Z21 Control System Racket API Specification

Florian Myter
fmyter@vub.ac.be
Vrije Universiteit Brussel

February 12, 2015

Contents

1 Introduction

This documents specifies the Racket API to the Z21 model train control system. The API is based on version 1.02 of the communication protocol as specified by Roco themselves. The API covers the entirety of the Z21's functionality with the exceptions of all RailCom and Loconet-based functionality (chapters 8 and 9 Roco's protocol specification). This document will specify the protocol employed for each functionality offered by the Z21 as well as the associated part of the Racket API.

1.1 Communication Protocol

A more extensive explication with regards to the communication protocol can be found in Roco's specification document [1]. This section will provide a brief overview for the sake of clarity.
All communication with the Z21 happens via the means of UDP, which entails that the communication is asynchronous. The Z21 has a default IP-address of *192.168.0.111* and listens for incoming packets on port *21105*. Furthermore, the Z21 assumes that clients listen on port *21105* for its answers.
Communication from the Z21 to clients can either happen in a direct Z21-to-client fashion or broadcast-wise. Concretely, Figure 1 shows an example of a typical communication sequence. The figure exemplifies the differences in the way the Z21 responds to various request. For example the *get-serial-number* request which is send by *Client1* is answered by the Z21 in a direct fashion. On the other hand, the Z21 will broadcast it's status to all clients that have subscribed to these broadcast messages (see Section ref?).

1.2 Passing addresses as arguments to message creating functions

The API provides a number of functions that need addresses in order to create specific messages (e.g. the ref?). These addresses are consistently divided into two parts the *add-lsb* which is the address' least significant byte and *add-msb* which is the address' most significant byte. Both bytes are to be passed to functions as string representing the hexadecimal value of the byte. For example, the ref? function sets the driving pace for a given loc. If one were to make the loc with address 3 drive forward with a speed of 7 in a speed range of 14, the message used to notify the Z21 if this intention would be created as follows: ref?

2 General Setup and Message functionality

The Racket API provides two general purpose set of files: *Z21Socket.rkt* and *Z21MessageUtil.rkt*. The latter should not be used as part of the API since it contains functions that are used internally. The former (i.e. *Socket.rkt*) contains the necessary functions allowing to

[1] `http://www.z21.eu/en/Downloads`

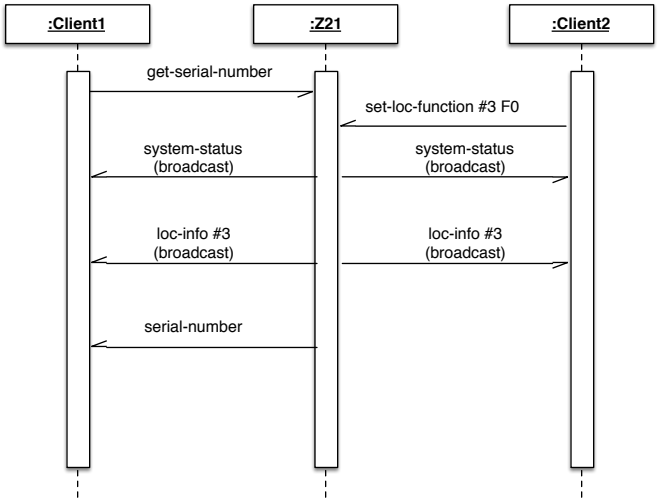

Figure 1: Example communication between the Z21 and two clients

easily set up the communication link with a Z21 as well as read out incoming messages. The provided functions are the following:

(setup) Initiates the necessary udp-ports in order to both send and receive messages to and from the Z21. Furthermore, the function returns the udp-socket which must be used to send messages to the Z21.

(send socket, message) Sends *message* to the Z21 provided the correct *socket*. Details on how to create appropriate messages can be found in the remaining sections.

