PHASE 1

Interface design,
Test-driven development,
Functional test design, and
Unit testing and jUnit tool

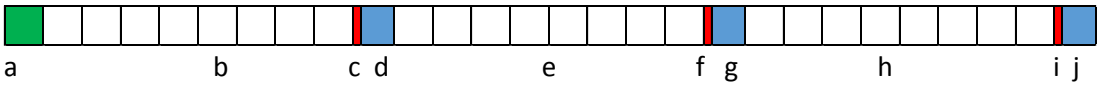
Team members:
Kristiyan Dimitrov
Martina Freiholtz
Kai Salmon
Aksel Strömberg

Part 1: Interface and test design

Packet structure

Our packets with torque_data, sensor_data and ir_data are always byte arrays consisting of 28 bytes.

a. An 8-bit start delimiter (green)
b. The torque data (8 bytes)
c. Control bit for torque data
d. Torque delimiter (7 bits)
e. The ultra_sensor data (8 bytes)
f. Control bit for ultra_sensor data
g. Ultra_sensor delimiter (7 bit)
h. The ir_sensor data (8 bytes)
i. Control bit for ir_sensor data
j. Ir_sensor delimiter (7 bit)



The delimiters are constants (different for each delimiter in a packet), and the ir delimiter marks the end of a packet. The received double values are first converted into binary and returned as an array of numeric representations of each byte.

The control bit (the most significant bit of adjacent delimiter) will be 0 if there are an even number of 1-bits in the preceding value, 1 otherwise. Hence, if the control bit is 1, the numerical representation of the delimiter will be deprecated by 128.

Descriptions and classification trees

getSensorData

Description:
Converts a SensorData object into a bitstream.

Pre-condition:
sensorData is an object with 3 8byte double precision floating point values.

Post-condition:
The returned value is a 28byte byte stream in the following format:
[8bit start_delimiter]
[8byte torque double][check bit for torque][7bit torque delimiter]
[8byte ultra_dist double][check bit for ultra_dist][7bit ultra_dist delimiter]
[8byte ir_dist double][check bit for ir_dist][7bit ir_dist delimiter]
The check bit is 1 if there are an odd number of 1-bits in the corresponding double, and otherwise 0.

Test case:
An arbitrary SensorData object returns the correct, pre-calculated stream.

SensorData:isValidStream

Description:

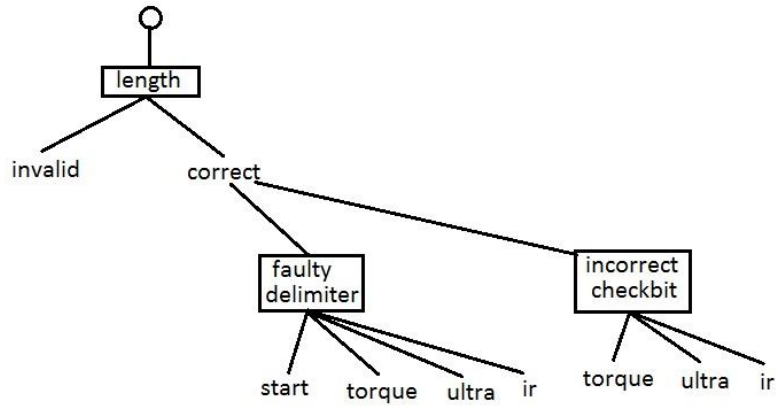
Checks if a given ByteArrayOutputStream is in a valid format.

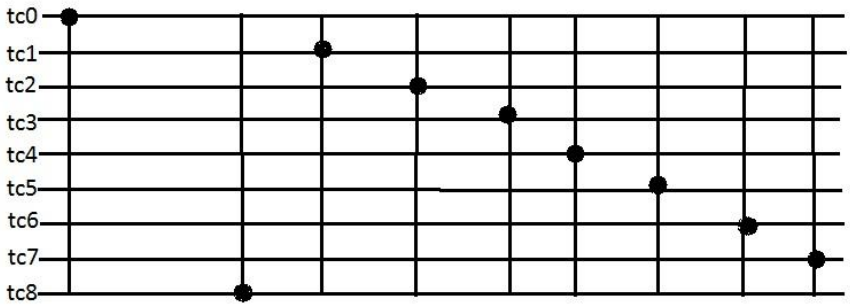
Pre-condition:

A non-null ByteArrayOutputStream is given.

