f1-2019-telemetry

Release 1.1.4

Sidney Cadot

Oct 09, 2019

CONTENTS

The *f1-2019-telemetry* package provides support for interpreting telemetry information as sent out over the network by the F1 2019 game by CodeMasters. It also provides *command line tools* to record, playback, and monitor F1 2019 session data.

CHAPTER

ONE

PROJECT INFORMATION

The *f1-2019-telemetry* package and its documentation are currently at version **1.1.4**.

The project is distributed as a standard *wheel* package on PyPI. This allows installation using the standard Python 3 *pip* tool as follows:

```
pip install f1-2019-telemetry
```

The project source code is hosted as a Git repository on GitLab:

https://gitlab.com/reddish/f1-2019-telemetry/

The pip-installable package is hosted on PyPI:

https://pypi.org/project/f1-2019-telemetry/

The documentation is hosted on Read the Docs:

https://f1-2019-telemetry.readthedocs.io/en/latest/

CHAPTER

TWO

DOCUMENTATION

The documentation comes in two parts:

- The *Package Documentation* provides guidance on installation and usage of the f1-2019-telemetry package, and documents the included command-line tools.
- The *F1 2019 Telemetry Packet Specification* is a non-authoritative copy of the CodeMasters telemetry packet specification, with some corrections applied.

2.1 Package Documentation

The *f1-2019-telemetry* package provides support for interpreting telemetry information as sent out over the network by the F1 2019 game by CodeMasters. It also provides *command line tools* to record, playback, and monitor F1 2019 session data.

With each yearly release of the F1 series game, CodeMasters post a descripton of the corresponding telemetry packet format on their forum. For F1 2019, the packet format is described here:

> https://forums.codemasters.com/topic/38920-f1-2019-udp-specification/

A formatted version of this specification, with some small issues fixed, is included in the *f1-2019-telemetry* package and can be found *here*.

The *f1-2019-telemetry* package should work on Python 3.6 and above.

2.1.1 Installation

The f1-2019-telemetry package is hosted on PyPI. To install it in your Python 3 environment, type:

```
pip3 install f1-2019-telemetry
```

When this completes, you should be able to start your Python 3 interpreter and execute this:

```
import f1_2019_telemetry.packet

help(f1_2019_telemetry.packet)
```

Apart from the *f1_2019_telemetry* package (and its main module *f1_2019_telemetry.packet*), the `pip3 install` command will also install some command-line utilities that can be used to record, playback, and monitor F1 2019 telemetry data. Refer to the *Command Line Tools* section for more information.

2.1.2 Usage

If you want to write your own Python script to process F1 2019 telemetry data, you will need to set up the reception of UDP packets yourself. After that, use the function *unpack_udp_packet()* to unpack the binary packet to an appropriate object with all the data fields present.

A minimalistic example is as follows:

```
import socket

from f1_2019_telemetry.packets import unpack_udp_packet

udp_socket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
udp_socket.bind(('', 20777))

while True:
    udp_packet = udp_socket.recv(2048)
    packet = unpack_udp_packet(udp_packet)
    print("Received:", packet)
    print()
```

This example opens a UDP socket on port 20777, which is the default port that the F1 2019 game uses to send packages; it then waits for packages and, upon reception, prints their full contents.

To generate some data, start your F1 2019 game, and go to the Telemetry Settings (these can be found under Game Options / Settings).

- Make sure that the *UDP Telemetry* setting is set to *On*.
- The *UDP Broadcast* setting should be either set to *On*, or it should be set to *Off*, and then the *UDP IP Address* setting should be set to the IP address of the computer on which you intend to run the Python script that will capture game session data. For example, if you want the Python script to run on the same computer that runs the game, and you don't want to send out UDP packets to all devices in your home network, you can set the *UDP Broadcast* setting to *Off* and the *UDP IP Address* setting to *127.0.0.1*.
- The *UDP Port* setting can be keep its default value of *20777*.
- The *UDP Send Rate* setting can be set to *60*, assuming you have a sufficiently powerful computer to run the game.
- The *UDP Format* setting should be set to *2019*.

Now, if you start a race session with the Python script given above running, you should see a continuous stream of game data being printed to your command line terminal.

The example script given above is about as simple as it can be to capture game data. For more elaborate examples, check the source code of the provided *f1_2019_telemetry.cli.monitor* and *f1_2019_telemetry.cli.recorder* scripts. Note that those examples are considerably more complicated because they use multi-threading.

2.1.3 Command Line Tools

The f1-2019-telemetry package installs three command-line tools that provide basic recording, playback, and session monitoring support. Below, we reproduce their command-line help for reference.

f1-2019-telemetry-recorder script

```
usage: f1-2019-telemetry-recorder [-h] [-p PORT] [-i INTERVAL]

Record F1 2019 telemetry data to SQLite3 files.

optional arguments:
  -h, --help                          show this help message and exit
  -p PORT, --port PORT                UDP port to listen to (default: 20777)
  -i INTERVAL, --interval INTERVAL    interval for writing incoming data to SQLite3␣
↪file, in seconds (default: 1.0)
```

f1-2019-telemetry-player script

```
usage: f1-2019-telemetry-player [-h] [-r REALTIME_FACTOR] [-d DESTINATION] [-p PORT]␣
↪filename

Replay an F1 2019 session as UDP packets.

positional arguments:
  filename                                  SQLite3 file to replay packets from

optional arguments:
  -h, --help                                show this help message and exit
  -r REALTIME_FACTOR, --rtf REALTIME_FACTOR    playback real-time factor (higher is␣
↪faster, default=1.0)
  -d DESTINATION, --destination DESTINATION    destination UDP address; omit to use␣
↪broadcast (default)
  -p PORT, --port PORT                      destination UDP port (default: 20777)
```

f1-2019-telemetry-monitor script

```
usage: f1-2019-telemetry-monitor [-h] [-p PORT]

Monitor UDP port for incoming F1 2019 telemetry data and print information.

optional arguments:
  -h, --help              show this help message and exit
  -p PORT, --port PORT    UDP port to listen to (default: 20777)
```

2.1.4 Package Source Code

The source code of all modules in the package is pretty well documented and easy to follow. We reproduce it here for reference.