(listen socket . reader) Spawns a new thread which will listen for all incoming traffic from the Z21on *socket*, which will be stored in a message queue for further processing (see *(read-messages reader)*). If the optional *reader* argument is provided the *reader* lambda will be called with the received message.

(read-messages reader) Recursively dequeues all messages from the message-queue built-up by *(listen socket . reader)* and calls the *reader* lambda with each dequeued message.

(get-message socket [length 65535]) Blocks until a message has been received, after which it returns the message inside a byte-string of *length* (which is default a byte-string of length 65535). This function need not to be used if *(listen socket . reader)* has been called.

(get-message* socket [length 65535]) Offers the same functionality as *(get-message socket [length 65535])* with the difference being that it is non-blocking.

3 System Status and Information

This section details the API functionality that concerns the system's general information (e.g. firmware versions) and the general status (e.g. temperature, voltage). All functions related to this part of the API can be found in *Z21MessageSysStat.rkt*.

3.1 Requesting the Z21's serial number

The *(make-serial-msg)* function will create a request message urging a Z21 to answer with its serial number. The *(is-serial-answer?)* function allows one to verify whether any given message is an answer to a serial number request. Furthermore, *(get-serial-number msg)* returns the serial number contained by *msg* provided that it was an answer to a serial request.

3.2 Logging off

One can explicitly log off by sending the message created with the *(make-logoff-msg)* function. Login into the Z21 system happens implicitly after the Z21 receives the client's first request. The Z21 does not respond to a "log off" message.

3.3 Requesting the x-bus version

The *(make-x-bus-msg)* function will create a request message urging a Z21 to answer with the current x-bus version. The *(is-x-bus-answer?)* function allows one to verify whether any given message is an answer to a serial number request. In order to extract both the x-bus version as well as the x-bus central from an answer message send by the Z21 one can respectively employ the *(get-x-bus-version msg)* and *(get-x-bus-central msg)* functions (provided that *msg* is an answer to an x-bus request).

3.4 Requesting the Z21's status

Sending a message created with the *(make-get-status-msg)* function will urge the Z21 to respond with message containing status information. This response message contains a subset of the information which can be acquired through a "get system state" message (see ref?). The Z21's response to a "get status" message (which can be verified with the *(is-status-changed-msg? msg)* function), can be one of the four following cases:

Emergency state The Z21 has been set in emergency state (physically indicated by the blinking of the front leds), the *(is-emergency-status-msg? msg)* function checks whether the the Z21's response signals this state (provided that *msg* is the Z21's response to a "get status" message.

Voltage off state The track current has been turned off, the *(is-voltage-off-status-msg? msg)* function checks whether the Z21's response signals this state.

Short circuit state A short circuit has happened on the track, the *(is-short-circuit-status-msg? msg)* function checks whether the Z21's response signals this state.

Programming mode The Z21 has been set in programming mode, the *(is-programming-mode-status-msg? msg)* function checks whether the Z21's response signals this state.

3.5 Turning off the track power

The track's power can be turned off by sending a message created with the *(make-set-track-power-off-msg)* function. Once the power has been turned off the Z21 answers with a "bc track power off" message. One can verify whether a response message is of the aforementioned kind by calling the *(is-bc-track-power-off-msg? msg)* (where *msg* is the newly received response message).

3.6 Turning on the track power

The track's power can be turned on by sending a message created with the *(make-set-track-power-on-msg)* function. Once the power has been turned on the Z21 answers with a "bc track power on" message. One can verify whether a response message is of the aforementioned kind by calling the *(is-bc-track-power-on-msg? msg)* (where *msg* is the newly received response message).

3.7 Command failure

If the Z21 receives an unknown command, because of a badly created or received udp-packet, it will answer with the "unknown command" message. The *(is-unknown-command-msg? msg)* functions checks whether a given *msg* is said "unknown command" message.

