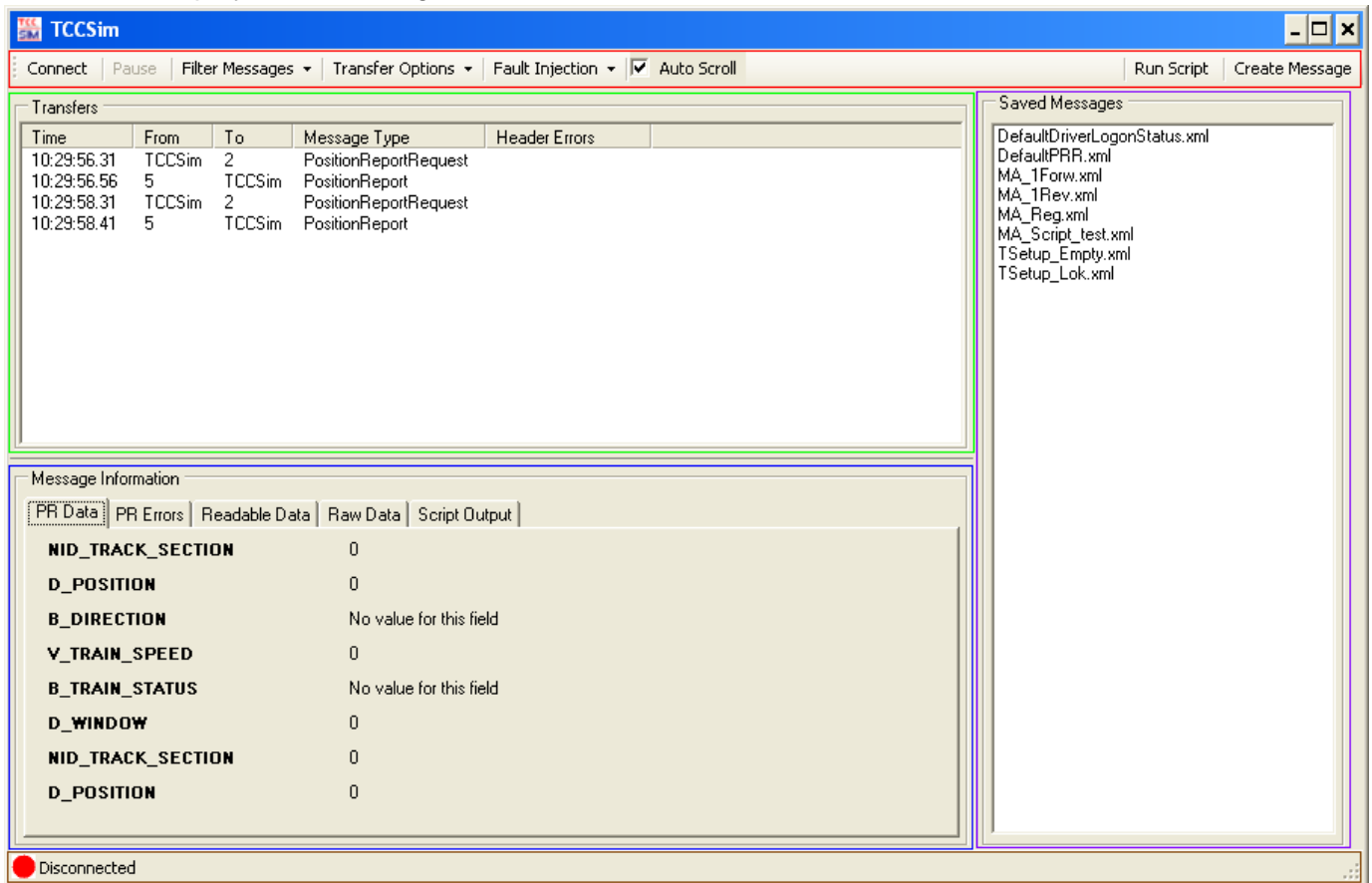


Manual for TCCSim

1 Main View

The main view is the window that will be presented when TCCSim is started. It consists of five main areas displayed in the image below.