Module: f1_2019_telemetry.packets

Module *f1_2019_telemetry.packets* is the main module of the package. It implements ctypes *struct* types for all kinds of packets, and it implements the *unpack_udp_packet()* function that take the contents of a raw UDP packet and interprets is as the appropriate telemetry packet, if possible.

```
"""F1 2019 UDP Telemetry support package

This package is based on the CodeMasters Forum post documenting the F1 2019 packet format:

    https://forums.codemasters.com/topic/38920-f1-2019-udp-specification/

Compared to the definitions given there, the Python version has the following changes:

(1) In the 'PacketMotionData' structure, the comments for the three m_angularAcceleration{X,Y,Z} fields erroneously
    refer to 'velocity' rather than 'acceleration'. This was corrected.
(2) In the 'CarSetupData' structure, the comment of the m_rearAntiRollBar refer to rear instead of front. This was corrected.
(3) In the Driver IDs table, driver 34 has name "Wilheim Kaufmann".
    This is a typo; whenever this driver is encountered in the game, his name is given as "Wilhelm Kaufmann".
"""

import ctypes
import enum

##########################################################
#                                                        #
#  ___________  PackedLittleEndianStructure  ___________  #
#                                                        #
##########################################################

class PackedLittleEndianStructure(ctypes.LittleEndianStructure):
    """The standard ctypes LittleEndianStructure, but tightly packed (no field padding), and with a proper repr() function.

    This is the base type for all structures in the telemetry data.
    """
    _pack_ = 1

    def __repr__(self):
        fstr_list = []
        for (fname, ftype) in self._fields_:
            value = getattr(self, fname)
            if isinstance(value, (PackedLittleEndianStructure, int, float, bytes)):
                vstr = repr(value)
            elif isinstance(value, ctypes.Array):
                vstr = "[{}]".format(", ".join(repr(e) for e in value))
            else:
                raise RuntimeError("Bad value {!r} of type {!r}".format(value, type(value)))
            fstr = "{}={}".format(fname, vstr)
            fstr_list.append(fstr)
        return "{}({})".format(self.__class__.__name__, ", ".join(fstr_list))


#########################################
#                                       #
#  ___________  Packet Header  ___________  #
#                                       #
#########################################
```

(continues on next page)

(continued from previous page)

```
52
53  class PacketHeader(PackedLittleEndianStructure):
54      """The header for each of the UDP telemetry packets."""
55      _fields_ = [
56          ('packetFormat'     , ctypes.c_uint16),  # 2019
57          ('gameMajorVersion' , ctypes.c_uint8 ),  # Game major version - "X.00"
58          ('gameMinorVersion' , ctypes.c_uint8 ),  # Game minor version - "1.XX"
59          ('packetVersion'    , ctypes.c_uint8 ),  # Version of this packet type, all␣
    ↪start from 1
60          ('packetId'         , ctypes.c_uint8 ),  # Identifier for the packet type,␣
    ↪see below
61          ('sessionUID'       , ctypes.c_uint64),  # Unique identifier for the session
62          ('sessionTime'      , ctypes.c_float ),  # Session timestamp
63          ('frameIdentifier'  , ctypes.c_uint32),  # Identifier for the frame the data␣
    ↪was retrieved on
64          ('playerCarIndex'   , ctypes.c_uint8 )   # Index of player's car in the array
65      ]
66
67
68  @enum.unique
69  class PacketID(enum.IntEnum):
70      """Value as specified in the PacketHeader.packetId header field, used to␣
    ↪distinguish packet types."""
71
72      MOTION         = 0
73      SESSION        = 1
74      LAP_DATA       = 2
75      EVENT          = 3
76      PARTICIPANTS   = 4  # 0.2 Hz (once every five seconds)
77      CAR_SETUPS     = 5
78      CAR_TELEMETRY  = 6
79      CAR_STATUS     = 7
80
81
82  PacketID.short_description = {
83      PacketID.MOTION         : 'Motion',
84      PacketID.SESSION        : 'Session',
85      PacketID.LAP_DATA       : 'Lap Data',
86      PacketID.EVENT          : 'Event',
87      PacketID.PARTICIPANTS   : 'Participants',
88      PacketID.CAR_SETUPS     : 'Car Setups',
89      PacketID.CAR_TELEMETRY  : 'Car Telemetry',
90      PacketID.CAR_STATUS     : 'Car Status'
91  }
92
93
94  PacketID.long_description = {
95      PacketID.MOTION         : 'Contains all motion data for player\'s car - only sent␣
    ↪while player is in control',
96      PacketID.SESSION        : 'Data about the session - track, time left',
97      PacketID.LAP_DATA       : 'Data about all the lap times of cars in the session',
98      PacketID.EVENT          : 'Various notable events that happen during a session',
99      PacketID.PARTICIPANTS   : 'List of participants in the session, mostly relevant␣
    ↪for multiplayer',
100     PacketID.CAR_SETUPS     : 'Packet detailing car setups for cars in the race',
101     PacketID.CAR_TELEMETRY  : 'Telemetry data for all cars',
102     PacketID.CAR_STATUS     : 'Status data for all cars such as damage'
```

(continues on next page)

(continued from previous page)

```
}

//////////////////////////////////////////////////////////
//                                                      //
//  ___________  Packet ID 0 : MOTION PACKET  ___________  //
//                                                      //
//////////////////////////////////////////////////////////

class CarMotionData_V1(PackedLittleEndianStructure):
    """This type is used for the 20-element 'carMotionData' array of the PacketMotionData_V1 type, defined below."""
    _fields_ = [
        ('worldPositionX'     , ctypes.c_float),  # World space X position
        ('worldPositionY'     , ctypes.c_float),  # World space Y position
        ('worldPositionZ'     , ctypes.c_float),  # World space Z position
        ('worldVelocityX'     , ctypes.c_float),  # Velocity in world space X
        ('worldVelocityY'     , ctypes.c_float),  # Velocity in world space Y
        ('worldVelocityZ'     , ctypes.c_float),  # Velocity in world space Z
        ('worldForwardDirX'   , ctypes.c_int16),  # World space forward X direction (normalised)
        ('worldForwardDirY'   , ctypes.c_int16),  # World space forward Y direction (normalised)
        ('worldForwardDirZ'   , ctypes.c_int16),  # World space forward Z direction (normalised)
        ('worldRightDirX'     , ctypes.c_int16),  # World space right X direction (normalised)
        ('worldRightDirY'     , ctypes.c_int16),  # World space right Y direction (normalised)
        ('worldRightDirZ'     , ctypes.c_int16),  # World space right Z direction (normalised)
        ('gForceLateral'      , ctypes.c_float),  # Lateral G-Force component
        ('gForceLongitudinal' , ctypes.c_float),  # Longitudinal G-Force component
        ('gForceVertical'     , ctypes.c_float),  # Vertical G-Force component
        ('yaw'                , ctypes.c_float),  # Yaw angle in radians
        ('pitch'              , ctypes.c_float),  # Pitch angle in radians
        ('roll'               , ctypes.c_float)   # Roll angle in radians
    ]


class PacketMotionData_V1(PackedLittleEndianStructure):
    """The motion packet gives physics data for all the cars being driven.

    There is additional data for the car being driven with the goal of being able to drive a motion platform setup.

    N.B. For the normalised vectors below, to convert to float values divide by 32767.0f - 16-bit signed values are
    used to pack the data and on the assumption that direction values are always between -1.0f and 1.0f.

    Frequency: Rate as specified in menus
    Size: 1343 bytes
    Version: 1
    """
    _fields_ = [
        ('header'                , PacketHeader          ),  # Header
        ('carMotionData'         , CarMotionData_V1 * 20),  # Data for all cars on track
```