3.8 Enable emergency state

Switching the Z21's emergency state on, which can be done by sending a message created with the *(make-stop-msg)* function, will cause all locs to stop moving immediately. However, the track power remains on. Once the "emergency state" message has been processed the Z21 will answer with a "bc stopped" message which can be verified using the *(is-bc-stopped-msg? msg)* function.

3.9 Requesting the Z21's firmware version

In order to find out the Z21's firmware version one can send a message created with the *(make-firmware-msg)* function. The *(is-firmware-answer? msg)* function can be used to check whether a given *msg* is an answer to a the aforementioned request. Furthermore, the *(get-firmware-version msg)* function can be used to extract the firmware version from the response message. The firmware is a decimal number for which the integer part indicates the major version and the fractional part indicates the minor version (e.g. *1.12*).

3.10 Enabling broadcast flags

The Z21 has a number of different broadcast messages which are packed into three categories:

Driving and switching information These messages concern everything related to track related information. Concretely this category concerns the following messages: ”bc track power off” (see 3.5), ”bc track power on” (see 3.6), ”bc programming mode” (see ref?), ”bc stopped” (see 3.8), ”loco info” if the client has subscribed to the appropriate loc-address (see ref?) and ”turnout info” (see ref?).

Position information Concerns the position of locs on the tracks. Concretely this category concerns the ”rmbus data changed” message (see ref?).

System information Concerns meta-information about the Z21 itself. This category concerns the ”system state” message (see 3.12)

The client is able to subscribe to any combination of these three categories by enabling certain flags in the Z21. To this end the API provides the *(make-broadcast-msg flags)* function which creates a message which, ones send, will enabling these flags. The *flags* argument of the function is the list of categories to which the client would like to subscribe. The API offers all three categories through the following top-level variables: *general-broadcast-flag*, *position-broadcast-flag* and *system-broadcast-flag*.

3.11 Requesting the enabled flags

In order to find out which flags have been enabled (i.e. which category of message will be broadcasted) for him/her, a client send a ”get broadcast flags” message which can be created via the *(make-get-broadcast-msg)* function. The answer to this message can be verified with the *(is-get-broadcast-answer? msg)* function. Moreover, the *(get-broadcast-flags msg)* function will return the list of enabled flags contained in the response message.

3.12 Requesting the Z21’s state

The functionality described in 3.4 will only allow to inspect part of the Z21 full state. In order to receive a complete overview of the Z21’s state one must send a message created with the *(make-get-system-state-msg)* function. This message will urge the Z21 to respond with a full report of it’s system state via a ”system state changed” message, which can be verified with the *(is-system-state-changed-msg? msg)* function. This response message contains the following data:

The ”system state” datum is exactly the same as the one specified in 3.4, the same functions can be used to determine in which state the Z21 is currently in. The ”system state ex” datum can be either one of four states:

high temperature The Z21 is overheating, function *(is-system-stateex-temp-high? msg)* checks whether this is the case.

Datum	Accessor	Comment
main current	(get-system-state-main-current msg)	current on the main track in mA
prog current	(get-system-state-prog-current msg)	current on the prog track in mA
filtered main current	(get-system-state-filtered-current msg)	filtered current on main track in mA
temperature	(get-system-state-temperature msg)	temperature inside the Z21 in C
supply voltage	(get-system-state-supply-voltage msg)	supply voltage in mV
VCC voltage	(get-system-state-vcc-voltage msg)	VCC voltage in mV
system state		see explanation bellow
system state ex		see explanation bellow

power lost The Z21 has not enough input power to function, verification with the *(is-system-stateex-power-lost? msg)* function.

external short circuit An external short circuit has happened, verification with the *(is-system-stateex-short-circuit-ex? msg)* function.

internal short circuit An internal short circuit has happened, verification with the *(is-system-stateex-short-circuit-in? msg)* function.

4 Settings

This section details the API functionality that concerns the settings of both locs as well as switches. All functions related to this part of the API can be found in *Z21MessageSettings.rkt*. Concretely the Z21 allows one to specify whether communication with a loc or a switch should happen as stated by DCC [2] or MM [3].