Red: Is the main control area of TCCSim and contains a number of buttons. Left to right:

- **Connect:** Clicking this button will display the connect dialog (Section 1.1). When TCCSim is connected(or trying to connect) the connect button will disconnect.
- **Pause:** Will be active when TCCSim is connected or trying to connect. Clicking it will put the conversation with the vehicle on hold.(Nothing is sent or received)
- **Filter Messages:** Brings out a check list of all message types. If a message type is checked, message of that type will be displayed in the **green** area. Which types that are checked by default is controlled by the message definitions (Chapter 3).
- **Transfer Options:** Brings out three options. Clear, save and load transfers. Clearing the transfers will clear all items in the **green** area and all items in the “PR

Errors”

tab in the **blue** area.

'Save' will save the current items in the **green** area to a file of your choice.

'Load' will load previously saved files into the **green** area. **NOTE:** Loading a transfer file will clear what is currently in the **green** area.

- Fault Injection: Brings out a number of options to inject faults into the next (and only the next) outbound message.
- Auto Scroll: If checked will make the newest item in the **green** area visible as well as the newest item under the "PR Errors" tab in the **blue** area visible.
- Run Script: Will open a file dialog and a python script can be loaded. If a script is running it can be cancelled with this button as well.
- Create Message: Will open the Message View window. (Chapter 2)

Green: Contains all messages that are sent or received when connected to a vehicle.

The column 'Time' shows the time of day when a messages was sent/received.

'Header Errors' displays problems that was encountered with the safety header.

The rows can be clicked and will by that show complete message information in the tabs "Readable Data" and "Raw Data" in the **blue** area.

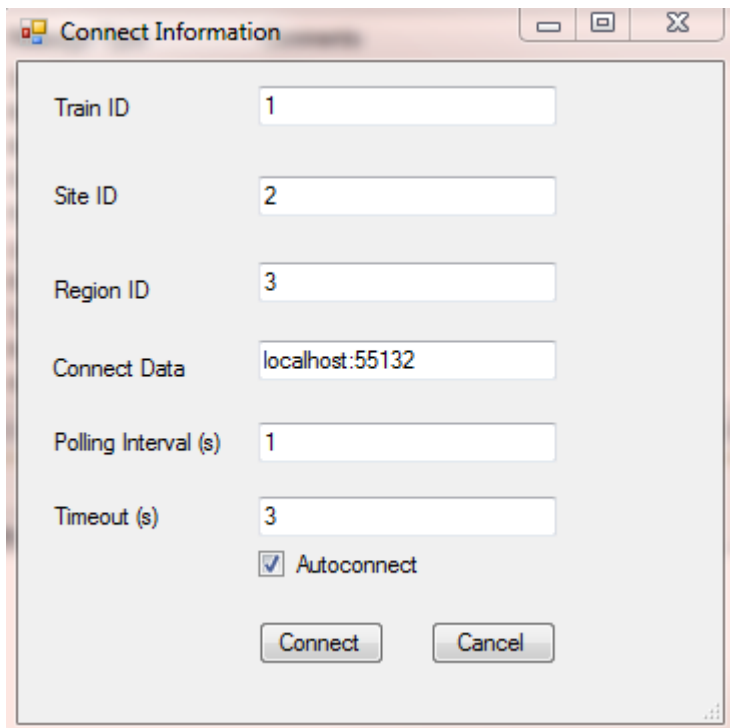
Blue: This area displays information of different kinds. Tabs left to right:

- PR Data: Acts as a status window for Position Report messages. Any message received that has the value of 'PRValue' in the config (Chapter 6) will have all its permanent fields printed in this tab.
- PR Errors: Any message that is printed in "PR Data" will be check for blocks that has the value of "PRErrorBlock" in the config(Chapter 6). Any such blocks will be added to the list in this tab.
- Readable Data: When clicking an item in the **green** area complete information of that message will be displayed in readable form in this tab. If the byte list can be parsed.
- Raw Data: When clicking an item in the **green** area the byte list of that message will be printed in hexadecimal form in this tab.
- Script Output: The output from a running script will be printed in this tab. It contains a "Clear" button which will clear the script output window.

