uknown, calls cmrs\_signal.sendLeadingUpdate() also for the referenced arg. |

1. Board documentation is available at <https://docs.arduino.cc/hardware/mega-2560/> [↑](#footnote-ref-1)
2. This has not been implemented yet but is a design element for the Mococo Line stations 62 and 63. [↑](#footnote-ref-2)
3. PCF8574 Datasheet: <https://www.ti.com/lit/ds/symlink/pcf8574.pdf> [↑](#footnote-ref-3)
4. PCF8574A Datasheet: <https://www.ti.com/lit/ds/symlink/pcf8574a.pdf> [↑](#footnote-ref-4)
5. MCP23017 Datasheet: <https://ww1.microchip.com/downloads/en/devicedoc/20001952c.pdf> [↑](#footnote-ref-5)
6. The I2C controller has solder pads if a different address is required. However, the software is anticipating the default address of 39. [↑](#footnote-ref-6)