Moba3 Control System

System Documentation

Andreas Wälchli

October 4, 2021

Contents

1 Introduction

1.1 Historical Context

During the construction of the current model railway the decision was made to use traditional analog control for the trains. Many models were not yet equipped with digital decoders and a retrofit would have been exceedingly expensive. Since no industry standard digital solution was used for the train control, the decision was made to design and construct a custom digital control solution for all other components of the railway. This solution would provide push button based control from a schematic control board to control all switches, signals and all lighting.

1.1.1 Moba1

A first design used custom control and decoder boards based on the PIC16F527 microcontroller. The entire railway was divided into 5 sectors:

Shadow station, *left entry to main station*, *right entry to main station*, *track switches* and *lighting*.

Each sector had a *master* board embedded in the control board that controlled up to 16 push buttons, 16 LEDs on the control board and continuously transmitted a single 8-bit data packet to the decoders in the railway. This packet did not encode commands but rather a fixed *desired state*. Each decoder would receive the current state data and check what modifications to its own outputs had to be done to conform to that state. As a consequence each decoder required bespoke decoding logic. The data also had to be encoded very efficiently to fit the fixed-length packet. The sector *right entry to main station* for example controlled 6 switches, 2 entry signals, 4 exit signals and could drive up to 2 independent paths at once.

Each packet contained a single *even parity bit* was packed between start and end marker bits:

START	data0..7	$\text{parity}_0(\text{data})$	END

Table 1.1: Packet structure

Transmission was done over a custom unshielded single-ended single-wire data bus with a relatively low baudrate and every bit of data was transmitted over 3 bus clock cycles.

encoded bit	transmitted sequence
START	000
0	100
1	110
END	111

Table 1.2: bit encoding

1.1.2 Moba2

During construction and testing of the first system it became clear that this was highly limited: Each sector had its own *master* board and these were not able to exchange any data. Additionally programs for the PIC16F527 microcontroller needed to be written in MIPS Assembly, which made high-level software design difficult.

It was therefore decided to move the high-level control logic onto a RaspberryPi 3 (RPi) single-board computer. This allowed for programming of the control logic in high-level languages (Java was used).

The *master* boards for each sector now no longer managed the entire control logic. It simply transmitted the button states to the RPi and received both the states of the control board LEDs and the sector status packet from the RPi. Communication between the RPi and the *master* boards was done using SPI in mode $(0, 0)$.

The decoders in the railway and the previously described transmission scheme remained in place. Over time this system showed its weaknesses: A model railway is inherently a high-EMI environment. Relay-based switch drivers and brushed motors, for example, produce significant interference. But the transmission scheme was not designed with EMI resistance in mind: An unshielded single-ended serial bus is obviously not suited for such environments. As a consequence we observed many transmission errors. To solve this, very extensive error detection was required which slowed the data busses down significantly. Additional problems manifested themselves as soon as microcontrollers from another batch were introduced: The microcontrollers all used internal 4MHz RC oscillators. But the bus required relatively tight timings to avoid reception errors. This could be partially solved by calibrating each decoder controller to the frequency of its *master* board. In the end we were forced to introduce a self-calibration routine that would calibrate each microcontroller based on the incoming serial data signal. This routine would run at every startup and whenever the decoder had too many transmission errors. The issues of transmission errors could however never be completely solved.

1.2 Moba3

The issues outlined above eventually lead to the decision to introduce a full redesign before the model railway was completed. This new solution would be designed with EMI-resistance in mind from the beginning. It would make away with the sector concept and the state transmissions. A single controller board would control the entire railway and the decoders would use unified software.

1.2.1 Design Goals

The full redesign of the control system allowed for the definition of clear design goals. These are outlined in the following sections.

Stability and Standardisation

The first iterations used a lot of custom concepts and hardware solutions. For the redesign we specifically chose to replace many of the custom design elements with industry standard solutions where applicable. Where industry solutions are not directly transferrable, they should at least be adapted to our needs. The design would also account for the interference issues observed with previous iteration wherever possible.