(continues on next page)

(continued from previous page)

```
        # Extra player car ONLY data
        ('suspensionPosition'     , ctypes.c_float * 4   ),  # Note: All wheel arrays have the following order:
        ('suspensionVelocity'     , ctypes.c_float * 4   ),  # RL, RR, FL, FR
        ('suspensionAcceleration' , ctypes.c_float * 4   ),  # RL, RR, FL, FR
        ('wheelSpeed'             , ctypes.c_float * 4   ),  # Speed of each wheel
        ('wheelSlip'              , ctypes.c_float * 4   ),  # Slip ratio for each wheel
        ('localVelocityX'         , ctypes.c_float       ),  # Velocity in local space
        ('localVelocityY'         , ctypes.c_float       ),  # Velocity in local space
        ('localVelocityZ'         , ctypes.c_float       ),  # Velocity in local space
        ('angularVelocityX'       , ctypes.c_float       ),  # Angular velocity x-component
        ('angularVelocityY'       , ctypes.c_float       ),  # Angular velocity y-component
        ('angularVelocityZ'       , ctypes.c_float       ),  # Angular velocity z-component
        ('angularAccelerationX'   , ctypes.c_float       ),  # Angular acceleration x-component
        ('angularAccelerationY'   , ctypes.c_float       ),  # Angular acceleration y-component
        ('angularAccelerationZ'   , ctypes.c_float       ),  # Angular acceleration z-component
        ('frontWheelsAngle'       , ctypes.c_float       )   # Current front wheels angle in radians
    ]

##############################################################
#                                                            #
#  ___________  Packet ID 1 : SESSION PACKET  ___________   #
#                                                            #
##############################################################

class MarshalZone_V1(PackedLittleEndianStructure):
    """This type is used for the 21-element 'marshalZones' array of the PacketSessionData_V1 type, defined below."""
    _fields_ = [
        ('zoneStart' , ctypes.c_float),  # Fraction (0..1) of way through the lap the marshal zone starts
        ('zoneFlag'  , ctypes.c_int8 )   # -1 = invalid/unknown, 0 = none, 1 = green, 2 = blue, 3 = yellow, 4 = red
    ]


class PacketSessionData_V1(PackedLittleEndianStructure):
    """The session packet includes details about the current session in progress.

    Frequency: 2 per second
    Size: 149 bytes
    Version: 1
    """
    _fields_ = [
        ('header'              , PacketHeader       ),  # Header
        ('weather'             , ctypes.c_uint8     ),  # Weather - 0 = clear, 1 = light cloud, 2 = overcast
                                                        # 3 = light rain, 4 = heavy rain, 5 = storm
```

(continues on next page)

(continued from previous page)

```
        ('trackTemperature'    , ctypes.c_int8      ),  # Track temp. in degrees celsius
        ('airTemperature'      , ctypes.c_int8      ),  # Air temp. in degrees celsius
        ('totalLaps'           , ctypes.c_uint8     ),  # Total number of laps in this race
        ('trackLength'         , ctypes.c_uint16    ),  # Track length in metres
        ('sessionType'         , ctypes.c_uint8     ),  # 0 = unknown, 1 = P1, 2 = P2, 3 = P3, 4 = Short P
                                                        # 5 = Q1, 6 = Q2, 7 = Q3, 8 = Short Q, 9 = OSQ
                                                        # 10 = R, 11 = R2, 12 = Time Trial
        ('trackId'             , ctypes.c_int8      ),  # -1 for unknown, 0-21 for tracks, see appendix
        ('m_formula'           , ctypes.c_uint8     ),  # Formula, 0 = F1 Modern, 1 = F1 Classic, 2 = F2,
                                                        # 3 = F1 Generic
        ('sessionTimeLeft'     , ctypes.c_uint16    ),  # Time left in session in seconds
        ('sessionDuration'     , ctypes.c_uint16    ),  # Session duration in seconds
        ('pitSpeedLimit'       , ctypes.c_uint8     ),  # Pit speed limit in kilometres per hour
        ('gamePaused'          , ctypes.c_uint8     ),  # Whether the game is paused
        ('isSpectating'        , ctypes.c_uint8     ),  # Whether the player is spectating
        ('spectatorCarIndex'   , ctypes.c_uint8     ),  # Index of the car being spectated
        ('sliProNativeSupport' , ctypes.c_uint8     ),  # SLI Pro support, 0 = inactive, 1 = active
        ('numMarshalZones'     , ctypes.c_uint8     ),  # Number of marshal zones to follow
        ('marshalZones'        , MarshalZone_V1 * 21),  # List of marshal zones - max 21
        ('safetyCarStatus'     , ctypes.c_uint8     ),  # 0 = no safety car, 1 = full safety car
                                                        # 2 = virtual safety car
        ('networkGame'         , ctypes.c_uint8     )   # 0 = offline, 1 = online
    ]

############################################################
#                                                          #
#  ___________  Packet ID 2 : LAP DATA PACKET  ___________  #
#                                                          #
############################################################

class LapData_V1(PackedLittleEndianStructure):
    """This type is used for the 20-element 'lapData' array of the PacketLapData_V1 type, defined below."""
    _fields_ = [

        ('lastLapTime'       , ctypes.c_float),  # Last lap time in seconds
        ('currentLapTime'    , ctypes.c_float),  # Current time around the lap in seconds
        ('bestLapTime'       , ctypes.c_float),  # Best lap time of the session in seconds
        ('sector1Time'       , ctypes.c_float),  # Sector 1 time in seconds
        ('sector2Time'       , ctypes.c_float),  # Sector 2 time in seconds
```