Post-condition:

Returns true if the stream is valid, and otherwise returns false.





readTorqueSpeed

Description:

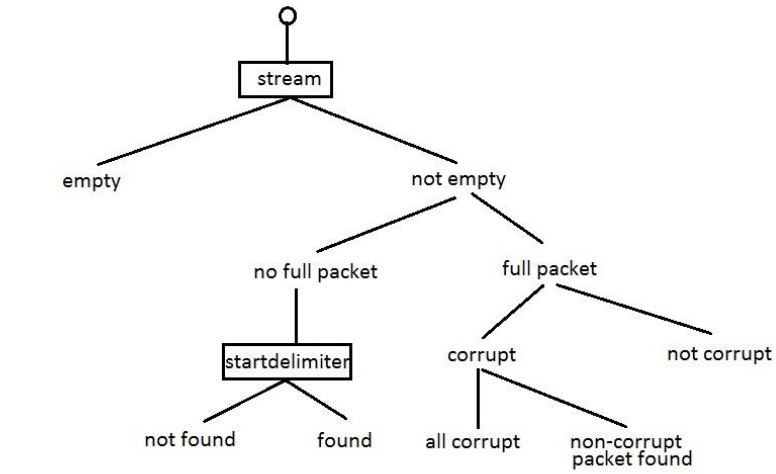
Returns an object with torque and speed values, after reading the first available non-corrupt data packet in the stream it receives.

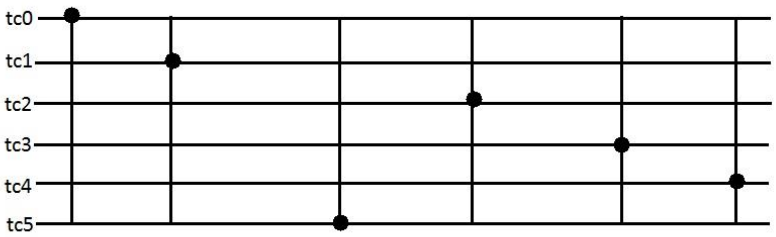
Pre-condition:

Receives a stream of type ByteArrayOutputStream which contains one or more packets.

Post-condition:

A reference to an object with speed and torque (given as doubles), or else a reference to an object with the value -1 for speed and torque.





SpeedTorque:isValidStream

Description:

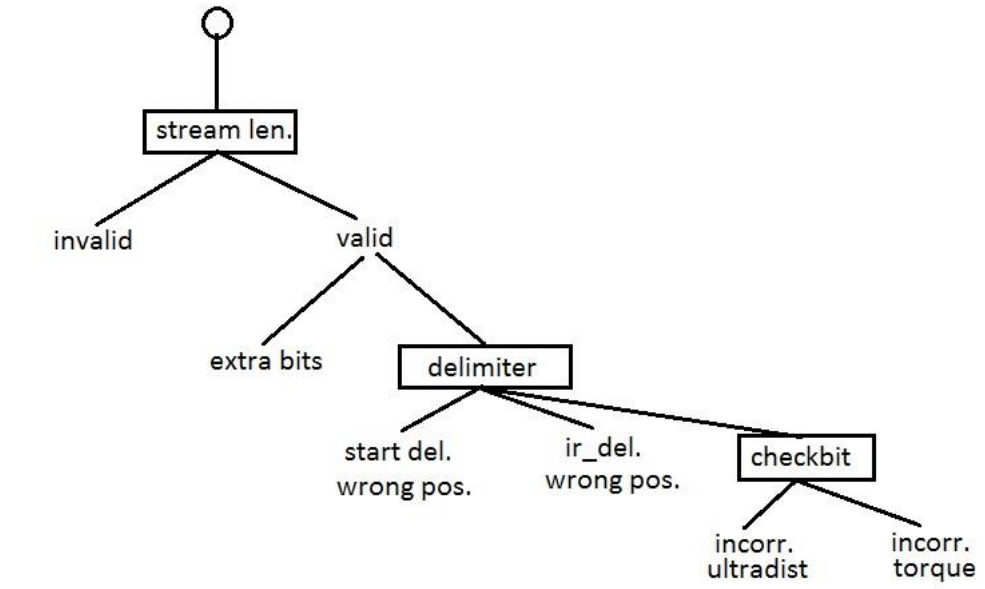
Checks if a given ByteArrayOutputStream is in a valid format.