4.1 Requesting a loc's mode

Requesting a loc's mode can be done by sending a message created with the *(make-get-loco-mode-msg add-lsb add-msb)* function, where *add-lsb* and *add-msb* specify the loc's address. The Z21 answers with a "loco mode" message which can be verified with the *(is-get-loco-mode-answer? msg)* function. Furthermore, the following functions are provided to extract information from the Z21's answer: *(get-loco-mode-address msg)* returns the loc-address for which the answer contains data, *(is-dcc-loco-mode? msg)* returns *true* if the specified loc is set in DCC mode and *(is-mm-loco-mode? msg)* returns *true* if the specified loc is set in MM mode.

4.2 Setting a loc's mode

Setting a loc's mode can be done by sending a message created with the *(make-set-loc-msg add-lsb add-msb mode)* function. Where *add-lsb* and *add-msb* specify the loc's

[2] http://www.nmra.org/dcc-working-group
[3] http://spazioinwind.libero.it/scorzoni/motorola.htm

address and *mode* is either of two top-level variables provided by the API: *dcc-mode* and *mm-mode*. The Z21 does not provide a response message to this request.

4.3 Requesting a turnout's mode

This functionality is similar to the one described in 4.1. The message used to get a turnout's mode can be created via the *(make-get-turnout-mode-msg add-lsb add-msb)* where *add-lsb* and *add-msb* specify the turnout's address. The Z21 responds with a "turnout mode" message which can be verified with the *(is-get-turnout-mode-answer? msg)* function. The *(get-turnout-mode-address msg)*, *(is-dcc-turnout-mode? msg)* and *(is-mm-turnout-mode? msg)* respectively get the turnout's address from the message and checks whether the Z21 responded that the turnout is either in DCC mode or in MM mode.

4.4 Setting a turnout's mode

In order to set the mode of a specific turnout one can create the necessary message with the *(make-set-turnout-msg add-lsb add-msb mode)* function, where *add-lsb* and *add-msb* specify the turnout's address and mode is either one of the top-level provided variables *dcc-mode* or *mm-mode*.

5 Driving

This section details the API functionality that concerns driving the locs as well as general loc-related information. All functions related to this part of the API can be found in *Z21MessageDriving.rkt*.
All data involving a loc's speed should be interpreted as follows: Every loc has a maximum speed specified in its CV (see ref?), a speed range and an actual speed. A speed range can be one of three pre-set ranges: 14,28,128 for which the actual speed must lie between $[0, level]$. Furthermore, the level indicates how the maximum speed should be specified. For example, if a given loc has a maximum speed of 250 km/u and its speed level has been set to 14 this entails that actual speed 14 = 250 km/u, actual speed 7 = 125 km/u etc. If the speed level were to be set to 28 then actual speed 28 would be 250 km/u and actual speed 14 would be 125 km/u.

5.1 Requesting loc information

Sending a message created with the *(make-get-loco-info-msg add-lsb add-msb)* function, where *add-lsb* and *add-msb* specify the loc's address, will result in the Z21 answering with a "loco info" message. Moreover, the client will automatically be subscribed to all future events involving this loc (such as changes to it's state by other clients and are physical events). This "loco info" message, which can be verified with the *(is-loco-info-msg? msg)* function, looks as follows:

The *function on* datum allows one to check which of a loc's functions has been turned

Datum	Accessor	Comment
loc address	(get-loco-info-address msg)	concerned loc's address
loc busy?	(loco-info-busy? msg)	boolean indicating the loc's status
speed range	(get-loco-info-speed-range msg)	loc's current speed range
direction	(get-loco-info-forward? msg)	boolean indicating the loc's direction
speed	(get-loco-info-speed msg)	loc's speed within the speed range
doppel traction	(loco-info-doppeltraction? msg)	boolean indicating whether doppeltraction is on
smart search	(loco-info-smartsearch? msg)	boolean indicating whether smart search is on
function on	(loco-info-func-on? msg func-index)	see explanation bellow

on. The *(loco-info-func-on? msg func-index)* function will return for the given *func-index* (.e.g. 0 is the index for function zero (lights)) a boolean indicating its status (true = on).