(continues on next page)

(continued from previous page)

```
        ('lapDistance'       , ctypes.c_float),  # Distance vehicle is around current lap in metres - could
                                                 # be negative if line hasn't been crossed yet
        ('totalDistance'     , ctypes.c_float),  # Total distance travelled in session in metres - could
                                                 # be negative if line hasn't been crossed yet
        ('safetyCarDelta'    , ctypes.c_float),  # Delta in seconds for safety car
        ('carPosition'       , ctypes.c_uint8),  # Car race position
        ('currentLapNum'     , ctypes.c_uint8),  # Current lap number
        ('pitStatus'         , ctypes.c_uint8),  # 0 = none, 1 = pitting, 2 = in pit area
        ('sector'            , ctypes.c_uint8),  # 0 = sector1, 1 = sector2, 2 = sector3
        ('currentLapInvalid' , ctypes.c_uint8),  # Current lap invalid - 0 = valid, 1 = invalid
        ('penalties'         , ctypes.c_uint8),  # Accumulated time penalties in seconds to be added
        ('gridPosition'      , ctypes.c_uint8),  # Grid position the vehicle started the race in
        ('driverStatus'      , ctypes.c_uint8),  # Status of driver - 0 = in garage, 1 = flying lap
                                                 # 2 = in lap, 3 = out lap, 4 = on track
        ('resultStatus'      , ctypes.c_uint8)   # Result status - 0 = invalid, 1 = inactive, 2 = active
                                                 # 3 = finished, 4 = disqualified, 5 = not classified
                                                 # 6 = retired
    ]


class PacketLapData_V1(PackedLittleEndianStructure):
    """The lap data packet gives details of all the cars in the session.

    Frequency: Rate as specified in menus
    Size: 843 bytes
    Version: 1
    """
    _fields_ = [
        ('header'  , PacketHeader   ),  # Header
        ('lapData' , LapData_V1 * 20)   # Lap data for all cars on track
    ]

##########################################################
#                                                        #
#  __________  Packet ID 3 : EVENT PACKET  __________    #
#                                                        #
##########################################################

class PacketEventData_V1(PackedLittleEndianStructure):
    """This packet gives details of events that happen during the course of a session.

    Frequency: When the event occurs
    Size: 32 bytes
    Version: 1
```

(continues on next page)

(continued from previous page)

```
    """
    _fields_ = [
        ('header'          , PacketHeader     ),  # Header
        ('eventStringCode' , ctypes.c_char * 4),  # Event string code, see below
        # Event details - should be interpreted differently for each type
        ('vehicleIdx'      , ctypes.c_uint8   ),  # Vehicle index of car (valid for events: FTLP, RTMT, TMPT, RCWN)
        ('lapTime'         , ctypes.c_float   )   # Lap time is in seconds (valid for events: FTLP)
    ]


@enum.unique
class EventStringCode(enum.Enum):
    """Value as specified in the PacketEventData_V1.eventStringCode header field, used to distinguish packet types."""
    SSTA = b'SSTA'
    SEND = b'SEND'
    FTLP = b'FTLP'
    RTMT = b'RTMT'
    DRSE = b'DRSE'
    DRSD = b'DRSD'
    TMPT = b'TMPT'
    CHQF = b'CHQF'
    RCWN = b'RCWN'


EventStringCode.short_description = {
    EventStringCode.SSTA : 'Session Started',
    EventStringCode.SEND : 'Session Ended',
    EventStringCode.FTLP : 'Fastest Lap',
    EventStringCode.RTMT : 'Retirement',
    EventStringCode.DRSE : 'DRS enabled',
    EventStringCode.DRSD : 'DRS disabled',
    EventStringCode.TMPT : 'Team mate in pits',
    EventStringCode.CHQF : 'Chequered flag',
    EventStringCode.RCWN : 'Race Winner'
}


EventStringCode.long_description = {
    EventStringCode.SSTA : 'Sent when the session starts',
    EventStringCode.SEND : 'Sent when the session ends',
    EventStringCode.FTLP : 'When a driver achieves the fastest lap',
    EventStringCode.RTMT : 'When a driver retires',
    EventStringCode.DRSE : 'Race control have enabled DRS',
    EventStringCode.DRSD : 'Race control have disabled DRS',
    EventStringCode.TMPT : 'Your team mate has entered the pits',
    EventStringCode.CHQF : 'The chequered flag has been waved',
    EventStringCode.RCWN : 'The race winner is announced'
}

##############################################################
#                                                            #
#  ___________  Packet ID 4 : PARTICIPANTS PACKET  ___________  #
#                                                            #
##############################################################
```

(continues on next page)

(continued from previous page)

```
330
331 class ParticipantData_V1(PackedLittleEndianStructure):
332     """This type is used for the 20-element 'participants' array of the␣
    ↪PacketParticipantsData_V1 type, defined below."""
333     _fields_ = [
334         ('aiControlled' , ctypes.c_uint8    ),  # Whether the vehicle is AI (1) or␣
    ↪Human (0) controlled
335         ('driverId'     , ctypes.c_uint8    ),  # Driver id - see appendix
336         ('teamId'       , ctypes.c_uint8    ),  # Team id - see appendix
337         ('raceNumber'   , ctypes.c_uint8    ),  # Race number of the car
338         ('nationality'  , ctypes.c_uint8    ),  # Nationality of the driver
339         ('name'         , ctypes.c_char * 48),  # Name of participant in UTF-8 format␣
    ↪- null terminated
340                                                 # Will be truncated with ... (U+2026)␣
    ↪if too long
341         ('yourTelemetry', ctypes.c_uint8    )   # The player's UDP setting, 0 =␣
    ↪restricted, 1 = public
342     ]
343
344
345 class PacketParticipantsData_V1(PackedLittleEndianStructure):
346     """This is a list of participants in the race.
347
348     If the vehicle is controlled by AI, then the name will be the driver name.
349     If this is a multiplayer game, the names will be the Steam Id on PC, or the LAN␣
    ↪name if appropriate.
350     On Xbox One, the names will always be the driver name, on PS4 the name will be␣
    ↪the LAN name if playing a LAN game,
351     otherwise it will be the driver name.
352
353     Frequency: Every 5 seconds
354     Size: 1104 bytes
355     Version: 1
356     """
357     _fields_ = [
358         ('header'        , PacketHeader           ),  # Header
359         ('numActiveCars' , ctypes.c_uint8         ),  # Number of active cars in the␣
    ↪data - should match number of
360                                                       # cars on HUD
361         ('participants'  , ParticipantData_V1 * 20)
362     ]
363
364 ##############################################################
365 #                                                            #
366 #  __________  Packet ID 5 : CAR SETUPS PACKET  __________   #
367 #                                                            #
368 ##############################################################
369
370 class CarSetupData_V1(PackedLittleEndianStructure):
371     """This type is used for the 20-element 'carSetups' array of the␣
    ↪PacketCarSetupData_V1 type, defined below."""
372     _fields_ = [
373         ('frontWing'            , ctypes.c_uint8),  # Front wing aero
374         ('rearWing'             , ctypes.c_uint8),  # Rear wing aero
375         ('onThrottle'           , ctypes.c_uint8),  # Differential adjustment on␣
    ↪throttle (percentage)
376         ('offThrottle'          , ctypes.c_uint8),  # Differential adjustment off␣
    ↪throttle (percentage)
```