Brown: This row contains status information regarding the connecting. A red dot means the status is connected or trying to connect. A green dot means a connection is established. The text tells more about the status.

Purple: This box displays saved messages which are located in the Messages folder. The Messages folder is located in TCCSim's main folder. When double clicking a file in the **purple** area the Message View(Chapter 2) will be displayed with the clicked file loaded. The files can also be right clicked which displays a menu with options to send, edit and delete the selected file. The send option will only be available when TCCSim is connected or trying to connect.

1.1 Connect Dialog

The image shows a Windows-style dialog box titled "Connect Information". It contains six text input fields arranged vertically. The first three fields are labeled "Train ID", "Site ID", and "Region ID", each containing the number 1, 2, and 3 respectively. The fourth field is labeled "Connect Data" and contains the text "localhost:55132". The fifth field is labeled "Polling Interval (s)" and contains the number 1. The sixth field is labeled "Timeout (s)" and contains the number 3. Below these fields is a checkbox labeled "Autoconnect" which is checked. At the bottom of the dialog are two buttons: "Connect" and "Cancel".

The Connect Dialog has six field used when setting up the connection. These six field will have default values specified in the config file (Chapter 6). If no default values are available, the fields will be blank.

- Train ID: What ID should be used in all outgoing messages
- Site ID: What Site ID should be used in all outgoing messages
- Region ID: What Region ID should be used in all outgoing messages
- Connect Data: At the time of writing TCCSim is built for TCP/IP connection. Preferable this field contains an IP address and a port number on form '<IP address>:<Port>'. (DNS address would also be valid.)
- Polling Interval: Decides the time in second between messages being sent from TCCSim.
- Timeout: A timeout value in seconds deciding how long the socket will listen for a response from the vehicle before calling timeout.
- If Autoconnect is checked then TCCSim will connect to saved values upon startup.

2 Message View

TCCSim message creator - (C:\TCCSim\TCCSim\TCCSim\Messages\TSetup_Empty.xml)

Message type	MovementAuthority	Resolution
NID_MSG	0	
T_VALID	0	minutes
Target speed	0	cm/s
G_GRADIENT	0	1 per mill
A_BRAKEABILITY	1	0.01cm/s ²
B_DIRECTION	<input type="checkbox"/> Driving direction (1 - Reverse) <input type="checkbox"/> Orientation in track section (1 - Loco towards 0-leg) <input type="checkbox"/> Locomotive orientation (1 - A-end facing cars)	
Q_ROUTE_TYPE	Location start	
NID_TRACK	0	
D_POSITION	0	cm
D_MA_MARGIN	0	cm
D_SAFETY_MARGIN	0	cm
KEEP_TRACK_DATA	Off	
PARTLY_MA	Off	
MAX_SEARCH_DIST	Off	
LOCATION_DATA	Off	
ATO_STOP_POSITION	Off	
PANTO_START_POS...	Off	
CAR_UNLOAD_DATA	Off	
TRACK_DATA	Off	
BALISE_DATA	Off	
GRADIENT_DATA	Off	
CEILING_SPEED_DA...	Off	
ACOUSTIC_SIGNAL	Off	
PANTOGRAPH_SHIFT	Off	
TRACK_DATA_ITEM	Off	
M_END_OF_MESSAGE		

Send Save Save As Load Clear Cancel

Message type - Selector for which message type to create.

Send - Send current message values.

Load - Load a previously saved message.

Clear - Load an empty message of selected type.
Cancel - Close window.

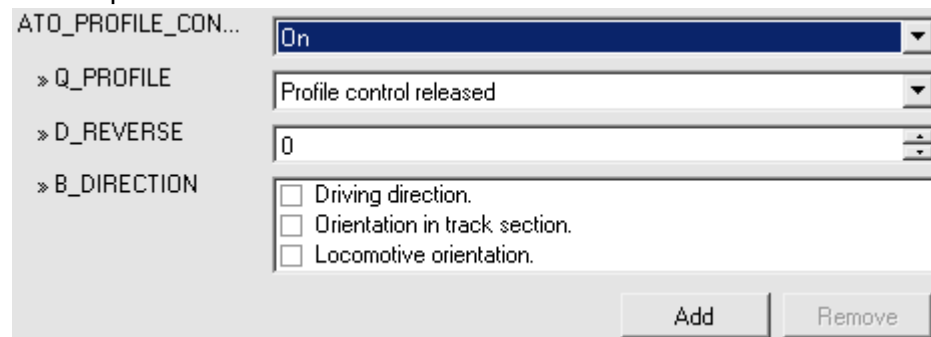
Different input formats.