Improved Commonality

The entire system shall use common hardware design as far as feasible: All boards should use the same microcontroller type and common areas of the PCBs should use identical routing, component and mounting hole positions. Different decoder types should also share as much hardware as possible. Care must be taken that this not lead to an overly unified PCB design that tries to cover all possible use cases. But where external connectivity requirements are shared between multiple decoder types it shall be attempted to use common hardware. Behavioural distinctiveness should - where possible - be provided through software only.

Ease of Develoment

The microcontroller type chosen should allow for easy software development in C/C++. It should also facilitate debugging and prototyping. Each board should have a clearly defined set of responsibilities as this complicates both development and testing. E.g. a switch decoder should only be responsible for receive switching commands and for driving the switch motors.

The PCBs should also support in-circuit reprogramming. Previous iterations required extraction of the microcontrollers for reprogramming. This made development and testing more tedious and increased the risk for premature wear on the boards and damage to the microcontrollers.

Scalabilty

The entire system should be easily scalable through the addition of additional decoders or decoder types. Where sensible address spaces shall be defined large enough for substantial increase of the number of decoders. The entire system shall support at least 50% more decoders per type as is required on introduction.

2 Design Specification

The MOBA3 control system consists of a central *controller board* (CTRB), an *I/O interface board* (IIB) and several *decoder boards* (decoder). The I/O interface board handles button presses, forwards them to the controller board and drives the control panel LEDs based on commands received from the controller board. The controller board processes button presses and issues corresponding commands to the I/O interface board and the decoders. The decoders are not as smart as with previous system iterations and do not require bespoke software. MOBA3 supports numerous decoder types, each designed for a specific purpose. At the moment there are decoders for switches, station entry signals, station exit signals and lighting.

2.1 Common Hardware

To facilitate design and scalability we defined a set of common hardware to be used for all board types. This defines common set of components that should be used if required. Additional components can be introduced on specific boards if needed.

2.1.1 Microcontroller

Each board uses a single central microcontroller. The model used is an ATmega328p. This is the exact same microcontroller as is used on *Arduino Uno* and *Arduino Nano* boards. This allows for easy prototyping and debugging using that board. The microcontroller is provided with an external clock signal from a 16Mhz crystal oscillator. All configuration fuses are set identically to microcontrollers on Arduino boards. This commonality in configuration between Arduino boards and our MOBA3 boards also facilitate testing and development.

2.1.2 Bus Transceivers

The data busses (specified below) require RS422/RS485 transceiver chips. All transceivers should be from the MAX481 transceiver family or compatible hardware. The MAX481 family consists of multiple different transceivers in different configurations (simplex, half-duplex, full-duplex, etc.). Any component of this family can be used and for each board the best suited one can be selected.

2.1.3 Multi-Pin Input and Output

For multi-pin I/O the individual pins should not be connected directly to the microcontroller. PISO and POSI circuits should be used instead. Standard ICs used for Arduino-based applications are the CD4021 and 74HC595 ICs for input and output respectively. These are buffered shift registers that can be daisy chained.

The CD4021 is an 8-bit shift register with *parallel load* functionality. This makes it useful for *parallel-to-serial* data input.

The 74HC595 is an 8-bit shift register with a *tri-state buffered parallel output*. This makes it useful for *serial-to-parallel* data output.

Both shift registers can be read and written using SPI hardware in mode $(0, 0)$.