(continues on next page)

(continued from previous page)

```
        ('frontCamber'           , ctypes.c_float),  # Front camber angle (suspension geometry)
        ('rearCamber'            , ctypes.c_float),  # Rear camber angle (suspension geometry)
        ('frontToe'              , ctypes.c_float),  # Front toe angle (suspension geometry)
        ('rearToe'               , ctypes.c_float),  # Rear toe angle (suspension geometry)
        ('frontSuspension'       , ctypes.c_uint8),  # Front suspension
        ('rearSuspension'        , ctypes.c_uint8),  # Rear suspension
        ('frontAntiRollBar'      , ctypes.c_uint8),  # Front anti-roll bar
        ('rearAntiRollBar'       , ctypes.c_uint8),  # Rear anti-roll bar
        ('frontSuspensionHeight' , ctypes.c_uint8),  # Front ride height
        ('rearSuspensionHeight'  , ctypes.c_uint8),  # Rear ride height
        ('brakePressure'         , ctypes.c_uint8),  # Brake pressure (percentage)
        ('brakeBias'             , ctypes.c_uint8),  # Brake bias (percentage)
        ('frontTyrePressure'     , ctypes.c_float),  # Front tyre pressure (PSI)
        ('rearTyrePressure'      , ctypes.c_float),  # Rear tyre pressure (PSI)
        ('ballast'               , ctypes.c_uint8),  # Ballast
        ('fuelLoad'              , ctypes.c_float)   # Fuel load
    ]


class PacketCarSetupData_V1(PackedLittleEndianStructure):
    """This packet details the car setups for each vehicle in the session.

    Note that in multiplayer games, other player cars will appear as blank, you will only be able to see your car setup and AI cars.

    Frequency: 2 per second
    Size: 843 bytes
    Version: 1
    """
    _fields_ = [
        ('header'    , PacketHeader        ),  # Header
        ('carSetups' , CarSetupData_V1 * 20)
    ]

##############################################################
#                                                            #
#  ___________  Packet ID 6 : CAR TELEMETRY PACKET  ___________  #
#                                                            #
##############################################################

class CarTelemetryData_V1(PackedLittleEndianStructure):
    """This type is used for the 20-element 'carTelemetryData' array of the PacketCarTelemetryData_V1 type, defined below."""
    _fields_ = [
        ('speed'                    , ctypes.c_uint16    ),  # Speed of car in kilometres per hour
        ('throttle'                 , ctypes.c_float     ),  # Amount of throttle applied (0.0 to 1.0)
        ('steer'                    , ctypes.c_float     ),  # Steering (-1.0 (full lock left) to 1.0 (full lock right))
        ('brake'                    , ctypes.c_float     ),  # Amount of brake applied (0 to 1.0)
        ('clutch'                   , ctypes.c_uint8     ),  # Amount of clutch applied (0 to 100)
```

(continues on next page)

(continued from previous page)

```
        ('gear'                    , ctypes.c_int8      ),  # Gear selected (1-8, N=0,
↪ R=-1)
        ('engineRPM'               , ctypes.c_uint16    ),  # Engine RPM
        ('drs'                     , ctypes.c_uint8     ),  # 0 = off, 1 = on
        ('revLightsPercent'        , ctypes.c_uint8     ),  # Rev lights indicator␣
↪(percentage)
        ('brakesTemperature'       , ctypes.c_uint16 * 4),  # Brakes temperature␣
↪(celsius)
        ('tyresSurfaceTemperature' , ctypes.c_uint16 * 4),  # Tyres surface␣
↪temperature (celsius)
        ('tyresInnerTemperature'   , ctypes.c_uint16 * 4),  # Tyres inner temperature␣
↪(celsius)
        ('engineTemperature'       , ctypes.c_uint16    ),  # Engine temperature␣
↪(celsius)
        ('tyresPressure'           , ctypes.c_float  * 4),  # Tyres pressure (PSI)
        ('surfaceType'             , ctypes.c_uint8  * 4)   # Driving surface, see␣
↪appendices
    ]


class PacketCarTelemetryData_V1(PackedLittleEndianStructure):
    """This packet details telemetry for all the cars in the race.

    It details various values that would be recorded on the car such as speed,␣
↪throttle application, DRS etc.

    Frequency: Rate as specified in menus
    Size: 1347 bytes
    Version: 1
    """
    _fields_ = [
        ('header'           , PacketHeader            ),  # Header
        ('carTelemetryData' , CarTelemetryData_V1 * 20),
        ('buttonStatus'     , ctypes.c_uint32         )   # Bit flags specifying␣
↪which buttons are being
                                                          # pressed currently - see␣
↪appendices
    ]

##############################################################
#                                                            #
#  ___________  Packet ID 7 : CAR STATUS PACKET  ___________  #
#                                                            #
##############################################################

class CarStatusData_V1(PackedLittleEndianStructure):
    """This type is used for the 20-element 'carStatusData' array of the␣
↪PacketCarStatusData_V1 type, defined below.

    There is some data in the Car Status packets that you may not want other players␣
↪seeing if you are in a multiplayer game.
    This is controlled by the "Your Telemetry" setting in the Telemetry options. The␣
↪options are:

        Restricted (Default) - other players viewing the UDP data will not see values␣
↪for your car;
        Public - all other players can see all the data for your car.
```

(continues on next page)