5.2 Setting a loc in motion

In order to make a loc move on needs to create the appropriate message with the *(make-set-loco-drive-msg add-lsb add-msb range forward speed)* function, where *add-lsb* and *add-msb* specify the loc's address, *range* specifies the speed range (for which three top-level variables are provided by the API: *low-speed-range*, *med-speed-range* and *high-speed-range*), *forward* is a boolean indicating whether the loc should move forwards and *speed* is the actual speed specified within the *range*.
The Z21 will not provide an answer unless the client has subscribed themselves to events involving the concerned loc (see 5.1).

5.3 Enabling a loc's function

In order to use a loc's specific function one needs to send a message created with the *(make-set-loco-function-msg add-lsb add-msb type func-num)* function. The *add-lsb* and *add-msb* arguments specify the loc's address, *type* can be either one of four top-level defined types: *func-off* which turns the specified function off, *func-once* which triggers the function once (e.g. a horn), *func-switch* which turns the function on until a new message arrives setting it off and *func-nap* stating that none of the previous are applicable to the specific function. The *func-num* indicates which of the loc's function the message concerns, this number depends on the actual loc (a list of available functions should given in the loc's manual).

6 Switch functionality

This section details the API functionality that concerns activating switches as well as requesting switch-related information. All functions related to this part of the API can be found in *Z21MessageSwitches.rkt*.

The status of any given switch can be in either of two positions: status 1 or status 2. The mapping from these abstract positions to the physical position (i.e. left or right) depends on how the switch has been physically laid out.

6.1 Requesting switch position

To request the status of a switch a message created with the *(make-get-switch-info-msg add-lsb add-msb)* function needs to be send to the Z21, where *add-lsb* and *add-msb* specifies the switch's address. The Z21 answers with a "switch info" message, which can be verified with the *(is-switch-info-msg? msg)* function.
This message contains data that can be accessed by the following functions: *(get-switch-info-address msg)* returns the switch's address for which the data was requested, *(get-switch-info-position msg)* returns the position in which the switch is currently in this position can be one of four top-level defined positions: *switch-position1*, *switch-position2*, *switch-position-no-position* (which is the position when the switch has not been set yet) and *switch-illegal-position* (which signals that the switch is wrongly set).

6.2 Setting switch position

In order to set the position of a switch, one needs to send a message created by the *(make-set-switch-msg add-lsb add-msb active position)*. *add-lsb* and *add-msb* specify the switch's address, *active* is a boolean whether the position should be activated (true = active) and position is an integer (either 1 or 2).
The Z21 will not answer the message unless the client has subscribed to events concerning this switch by calling the message detailed in 6.1.

7 Decoder CV functionality

This section details the API functionality that concerns reading and writing values from and to decoder CV's (be them loc-decoders or other). All functions related to this part of the API can be found in *Z21MessageDecoderCV.rkt*.
One can distinct two kinds of functions provided by the API: either regular CV reading and writing (i.e. the CV's of loc's) or the reading and writing of CV's belonging to accessory decoders (e.g. signalisation lights).
Furthermore, in case one is reading or writing loc CV's one can either put the loc on the prog track, or one can programme it directly on the main track with "main track" functions (if the Z21's firmware is $>= 1.22$) .