NID_MSG is a Identification of the message and values can be directly inserted or incrementally increased or decreased with scroll wheel, arrow keys or clicking up/down buttons.

T_VALID is an input control for INT/UINT, values can be directly inserted or incrementally increased or decreased with scroll wheel, arrow keys or clicking up/down buttons.

B_DIRECTON is a BITMASK control. Each bits value is represented with its description.

Q_ROUTE_TYPE is an UINT where each value have a description. Each value is represented in a dropdown list.



Block types, ATO_PROFILE_CONTROL is a block type that can exists zero or more times in a message. When activated the first set of fields gets added. If more entries are added the Remove button will be enabled for each entry. New entries can be added or inserted in which every way wanted, just press add where wanted. All entries can be removed immediately by disabling the block type with the On/Off control.

NOTE: Please refer the FFFIS TCC-AOS Interflo 150 Interface Spec (Doc no. 3NSS004300D0107) version 5.0 for details of each messages/block and field description.

3 Message Definitions

TCCSim uses three different XML files to define message, block and field types. The definitions is very similar to the structure of the protocol specification.

Message types

In the file for message structure all message types are stored in the tag MessageDescriptions. There each message type is represented in a Message tag. This tag must have the attributes type and value. The name of the message type should be in **type** and NID_MESSAGE_TYPE in **value**. The first tag in the body of Message must be StationaryMessage, this is a Boolean value telling if this is a message from the stationary side. This is used to determine if the message type should be displayed in the message creation view. Next tag is Filtered, this tells TCCSim if

the message type should be shown or not by default in the message log in main view. From here the field types follow, each represented by a FieldType tag. This is a tag without any body and the attribute Name where the name of the field type is represented. Then the block types follows, BlockType is the name of the tag. BlockType has the attribute Name for the name of the block type, this cannot be omitted since TCCSim uses the name of the block types to identify them when parsing. This so that multiple block types can exist, the correct ones and faulty ones for fault injection. A BlockType also has the attribute Numeric for NID_BLOCK_TYPE. In the body of BlockType maximum and minimum occurrences of the block type can be specified with Min and Max. If it is desirable that a field or a block is displayed with another name than its type name then the Display tag can be used.

```
<Message type="MovementAuthority" value="4">
  <StationaryMessage>True</StationaryMessage>
  <Filtered>False</Filtered>
  <FieldType Name="T_VALID">
    <Display>
      Valid time
    </Display>
  </FieldType>
  <BlockType Name="REQ_CAR_STATUS" Numeric="10">
    <Min>0</Min>
    <Max>1</Max>
  </BlockType>
  <BlockType Name="TRAIN_NAME" Numeric="8">
    <Min>0</Min>
    <Max>0</Max>
    <Display>
      Train name
    </Display>
  </BlockType>
  <FieldType Name="M_END_OF_MESSAGE" />
</Message>
```

Block types

A block type is represented by a Block tag in the block descriptions file. A Block have the attributes type, for the block type name, and value for NID_BLOCK_TYPE. In the body of a Block the field types of the block is represented by a FieldType tag with the attribute Name for the field type name.

```
<Block type="BALISE_DATA" value="4">
  <FieldType Name="NID_TRACK_SECTION"/>
  <FieldType Name="D_POSITION"/>
  <FieldType Name="NID_BG">
    <Display>
      Balise ID
    </Display>
  </FieldType>
  <FieldType Name="D_PREVIOUS_BG"/>
</Block>
```

Field types

A field type is represented with the tag FieldDescription, this tag has the attribute **type**. In the attribute the name of the field is represented. Tags in the body of a field type:

Detail - Detailed description of field type.