(continued from previous page)

```
    Note: You can always see the data for the car you are driving regardless of the setting.

    The following data items are set to zero if the player driving the car in question has their "Your Telemetry" set to "Restricted":

        fuelInTank
        fuelCapacity
        fuelMix
        fuelRemainingLaps
        frontBrakeBias
        frontLeftWingDamage
        frontRightWingDamage
        rearWingDamage
        engineDamage
        gearBoxDamage
        tyresWear (All four wheels)
        tyresDamage (All four wheels)
        ersDeployMode
        ersStoreEnergy
        ersDeployedThisLap
        ersHarvestedThisLapMGUK
        ersHarvestedThisLapMGUH
    """
    _fields_ = [
        ('tractionControl'         , ctypes.c_uint8    ),  # 0 (off) - 2 (high)
        ('antiLockBrakes'          , ctypes.c_uint8    ),  # 0 (off) - 1 (on)
        ('fuelMix'                 , ctypes.c_uint8    ),  # Fuel mix - 0 = lean, 1 = standard, 2 = rich, 3 = max
        ('frontBrakeBias'          , ctypes.c_uint8    ),  # Front brake bias (percentage)
        ('pitLimiterStatus'        , ctypes.c_uint8    ),  # Pit limiter status - 0 = off, 1 = on
        ('fuelInTank'              , ctypes.c_float    ),  # Current fuel mass
        ('fuelCapacity'            , ctypes.c_float    ),  # Fuel capacity
        ('fuelRemainingLaps'       , ctypes.c_float    ),  # Fuel remaining in terms of laps (value on MFD)
        ('maxRPM'                  , ctypes.c_uint16   ),  # Cars max RPM, point of rev limiter
        ('idleRPM'                 , ctypes.c_uint16   ),  # Cars idle RPM
        ('maxGears'                , ctypes.c_uint8    ),  # Maximum number of gears
        ('drsAllowed'              , ctypes.c_uint8    ),  # 0 = not allowed, 1 = allowed, -1 = unknown
        ('tyresWear'               , ctypes.c_uint8 * 4),  # Tyre wear percentage
        ('actualTyreCompound'      , ctypes.c_uint8    ),  # F1 Modern - 16 = C5, 17 = C4, 18 = C3, 19 = C2, 20 = C1
                                                           # 7 = inter, 8 = wet
                                                           # F1 Classic - 9 = dry, 10 = wet
                                                           # F2 - 11 = super soft, 12 = soft, 13 = medium, 14 = hard
                                                           # 15 = wet
        ('tyreVisualCompound'      , ctypes.c_uint8    ),  # F1 visual (can be different from actual compound)
                                                           # 16 = soft, 17 = medium, 18 = hard, 7 = inter, 8 = wet
```

(continues on next page)

(continued from previous page)

```
                                                                # F1 Classic - same as␣
↪above
                                                                # F2 - same as above
        ('tyresDamage'              , ctypes.c_uint8 * 4),  # Tyre damage (percentage)
        ('frontLeftWingDamage'      , ctypes.c_uint8    ),  # Front left wing damage␣
↪(percentage)
        ('frontRightWingDamage'     , ctypes.c_uint8    ),  # Front right wing damage␣
↪(percentage)
        ('rearWingDamage'           , ctypes.c_uint8    ),  # Rear wing damage␣
↪(percentage)
        ('engineDamage'             , ctypes.c_uint8    ),  # Engine damage␣
↪(percentage)
        ('gearBoxDamage'            , ctypes.c_uint8    ),  # Gear box damage␣
↪(percentage)
        ('vehicleFiaFlags'          , ctypes.c_int8     ),  # -1 = invalid/unknown, 0␣
↪= none, 1 = green
                                                                # 2 = blue, 3 = yellow, 4␣
↪= red
        ('ersStoreEnergy'           , ctypes.c_float    ),  # ERS energy store in␣
↪Joules
        ('ersDeployMode'            , ctypes.c_uint8    ),  # ERS deployment mode, 0 =␣
↪none, 1 = low, 2 = medium
                                                                # 3 = high, 4 = overtake,␣
↪5 = hotlap
        ('ersHarvestedThisLapMGUK' , ctypes.c_float    ),  # ERS energy harvested␣
↪this lap by MGU-K
        ('ersHarvestedThisLapMGUH' , ctypes.c_float    ),  # ERS energy harvested␣
↪this lap by MGU-H
        ('ersDeployedThisLap'       , ctypes.c_float    )   # ERS energy deployed this␣
↪lap
    ]


class PacketCarStatusData_V1(PackedLittleEndianStructure):
    """This packet details car statuses for all the cars in the race.

    It includes values such as the damage readings on the car.

    Frequency: Rate as specified in menus
    Size: 1143 bytes
    Version: 1
    """
    _fields_ = [
        ('header'        , PacketHeader         ),  # Header
        ('carStatusData' , CarStatusData_V1 * 20)
    ]

######################################################################
#                                                                    #
#  Appendices: various value enumerations used in the UDP output     #
#                                                                    #
######################################################################

TeamIDs = {
     0 : 'Mercedes',
     1 : 'Ferrari',
     2 : 'Red Bull Racing',
```

(continues on next page)

(continued from previous page)

```
    3 : 'Williams',
    4 : 'Racing Point',
    5 : 'Renault',
    6 : 'Toro Rosso',
    7 : 'Haas',
    8 : 'McLaren',
    9 : 'Alfa Romeo',
   10 : 'McLaren 1988',
   11 : 'McLaren 1991',
   12 : 'Williams 1992',
   13 : 'Ferrari 1995',
   14 : 'Williams 1996',
   15 : 'McLaren 1998',
   16 : 'Ferrari 2002',
   17 : 'Ferrari 2004',
   18 : 'Renault 2006',
   19 : 'Ferrari 2007',
   21 : 'Red Bull 2010',
   22 : 'Ferrari 1976',
   23 : 'ART Grand Prix',
   24 : 'Campos Vexatec Racing',
   25 : 'Carlin',
   26 : 'Charouz Racing System',
   27 : 'DAMS',
   28 : 'Russian Time',
   29 : 'MP Motorsport',
   30 : 'Pertamina',
   31 : 'McLaren 1990',
   32 : 'Trident',
   33 : 'BWT Arden',
   34 : 'McLaren 1976',
   35 : 'Lotus 1972',
   36 : 'Ferrari 1979',
   37 : 'McLaren 1982',
   38 : 'Williams 2003',
   39 : 'Brawn 2009',
   40 : 'Lotus 1978',
   63 : 'Ferrari 1990',
   64 : 'McLaren 2010',
   65 : 'Ferrari 2010'
}


DriverIDs = {
    0 : 'Carlos Sainz',
    1 : 'Daniil Kvyat',
    2 : 'Daniel Ricciardo',
    6 : 'Kimi Räikkönen',
    7 : 'Lewis Hamilton',
    9 : 'Max Verstappen',
   10 : 'Nico Hulkenberg',
   11 : 'Kevin Magnussen',
   12 : 'Romain Grosjean',
   13 : 'Sebastian Vettel',
   14 : 'Sergio Perez',
   15 : 'Valtteri Bottas',
   19 : 'Lance Stroll',
```