2.2 Data Bus Specification

The MOBA3 control system is based on a UART serial bus with a baudrate of 9600. [1] The data is transmitted over differential pairs on an RS422/RS485 derived serial bus. The differential pairs are driven using MAX481 compatible transceivers operating at 5V. Physically the data bus uses 8P8C (RJ45) connections for both power (5V only) and data. There are 2 busses: The *local bus* and the *remote bus*. To improve EMI resistance shielded CAT6a cabling[2] is used. Low power decoders can be powered directly by the bus, higher power decoders do require external power delivery. Some decoders may require additional voltages. These should *always* use external power delivery. Any externally powered decoder also provides power to the bus. The current draw on a single connection in the bus is limited to 600mA. Care must be taken to ensure that no bus segment exceeds this limit. This can be solved either by introducing additional external power connections where required or by simply connecting decoders strategically. E.g. alternating high-current decoders and low current decoders. In a scenario where a single bus powered decoder is connected between 2 externally powered decoders that bus powered decoder could theoretically pull up to 1.2A assuming equal cable lengths. [3]

Bus Loading and Biasing

In a deviation from the RS422 specification the bus does not need to be terminated. Each decoder loads the differential pair with a $10k\Omega$ resistor. This loading introduces a theoretical limit on the number of supported decoders to around 150. But this is very

[1] higher baudrates up to 2 Mbaud are theoretically possible

[2] S/FTP with a grounded shield

[3] The resistance of each cable depends on the cable length. When mismatched cable lengths are used it must be ensured that no single cable exceeds the 600mA limit. If e.g. one cable is 1m long and the other one 2m long, 2/3 of the current will flow through the 1m long cable. In that scenario the entire structure would be limited to 900mA. In general if the cable lengths differ by a factor n, the current limit would be $(n + 1)/n * 600mA$.

obviously only a theoretical limit and does not need to be considered further. In case the system should be used for an application with more than around 100 decoders, we suggest the introduction active repeaters.

The differential pair is also not biased. The loading resistor equalises the voltage potential in the differential pair on disconnected decoders. The MAX481 transceiver will always read equal voltages as a logical 0. The transmitters are always powered. For situations where the busses are correctly connected biasing becomes irrelevant as the transmitters will always drive the differential pair.

2.2.1 Local Bus Specification

The *local bus* is a full-duplex bus used for communication between the central CTRB and the IOIB that manages I/O on the control panel. 2 differential pairs are used for bidirectional data transmission, the remaining 2 are used for power.

pin	colour	function	description
1	G/	MOSI_B	CTRB transmission, negative channel
2	G	MOSI_A	CTRB transmission, positive channel
3	O/	MISO_B	IOIB transmission, negative channel
4	B	GND	ground wire
5	B/	GND	ground wire
6	O	MISO_A	IOIB board transmission, positive channel
7	Br/	VCC	5V power
8	Br	VCC	5V power

Table 2.1: *local bus* RJ45 pinout

The *local bus* is a a direct connection bus between the CTRB and the IOIB and must connect exactly one of each of these devices.

2.2.2 Remote Bus Specification

The *remote bus* is a simplex bus used for communication between the CTRB and the decoders. The decoders are connected using daisy chaining. It only carries data from the CTRB to the decoders. The differential pair used for IOIB transmissions on the *local bus* is used for connection sensing on the *remote bus*: Each decoder feeds back power through this pair to the previous board in the chain. This allows for easy identification of a break in the chain. It also ensures that in case a decoder is connected to the *local port* the CTRB is presented with a permanently idle IOIB transmission channel.

Please note that it is not required for the controller transmission channels for the 2 busses are separated. The controller is allowed to transmit all data on both channels. It is however recommended that each channel is driven by a separate transceiver. The

pin	colour	function	description
1	G/	MOSI_B	CTRB transmission, negative channel
2	G	MOSI_A	CTRB transmission, positive channel
3	O/	STATUS-	sensing feedback ground connection
4	B	GND	ground wire
5	B/	GND	ground wire
6	O	STATUS+	sensing feedback 5V
7	Br/	VCC	5V power
8	Br	VCC	5V power

Table 2.2: *remote bus* RJ45 pinout

decoders can simply pass through the differential pair and use stub connections to their local receivers. If required it is also possible to add repeating transceivers to the chain. This should however not be necessary as the specified design should tolerate distances of more than 100m.