Length - Length of field in bytes.

Min - Minimum value of field.

Max - Maximum value of field.

Resolution - Resolution of field.

Format - What data type the field represents. Can be STRING, INT, UINT and the special case of UINT BITMASK

Special - For special values.

BITMASKs are UINTs that have a descriptive string for each bit. The description of each bit is put in the body of Special. Here a tag Bits is put with a Bit tag for each bit of a BITMASK. Each Bit tag have the attribute value for the numeric number of the bit and in the body the descriptive string.

```
<FieldDescription type="B_TIC_STATUS">
  <Detail>TIC unit status</Detail>
  <Length>2</Length>
  <Min>0</Min>
  <Max>65535</Max>
  <Resolution></Resolution>
  <Format>BITMASK</Format>
  <Special>
    <Bits>
      <Bit value="0">Derail, front right</Bit>
      <Bit value="1">Derail, front left</Bit>
      <Bit value="2">Derail, rear right</Bit>
      <Bit value="3">Derail, rear left</Bit>
      <Bit value="4">Car dump bottom</Bit>
      <Bit value="5">Car dump top</Bit>
      <Bit value="6">Car dump closed</Bit>
      <Bit value="7">Load weight bad</Bit>
      <Bit value="8">Train config input</Bit>
    </Bits>
  </Special>
  <Default></Default>
</FieldDescription>
```

If Format is a UINT with a description for each value then Special can also be used. By putting a Field tag with the attribute value with the numeric value and the descriptive string in the body of the tag. These tags should be put in the body of the tag Fields.

```
<FieldDescription type="Q_ALERT">
  <Detail>Emergency alert code</Detail>
  <Length>1</Length>
  <Min>0</Min>
```

```

    <Max>255</Max>
    <Resolution></Resolution>
    <Format>UINT</Format>
    <Special>
        <Fields>
            <Field value="1">Initiated by driver</Field>
            <Field value="2">Initiated by dispatcher</Field>
            <Field value="3">Position report outside set route</Field>
            <Field value="4">Points inside set route in error</Field>
            <Field value="5">Profile control triggered</Field>
            <Field value="6">Powerless section</Field>
            <Field value="7">Gate forced open</Field>
            <Field value="8">Too many CRC errors on radio channel</Field>
        </Fields>
    </Special>
    <Default></Default>
</FieldDescription>

```

4 Communication

Some good to know things about how TCCSim handles communication with a vehicle.

TCCSim's connection status has three states. Disconnected, Connecting and Connected.

When disconnected, no connecting is trying to be established (TCCSim is in idle)

When connecting, either a connection is being established or one trying to be established. A connection will try to be established until a user disconnects or a connecting was established.

When connected, messages will be sent and received.

When connecting or connected, messages can be put on the send queue by the user. They will be sent when possible. If no message is in the send queue a default message will be sent. This message is specified in the config with the option "PRRMessage".

In the config two options called "DLSMessage" and "DIValue" is also specified. When a message with message type value of "DIValue" is received an instant reply with the message file from "DLSMessage" will be sent. (DI = DriverInformation, DLS=DriverLogonStatus)

TCCSim will alert of an inbound timestamp error if the incoming timestamp has a lower value than the previous and the difference (with wrap around) is higher than "Max (polling interval, timeout) * 2"

TCCSim will not accept messages which are longer than a certain length. The length is specified in the config with the option "BufferLength".

5 Script

The scripting language used by TCCSim is Python and the scripts are interpreted by [IronPython](#) 2.6. Using IronPython means not all Python features are implemented. C# libraries may, however, be imported and used.

Scripts can be loaded at any time or state (connected/disconnected). It's up to the user to load a script in the correct state it was written for. Example: Calling a script which assumes TCCSim is connected when in fact the status is disconnected would make no sense since no message will be sent or received.

The script will run until it returns or the user cancels it with the "Cancel Script" button in the top right corner of the main view.

Output from a script will be displayed under the "Script Output" tab in the main view of TCCSim. It will be more efficient to combine output statements.