(continues on next page)

(continued from previous page)

```
    20 : 'Arron Barnes',
    21 : 'Martin Giles',
    22 : 'Alex Murray',
    23 : 'Lucas Roth',
    24 : 'Igor Correia',
    25 : 'Sophie Levasseur',
    26 : 'Jonas Schiffer',
    27 : 'Alain Forest',
    28 : 'Jay Letourneau',
    29 : 'Esto Saari',
    30 : 'Yasar Atiyeh',
    31 : 'Callisto Calabresi',
    32 : 'Naota Izum',
    33 : 'Howard Clarke',
    34 : 'Wilhelm Kaufmann',
    35 : 'Marie Laursen',
    36 : 'Flavio Nieves',
    37 : 'Peter Belousov',
    38 : 'Klimek Michalski',
    39 : 'Santiago Moreno',
    40 : 'Benjamin Coppens',
    41 : 'Noah Visser',
    42 : 'Gert Waldmuller',
    43 : 'Julian Quesada',
    44 : 'Daniel Jones',
    45 : 'Artem Markelov',
    46 : 'Tadasuke Makino',
    47 : 'Sean Gelael',
    48 : 'Nyck De Vries',
    49 : 'Jack Aitken',
    50 : 'George Russell',
    51 : 'Maximilian Günther',
    52 : 'Nirei Fukuzumi',
    53 : 'Luca Ghiotto',
    54 : 'Lando Norris',
    55 : 'Sérgio Sette Câmara',
    56 : 'Louis Delétraz',
    57 : 'Antonio Fuoco',
    58 : 'Charles Leclerc',
    59 : 'Pierre Gasly',
    62 : 'Alexander Albon',
    63 : 'Nicholas Latifi',
    64 : 'Dorian Boccolacci',
    65 : 'Niko Kari',
    66 : 'Roberto Merhi',
    67 : 'Arjun Maini',
    68 : 'Alessio Lorandi',
    69 : 'Ruben Meijer',
    70 : 'Rashid Nair',
    71 : 'Jack Tremblay',
    74 : 'Antonio Giovinazzi',
    75 : 'Robert Kubica'
}


TrackIDs = {
     0 : 'Melbourne',
```

(continues on next page)

(continued from previous page)

```
    1 : 'Paul Ricard',
    2 : 'Shanghai',
    3 : 'Sakhir (Bahrain)',
    4 : 'Catalunya',
    5 : 'Monaco',
    6 : 'Montreal',
    7 : 'Silverstone',
    8 : 'Hockenheim',
    9 : 'Hungaroring',
   10 : 'Spa',
   11 : 'Monza',
   12 : 'Singapore',
   13 : 'Suzuka',
   14 : 'Abu Dhabi',
   15 : 'Texas',
   16 : 'Brazil',
   17 : 'Austria',
   18 : 'Sochi',
   19 : 'Mexico',
   20 : 'Baku (Azerbaijan)',
   21 : 'Sakhir Short',
   22 : 'Silverstone Short',
   23 : 'Texas Short',
   24 : 'Suzuka Short'
}


NationalityIDs = {
    1 : 'American',
    2 : 'Argentinian',
    3 : 'Australian',
    4 : 'Austrian',
    5 : 'Azerbaijani',
    6 : 'Bahraini',
    7 : 'Belgian',
    8 : 'Bolivian',
    9 : 'Brazilian',
   10 : 'British',
   11 : 'Bulgarian',
   12 : 'Cameroonian',
   13 : 'Canadian',
   14 : 'Chilean',
   15 : 'Chinese',
   16 : 'Colombian',
   17 : 'Costa Rican',
   18 : 'Croatian',
   19 : 'Cypriot',
   20 : 'Czech',
   21 : 'Danish',
   22 : 'Dutch',
   23 : 'Ecuadorian',
   24 : 'English',
   25 : 'Emirian',
   26 : 'Estonian',
   27 : 'Finnish',
   28 : 'French',
   29 : 'German',
```

(continues on next page)

(continued from previous page)

```
    30 : 'Ghanaian',
    31 : 'Greek',
    32 : 'Guatemalan',
    33 : 'Honduran',
    34 : 'Hong Konger',
    35 : 'Hungarian',
    36 : 'Icelander',
    37 : 'Indian',
    38 : 'Indonesian',
    39 : 'Irish',
    40 : 'Israeli',
    41 : 'Italian',
    42 : 'Jamaican',
    43 : 'Japanese',
    44 : 'Jordanian',
    45 : 'Kuwaiti',
    46 : 'Latvian',
    47 : 'Lebanese',
    48 : 'Lithuanian',
    49 : 'Luxembourger',
    50 : 'Malaysian',
    51 : 'Maltese',
    52 : 'Mexican',
    53 : 'Monegasque',
    54 : 'New Zealander',
    55 : 'Nicaraguan',
    56 : 'North Korean',
    57 : 'Northern Irish',
    58 : 'Norwegian',
    59 : 'Omani',
    60 : 'Pakistani',
    61 : 'Panamanian',
    62 : 'Paraguayan',
    63 : 'Peruvian',
    64 : 'Polish',
    65 : 'Portuguese',
    66 : 'Qatari',
    67 : 'Romanian',
    68 : 'Russian',
    69 : 'Salvadoran',
    70 : 'Saudi',
    71 : 'Scottish',
    72 : 'Serbian',
    73 : 'Singaporean',
    74 : 'Slovakian',
    75 : 'Slovenian',
    76 : 'South Korean',
    77 : 'South African',
    78 : 'Spanish',
    79 : 'Swedish',
    80 : 'Swiss',
    81 : 'Thai',
    82 : 'Turkish',
    83 : 'Uruguayan',
    84 : 'Ukrainian',
    85 : 'Venezuelan',
    86 : 'Welsh'
```

(continues on next page)