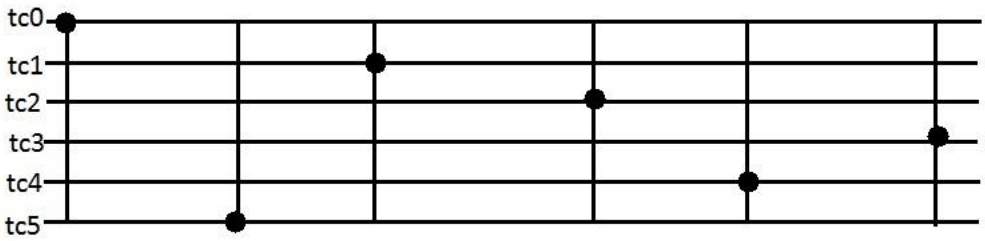
Pre-condition:

A non-null ByteArrayOutputStream is given.

Post-condition:

Returns true if the stream is valid, and otherwise returns false.





writeBits

Description:

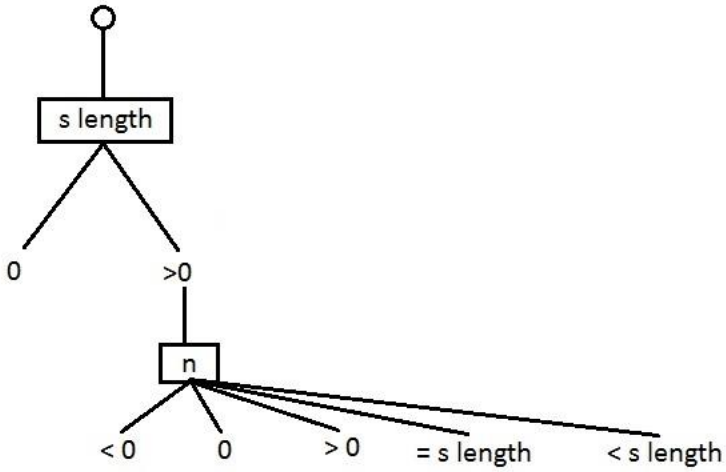
Appends first n number of bits from bitstream 'stream' into the end of the input buffer bitstream.
Since we are working with bits (instead of bytes, as in our other functions), we are letting a string represent our bitstream.

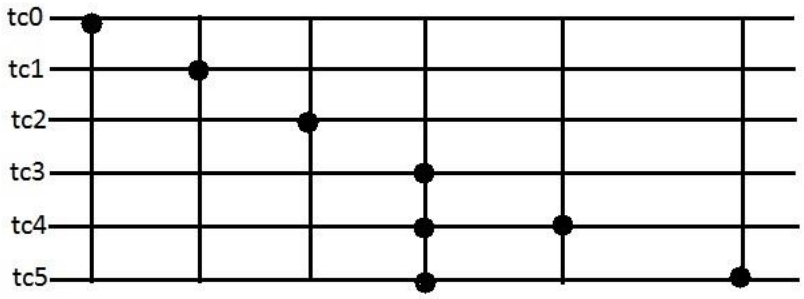
Pre-condition:

'n' is non-negative, 's' is not empty and contains at least 'n' bits

Post-condition:

All 'n' bits have been inserted at the end of the input buffer bitstream. 0 is returned if successful. If not, 1 is returned.





readBits

Description:

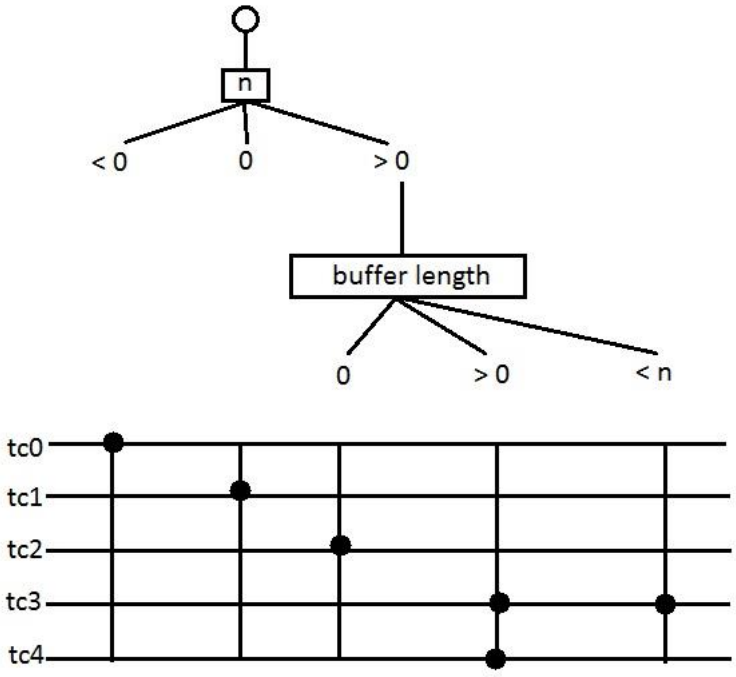
Removes 'n' bits from the beginning of the output buffer stream.
Since we are working with bits here (as opposed to bytes, as in our other functions), we use a string to represent a bitstream.

Pre-condition:

'n' is non-negative and smaller than or equal to the length of the output buffer stream.

Post-condition:

'n' bits are removed from the buffer, and an object containing an error code (0 if successful, 1 otherwise) and the removed bits are returned.



Part 2: Test-driven development

For each of the functions described above, jUnit tests were designed after each test case the function needed to pass.

After this, the functions were implemented to pass one test case at a time. The group members mainly worked in the same room to discuss and build on each other's' solutions. A jUnit test and the corresponding function(s) were not always developed by the same individual.

See comments in code for further information.

Part 3: Member contribution

Test Cases (Planning and jUnit Development):

SensorDataTest.*testGetSensorData:*	Kai Salmon and Martina Freiholtz
SensorDataTest.*testIsValidStream*:	Kristiyan Dimitrov and Kai Salmon
SensorDataTest.*testHas_even_bits*:	Kai Salmon
ReadWriteBitsTest.*testWriteBits*:	Kai Salmon
ReadWriteBitsTest.*testReadBits*:	Martina Freiholtz
SpeedTorqueTest.*testReadSpeedTorque*:	Martina Freiholtz
SpeedTorqueTest.*testIsValidStream*:	Kai Salmon

Development:

ReadData:	Martina Freiholtz
ReadWriteBits.*writeBits***:**	Kristiyan Dimitrov
ReadWriteBits.*readBits*:	Martina Freiholtz
SensorData:	Kai Salmon
SpeedTorque.*readSpeedTorque***:**	Martina Freiholtz
SpeedTorque.*packetStartAt***:**	Martina Freiholtz
SpeedTorque.*isValidStream***:**	Kai Salmon
SpeedTorque.*toDouble***:**	Kai Salmon
SpeedTorqueObj:	Martina Freiholtz
TavAssignOne:	Kai Salmon

Documentation:

Martina Freiholtz

(Classification trees translated from physical drawing to digital through the magic of MS Paint. No shame.)