Print (string1 + "\n" + string2) is preferred over print (string1) print (string2).

5.1 Script Functions

TCCSim provides the script writer with a number of functions. These functions will be explained rather detailed below. Using them incorrectly will make the script hang or crash. The functions are reached through the object "engine". Example: "engine.Disconnect()"

int Connect(int ID, String ConnectData, int Interval, int Timeout)

Will try to connect with to 'ConnectData' with train id 'ID' in its safety header. Messages will be sent with interval 'Interval' seconds and the socket will timeout after 'Timeout' seconds.

Note: If TCCSim's status is connected when calling Connect, the call will hang and won't return until the connection is broken.

Return values:

0: Status is connected.

-1: Status is disconnected. Something was wrong with the parameters given. Example: Invalid Port/IP address.

-2: Status is connecting. The vehicle did not respond properly. Trying to connect again. Calling Disconnect brings the status back to disconnect. Calling Connect again would most likely not give wanted results.

void Disconnect()

Closes any established connection or ends any tries to establish one.

void Pause(bool Pause)

'Pause' value True: Enable pause.

'Pause' value False: Disable pause.

void SendMessage(String File)

Sends a message which is located in file 'File'. Status cannot be disconnected.

Note: No error codes will be given if the file cannot be located or if there are any other faults with the file.

void SendMessage(byte[] bstr)

Sends the provided byte array 'bstr'. Status cannot be disconnected.

void SendMessage(XElement xm)

Sends the provided XElement 'xm'. Status cannot be disconnected.

XElement LoadMessage(string path)

Loads the message file given in 'path'. Returns a XElement which is a C# object and contains the message in an XML structure. The XElement can be manipulated and used by other functions. Trying to load a file which does not exist will make the script crash.

bool SetF(XElement xm, string fname, int index, string new_value)

Number 'index' occurrence of field 'fname' in the provided XElement 'xm' will be set to value 'new_value'. Returns true on success.

string[] GetFields(Byte[] bstr, string fname)

Given a message in a byte array 'bstr', returns all values of the fields with name 'fname' in a string array.

String GetMsgName(Byte[] bstr)

Returns the message name of the provided byte array 'bstr'.

DeliverEventArgs GetMessage()

Waits for a message from the vehicle and returns it. The return object is a DeliverEventArgs which is described in section 5.2. The script has an incoming message queue of its own. The queue will not be dequeued unless GetMessage() is called. It means that if GetMessage() is not called for a while and suddenly is, then GetMessage() will return old messages.

GetMessage() will hang if called when TCCSim's status is disconnected.

If the connection is lost, null(Python: None) will be returned. It will keep getting returned until the connection is reestablished or Disconnect() is called.

Void FaultyCRC()

Enables a faulty CRC calculation on the next (and only the next) outbound message.

Void ReuseTimestamp()

Reuses the timestamp from the previous outbound message on the next (and only the next) outbound message.

Void DecTimestamp()

Decreases the timestamp from the previous outbound message on the next (and only the next) outbound message.

Void IncRefTimestamp()

Increases time on the reference timestamp on the next (and only the next) outbound message.

Void DecRefTimestamp()

Decreases time on the reference timestamp on the next (and only the next) outbound message.

Void FaultyID()

Changes the train id value on the next (and only the next) outbound message.

Void Fragment (int msDelay)

The next outbound message will be fragmented into two parts and sent with a gap of 'msDelay' milliseconds. The number of outbound messages that are fragmented is equal to the number of times this function is called.

5.2 DeliverEventArgs

DeliverEventArgs is a class used by TCCSim to handle and contain messages. An object of type DeliverEventArgs is returned when calling GetMessage(). The properties available are:

- **Byte[] Data**
The received message in a byte array.
- **String Time**
The time of day when the message was received.
- **String Name**
The message type name.
- **bool TSSenderError**
If any errors with the timestamp was encountered, this value is true.
- **bool TSRefError**
If any errors with the reference timestamp was encountered, this value is true.
- **bool CRCError**
If the CRC calculation was faulty, this value is true.

Changing these properties is possible but prohibited. It will affect the outcome of the test run.