7.1 Reading a CV

Reading the value of a particular CV can be done by sending a message constructed with the *(make-cv-read-msg add-lsb add-msb)* function, where *add-lsb* and *add-msb* specify the CV's address. The Z21 will answer with a "bc programming mode" message, which can be verified with the *(is-bc-programming-mode-msg? msg)* function provided by *Z21MessageSysStat.rkt*. Furthermore the Z21 will either answer with "cv

nack sc", "cv nack" or with "cv result" messages.
In the former case, which can be verified with the *(is-cv-nack-sc-msg? msg)* and *(is-cv-nack-msg? msg)* functions, this entails that the Z21 was unable to read the specified CV. In the latter case, which can be verified with the *(is-cv-result-msg? msg)* function the read was successful and one can get the CV's address out of the message with the *(get-cv-result-address msg)* function and the result from the read operation with the *(get-cv-result-result msg)* function.

7.2 Writing a CV

Writing a value to a particular CV can be done by sending a message constructed with the *(make-cv-write-msg add-lsb add-msb value)* function, where *add-lsb* and *add-msb* specify the CV's address and *value* is the value to be written. The Z21 answers in the same way it responds to a "read cv" message (see 7.1). In case the CV was written successful it will still return the newly written value (i.e. with the "cv result" message).

7.3 Writing a CV byte on main track

This function is used to write a byte to the CV of a loc while the loc is not on the prog track (which is why one needs to specify the loc's specific address). In order to do so one must send a message created with the *(make-cv-pom-write-byte-msg loc-add-lsb loc-add-msb cv-add-lsb cv-add-msb value)*. *loc-add-lsb* and *loc-add-msb* specify the loc's address while *cv-add-lsb* and *cv-add-msb* specify the CV's address. Furthermore *value* specifies the value to be written, which must be a byte (i.e. a value in the range $[0, 255]$).
The Z21 will not provide an answer to this message.

7.4 Writing a CV bit on main track

This function is used to write a single bit to the CV of a loc while the loc is not on the prog track (which is why one needs to specify the loc's specific address). In order to do se one must send a message created with the *(make-cv-pom-write-bit-msg loc-add-lsb loc-add-msb cv-add-lsb cv-add-msb value bit-pos)*. *loc-add-lsb* and *loc-add-msb* specify the loc's address while *cv-add-lsb* and *cv-add-msb* specify the CV's address. Furthermore, *value* specifies the value to be written, which must be a single bit (i.e. 0 or 1) and *bit-pos* specifies the bit's position in the CV's byte (which must be a value in the range $[0, 7]$).
The Z21 will not provide an answer to this message.

7.5 Reading a CV byte on main track

This functionality is only provided by Z21's with firmware version $>= 1.22$.
This function is used to read a loc's CV while the loc is on the main track. The message needed to be send can be created with the *(make-cv-pom-read-msg loc-add-lsb loc-add-msb cv-add-lsb cv-add-msb)* function. *loc-add-lsb* and *loc-add-msb* specify the loc's address while *cv-add-lsb* and *cv-add-msb* specify the CV's address.

Programming mode is not enabled by sending this message, entailing that the Z21 will only respond with either the "cv nack sc" and "cv nack" messages or the "cv result" message (see 7.1).

7.6 Writing an accessory decoder's CV byte on main track

This functionality is only provided by Z21's with firmware version $>= 1.22$.
In order to write a CV of an accessory decoder one needs to send a message created with the *(make-cv-pom-acc-write-byte-msg dec-add-lsb dec-add-msb whole-decoder cv-add-lsb cv-add-msb value . output-port)* function. *dec-add-lsb* and *dec-add-msb* specify the decoder's address while *cv-add-lsb* and *cv-add-msb* specify the CV's address. Furthermore, one needs to specify by means of a boolean (*whole-decoder*) whether the CV refers to the whole decoder, if this is not the case the optional *output-port* argument needs to specify which port of the decoder needs to be programmed. The *value* is the byte that is to be written (i.e. a value in the range $[0, 255]$).
The Z21 does not provide a response to this message.