2.3 Data Transmission Format

Unlike the previous systems MOBA3 transmits *commands*, not *states*. All commands follow a common structure, but each decoder type defines its own *payload semantics*. It is however generally required that all commands are human readable and restricted to the printable ASCII character set. Each command is transmitted as a *line* terminated by a line break (`\n`) character.

There are 3 line formats: *display command* and *decoder command* lines are transmitted by the CTRB. *button event* lines are transmitted by the IOIB.

2.3.1 Binary Encoding and Endianness

Since we are restricted to the printable ASCII character set we need to use an alphanumeric encoding for binary data. A hexadecimal encoding is used, where for each byte the high nibble is transmitted before the log nibble. For example, the number `0x2f` would be transmitted as the sequence `2f`. Hexadecimal values are always transmitted in lowercase letters.

Multi-byte values also use a hexadecimal encoding. The bytes are ordered in *big-endian*: The least significant byte is transmitted first and the most significant byte is transmitted last. For example, the 16-bit number `0xba42` is transmitted as the sequence `42ba`. This holds true for values of any size.

2.3.2 Display Command

Display commands begin with a constant character prefix `D`. This is followed by a 12 character hexadecimal encoding of the 48-bit display data. Each bit of the display data directly corresponds to a single LED on the control board. The display data (`data`) is treated as a single 6 bytes long unsigned integer.

0	1..12	13
`D`	HEX(`data`)	`\n`

Table 2.3: display command structure

2.3.3 Decoder Command

Decoder commands start with a *type selector* (`type`) and a *decoder address* (`addr`). After this common prefix a decoder specific payload of arbitrary length can follow. The command is terminated - as always - with a line break. While theoretically arbitrary, the payload length should not exceed 12 characters.

0	1	2..n+2	n+3
`type`	HEX(`addr`)$_0$	`payload`	`\n`

Table 2.4: decoder command structure

Type Selector

The type selector indicates the decoder type the command is relevant for. Each decoder type has a unique type selector associated with it. Type selectors are always capital letters.

`type`	decoder type
`A`	station exit signals
`D`	*reserved: used for display commands*
`E`	station entry signals
`L`	lighting
`W`	railway switches

Table 2.5: assigned type selectors

Decoder Address

The second byte encodes the decoder address. The address is a hex-encoded 4-bit number. This is a special case where hexadecimal numbers do not use an even number of

characters. At the moment all addresses must lie in the range 1..9, but it is possible to relax this to the entire 0..f range in a future revision. In addition to a specific address there is also a *broadcast address* that is accepted by all decoders of a specific type. This address is the character "+".

Command Payload

The payload is an arbitrary character sequence with a minimum length of 1 and a maximum length of 12. The payload must match the regular expression `[a-z0-9!]{1,12}`. This limitation is primarily introduced to ensure that no command payload contains uppercase letters. These could be incorrectly interpreted by a decoder as the start of a command line. The list of allowed characters may be expanded in future revisions.

2.3.4 Button Event

The IOIB continuously monitors the states of all push buttons on the control panel. Whenever a change in a button state is detected, a corresponding *button event* is sent from the IOIB to the CTRB.

Each button has a 1 byte *button address* (`addr`). As the third character, a *direction identifier* (`dir`) is sent. When a new button is pressed, a *key-down* event is sent, when a button is released, a *key-up* event is sent.

0..1	2	3
HEX(addr)	dir	\n

Table 2.6: button event data structure

event type	dir
key-down event	+
key-up event	-

Table 2.7: button event direction identifiers

3 Circuit Boards

3.1 Controller

3.2 I/O Interface

3.3 Decoder Boards

3.3.1 General Purpose Decoder

3.3.2 Switch Decoder

4 Decoder Command Specification