Scripting example:

```
# This is not a complete script, just examples of how to use
# the functions.

#Elementary part of all scripts, gives interface to TCCSim
import TCCSim.Script as Script

# Fetch message. This method is blocking.
m = engine.GetMessage()

# Print the name of the message.
print m.Name

# Read field from message.
# Using D_POSITION as example.
d_position = engine.GetFields(m.Data, "D_POSITION")
# This returns all fields of the type D_POSITION.
# You have to know which occurrence of the wanted field
# to use it.
```

```

current_position = d_position[1]

# Read PR messages and wait until a given position is reached
# and the train is idle.
while (True):
    m = engine.GetMessage()

    if (not (m.Name == "PositionReport")):
        continue

    d_position = engine.GetFields(m.Data, "D_POSITION")
    v_train_speed = engine.GetFields(m.Data, "V_TRAIN_SPEED")

    if (int(d_position[0]) < 3800 and int(v_train_speed[0]) == 0):
        break

# Done.

# sending messages.
message_path = "d:\\simulation_messages\\"

# Send a message from file.
engine.SendMessage(message_path + "message1.xml")

# Send a loaded message.
loaded_message = engine.LoadMessage(message_path + "message_load.xml")
engine.SendMessage(loaded_message)

# Alter a field of a loaded message.
m = engine.LoadMessage(message_path + "message_alter")
status = engine.SetF(m, "Name of field to alter", number of field to
alter as int, new value)
if (status): # SetF returns boolean
    engine.SendMessage(m)

```

6 Config File

Config.xml is used as TCCSim's configuration file and is located in the same path as the executable. The config contains data for the Connect Dialog(Section 1.1) so that complete information will not have to be entered each time the dialog is opened.

```

<TrainID>2</TrainID>
<Connect>127.0.0.1:45000</Connect>
<Timer>2</Timer>
<Timeout>5</Timeout>

```

The config also contains data that aids in keeping the protocol specification as dynamical as possible. Default PositionReportRequest and DriverLogonStatus files are specified here. The

message type value for when the default DriverLogonStatus message should be sent can be selected with 'DIValue'

```
<PRRMessage>DefaultPRR.xml</PRRMessage>
<DLSMessage>DefaultDriverLogonStatus.xml</DLSMessage>
<DIValue>128</DIValue>
```

The status tabs in the main view is updated when a PositionReport is received. The message type value for when to update the information can be selected as well as what the the name of the error blocks is.

```
<PRValue>132</PRValue>
<PRErrorBlock>ERROR_MESSAGE_DATA</PRErrorBlock>
```

TCCSim will not accept message that are longer than a certain length. The length is specified with BufferLength. That is the length including all safety header fields.

```
<BufferLength>1000</BufferLength>
```

A complete config follows.

```
<?xml version="1.0" encoding="utf-8" ?>
<Config>
  <!-- Connection dialog default data-->
  <TrainID>2</TrainID>
  <!--<Connect>10.160.241.92:45000</Connect>-->
  <Connect>127.0.0.1:45000</Connect>
  <Timer>2</Timer>
  <Timeout>5</Timeout>

  <!-- All values below are necessary for TCCSim to work properly -->

  <!-- Default PRR message in Messages folder -->
  <PRRMessage>DefaultPRR.xml</PRRMessage>

  <!-- Default DriverLogonStatus message in Messages folder
       sent when message with value DIValue is received -->
  <DLSMessage>DefaultDriverLogonStatus.xml</DLSMessage>
  <DIValue>128</DIValue>

  <!-- PositionReport value and error block -->
  <PRValue>132</PRValue>
  <PRErrorBlock>ERROR_MESSAGE_DATA</PRErrorBlock>

  <!-- Max length of received messages. Safety header + Message + CRC.
       This is not bound to the length of outbound messages -->
  <BufferLength>1000</BufferLength>
</Config>
```

7 ErrorCodes.xml

In ErrorCodes.xml the descriptions of error codes are stored. If PositionReport is received with an error, the code is translated using this file. The file has the following

```
<Root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Row>
    <ST_Error_Code>50000</ST_Error_Code>
    <Description>Invalid bool </Description>
  </Row>
  <Row>
    <ST_Error_Code>50259</ST_Error_Code>
    <Description>All A/B mismatches in ACT</Description>
  </Row>
</Root>
```

structure:

It was built from an Excel file. The XML data was extracted using XML Tools which is an Excel Add-In and can, at the time of writing, be found here:

<http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=3108>

Use "Convert a Range to an XML List" and select the range which should be converted. When selecting the range only one block can be selected and extracted, like this example:

	A	B	C	D	E	F	G
1							English
2	Data sent to stationary		Error Codes				Uk
3	ST Error Code	To ST	Blockname	Errorcode	Grade	File	Description
4	50000	X	UndefBlock	0	Log	ABVER	Invalid bool
5	50259	X	ACT	3	Log	ACTCA	All A/B mismatches in ACT
6	50265	X	ACT	9	Log	ACT02A	Balise outside window (slip detected)
7	50281	X	ACT	25	Log	ACT01A	Stop train rec. in not allowed mode!
8	50289	X	ACT	33	Log	ACT02A	First missed balise
9	50314	X	ACT	58	Log	ACT01A	Shunting not possible for special vehicle
10	50326	X	ACT	70	Log	ACT01A	Fire Alarm Locomotive
11	50516	X	BRK	4	Log	BRK01CA	A/B diff, serviceBrakeFlag
12	50517	X	BRK	5	Log	BRK01CA	A/B diff, emergencyBrakeFlag
13	50518	X	BRK	6	Log	BRK01CA	A/B diff, flashBrakeButtonFlag
14	50519	X	BRK	7	Log	BRK02CA	A/B diff, emergencyBrakeFlag
15	50520	X	BRK	8	Log	BRK02CA	A/B diff, flashBrakeButtonFlag
16	50522	X	BRK	10	Log	BRK01CA	A/B diff, mmiReleasableFlag
17	50523	X	BRK	11	Log	BRK01CA	A/B diff, dispReleasableFlag

The headers might need to be changed. When the range is made into an XML List it can be right clicked -> XML -> Export which should give a file looking like this:

```

<Root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Row>
    <ST_Error_Code>50000</ST_Error_Code>
    <To_ST>X</To_ST>
    <Blockname>UndefBlock</Blockname>
    <Errorcode>0</Errorcode>
    <Grade>Log</Grade>
    <File>ABVER</File>
    <Description>Invalid bool </Description>
  </Row>
  <Row>
    <ST_Error_Code>50259</ST_Error_Code>
    <To_ST>X</To_ST>
    <Blockname>ACT</Blockname>
    <Errorcode>3</Errorcode>
    <Grade>Log</Grade>
    <File>ACTCA</File>
    <Description>All A/B mismatches in ACT</Description>
  </Row>

```

This file needs some clean up. An example in vim/gvim will be explained here.

A lot of rows has to be changed so a recording should be the best option. For example:

- Type “/To_ST” - Search for “To_ST”
- Press ‘q’ followed by ‘a’. - Start a recording and store it at button a.
- Press ‘n’ - Search the next “To_ST”
- Press ‘5dd’ - Deletes the next 5 rows.
- Press ‘q’ - Stop recording.

The recording will now be stored at button ‘a’ and can be played by typing “@a” or in our case played, for example, 400 times: “400@a”.

Save and quit: “:wq”

There is also a sample tool for this purpose. ErrorCodeCleanup.

It is used from the command line with the arguments <in file> <out file>.

8 Building TCCSim for TCP

TCCSim is be built for communication over TCP.

When building Release version (F6) all necessary files are copied to the release folder.

9 TCCSim remote

The possibility to alter TCCSim for remote management instead of a GUI have been researched. The result of this can be found in the project TCCRemote. This is not a usable version of TCCSim, yet. It can thou be expanded from the example provided. It is recommended that a copy of the latest version of TCCSim is created and modifications of the user interface from TCCRemote is added so that the latest version of message handling is ensured.

This is done by removing main view and message view from the TCCSim project, add Server.cs and infer the modifications in Controller.cs from TCCRemote to the new project.