7.7 Writing an accessory decoder's CV bit on main track

This functionality is only provided by Z21's with firmware version $>= 1.22$.
In order to write a single bit of an accessory decoder's CV one needs to send a message created with the *(make-cv-pom-acc-write-bit-msg dec-add-lsb dec-add-msb whole-decoder cv-add-lsb cv-add-msb value bit-pos . output-port)* function. *dec-add-lsb* and *dec-add-msb* specify the decoders address while *cv-add-lsb* and *cv-add-msb* specify the CV's address. Furthermore, one needs to specify by means of a boolean (*whole-decoder*) whether the CV refers to the whole decoder, if this is not the case the optional *output-port* argument needs to specify which port of the decoder needs to be programmed . The *value* is the bit that is to be written (i.e. either 0 or 1) and the *bit-pos* indicates the bit's position in the CV byte (i.e. it needs to be in the range $[0, 7]$).
The Z21 does not provide a response to this message.

7.8 Reading an accessory decoder's CV byte on main track

This functionality is only provided by Z21's with firmware version $>= 1.22$.
In order to read an accessory decoder's byte one needs to send a message with the *(make-cv-pom-acc-read-byte-msg dec-add-lsb dec-add-msb whole-decoder cv-add-lsb cv-add-msb . output-port)* function. *dec-add-lsb* and *dec-add-msb* specify the decoder's address while *cv-add-lsb* and *cv-add-msb* specify the CV's address. Furthermore, one needs to specify be means of a boolean (*whole-decoder*) whether the CV refers to the whole decoder, if this is not the case the optional *output-port* argument needs to specify which port of the decoder needs to be programmed.
The Z21 will answer either answer with a "cv nack" message or a "cv result" message (see 7.1).

8 Position tracking

This section details the API functionality that concerns loc position detection. All functions related to this part of the API can be found in *Z21MessageLocation.rkt*.
In general position data provided by the Z21 should be interpreted as follows. The Z21 can handle two 20 position modules which are divided into two groups: modules 1 to 10 are said to belong to group with index 0 and modules 11 to 20 are said to belong to group with index 1. Upon sending position data to the client, the Z21 sends a single message per group of modules. This single message allocates 10 bytes dedicated to the position of locs as measured by these 10 position modules. Each byte in these allocated 10 bytes indicates whether or not locs have been detected on a specified module and on which port of these modules the locs have been detected.
For example, if the Z21 returns the following 10 bytes for the modules with group index 1: *0x01 0x00 0xC5 0x00 0x00 0x00 0x00 0x00 0x00 0x00* . One needs to interpret these bytes as follows: position module 11 has detected a loc on output port 1, position module 13 has detected locs on output ports 8,7,3 and 1.
The mappings from group indices and position module numbers to physical devices depends on the layout.

8.1 Requesting location data

In order to request position data from the Z21 one needs to send a message created with the *(make-rmbus-get-data-msg group-index)* function, where *group-index* indicates for which group of position modules one wants the data.
The Z21 answers with a ”rmbus data changed” message, which can be verified with the *(is-rmbus-datachanged-msg? msg)* function. In order to read the position data from this answer one needs to use the *(get-rmbus-data msg)* function. The aforementioned function returns a data structure of which the data can be accessed with the following functions: *(get-group-index struc)* returns the group-index for which the structure contains position data, *(get-occupancies struc)* returns a list of data structure containing information for each specific position module, *(get-module pos-struc)* returns the position module at hand for a given position structure and *(get-ports pos-struc)* return a list of ports on which a loc has been detected.

8.2 Programming position modules

The Z21 allows one the set the addresses of position modules. Do note that only a single position module may be connected while programming a position module. In order to set the address of the only connected position module one must send a message created by the *(make-rmbus-programmodule-msg new-address)* function, where *new-address* is a value in the range $[1, 20]$. One must note that the communication protocol for this functionality differs from the rest of the aforementioned functions. In order to finalise the communication one needs to send the following message to specify the Z21 that the programming of the position module has been completed: *(make-rmbus-programmodule-msg 0)*. This ”set address to 0” message may only be sent once the front led lights on the Z21 have stopped blinking.