(continued from previous page)

```
}


# These surface types are from physics data and show what type of contact each wheel is experiencing.
SurfaceTypes = {
     0 : 'Tarmac',
     1 : 'Rumble strip',
     2 : 'Concrete',
     3 : 'Rock',
     4 : 'Gravel',
     5 : 'Mud',
     6 : 'Sand',
     7 : 'Grass',
     8 : 'Water',
     9 : 'Cobblestone',
    10 : 'Metal',
    11 : 'Ridged'
}


@enum.unique
class ButtonFlag(enum.IntEnum):
    """Bit-mask values for the 'button' field in Car Telemetry Data packets."""
    CROSS              = 0x0001
    TRIANGLE           = 0x0002
    CIRCLE             = 0x0004
    SQUARE             = 0x0008
    D_PAD_LEFT         = 0x0010
    D_PAD_RIGHT        = 0x0020
    D_PAD_UP           = 0x0040
    D_PAD_DOWN         = 0x0080
    OPTIONS            = 0x0100
    L1                 = 0x0200
    R1                 = 0x0400
    L2                 = 0x0800
    R2                 = 0x1000
    LEFT_STICK_CLICK   = 0x2000
    RIGHT_STICK_CLICK  = 0x4000


ButtonFlag.description = {
    ButtonFlag.CROSS              : "Cross or A",
    ButtonFlag.TRIANGLE           : "Triangle or Y",
    ButtonFlag.CIRCLE             : "Circle or B",
    ButtonFlag.SQUARE             : "Square or X",
    ButtonFlag.D_PAD_LEFT         : "D-pad Left",
    ButtonFlag.D_PAD_RIGHT        : "D-pad Right",
    ButtonFlag.D_PAD_UP           : "D-pad Up",
    ButtonFlag.D_PAD_DOWN         : "D-pad Down",
    ButtonFlag.OPTIONS            : "Options or Menu",
    ButtonFlag.L1                 : "L1 or LB",
    ButtonFlag.R1                 : "R1 or RB",
    ButtonFlag.L2                 : "L2 or LT",
    ButtonFlag.R2                 : "R2 or RT",
    ButtonFlag.LEFT_STICK_CLICK   : "Left Stick Click",
    ButtonFlag.RIGHT_STICK_CLICK  : "Right Stick Click"
```

(continues on next page)

(continued from previous page)

```
}

##################################
#                                #
#  Decode UDP telemetry packets  #
#                                #
##################################

# Map from (packetFormat, packetVersion, packetId) to a specific packet type.
HeaderFieldsToPacketType = {
    (2019, 1, 0) : PacketMotionData_V1,
    (2019, 1, 1) : PacketSessionData_V1,
    (2019, 1, 2) : PacketLapData_V1,
    (2019, 1, 3) : PacketEventData_V1,
    (2019, 1, 4) : PacketParticipantsData_V1,
    (2019, 1, 5) : PacketCarSetupData_V1,
    (2019, 1, 6) : PacketCarTelemetryData_V1,
    (2019, 1, 7) : PacketCarStatusData_V1
}

class UnpackError(Exception):
    pass

def unpack_udp_packet(packet: bytes) -> PackedLittleEndianStructure:
    """Convert raw UDP packet to an appropriately-typed telemetry packet.

    Args:
        packet: the contents of the UDP packet to be unpacked.

    Returns:
        The decoded packet structure.

    Raises:
        UnpackError if a problem is detected.
    """
    actual_packet_size = len(packet)

    header_size = ctypes.sizeof(PacketHeader)

    if actual_packet_size < header_size:
        raise UnpackError("Bad telemetry packet: too short ({} bytes).".format(actual_
↪packet_size))

    header = PacketHeader.from_buffer_copy(packet)
    key = (header.packetFormat, header.packetVersion, header.packetId)

    if key not in HeaderFieldsToPacketType:
        raise UnpackError("Bad telemetry packet: no match for key fields {!r}.".
↪format(key))

    packet_type = HeaderFieldsToPacketType[key]

    expected_packet_size = ctypes.sizeof(packet_type)

    if actual_packet_size != expected_packet_size:
        raise UnpackError("Bad telemetry packet: bad size for {} packet; expected {}␣
↪bytes but received {} bytes.".format(
```

(continues on next page)

(continued from previous page)

```
            packet_type.__name__, expected_packet_size, actual_packet_size))

    return packet_type.from_buffer_copy(packet)

##############################################################################
#                                                                            #
#  Verify packet sizes if this module is executed rather than imported       #
#                                                                            #
##############################################################################

if __name__ == "__main__":

    # Check all the packet sizes.

    assert ctypes.sizeof(PacketMotionData_V1)        == 1343
    assert ctypes.sizeof(PacketSessionData_V1)       ==  149
    assert ctypes.sizeof(PacketLapData_V1)           ==  843
    assert ctypes.sizeof(PacketEventData_V1)         ==   32
    assert ctypes.sizeof(PacketParticipantsData_V1)  == 1104
    assert ctypes.sizeof(PacketCarSetupData_V1)      ==  843
    assert ctypes.sizeof(PacketCarTelemetryData_V1)  == 1347
    assert ctypes.sizeof(PacketCarStatusData_V1)     == 1143
```

Module: f1_2019_telemetry.cli.recorder

Module *f1_2019_telemetry.cli.recorder* is a script that implements session data recorder functionality.

The script starts a thread to capture incoming UDP packets, and a thread to write captured UDP packets to an SQLite3 database file.

```
#! /usr/bin/env python3

"""This script captures F1 2019 telemetry packets (sent over UDP) and stores them into SQLite3 database files.

One database file will contain all packets from one session.

From UDP packet to database entry
---------------------------------

The data flow of UDP packets into the database is managed by 2 threads.

PacketReceiver thread:

  (1) The PacketReceiver thread does a select() to wait on incoming packets in the UDP socket.
  (2) When woken up with the notification that a UDP packet is available for reading, it is actually read from the socket.
  (3) The receiver thread calls the recorder_thread.record_packet() method with a TimedPacket containing
      the reception timestamp and the packet just read.
  (4) The recorder_thread.record_packet() method locks its packet queue, inserts the packet there,
      then unlocks the queue. Note that this method is only called from within the receiver thread!
```

(continues on next page)

(continued from previous page)

```
  (5) repeat from (1).

PacketRecorder thread:

  (1) The PacketRecorder thread sleeps for a given period, then wakes up.
  (2) It locks its packet queue, moves the queue's packets to a local variable, empties the packet queue,
      then unlocks the packet queue.
  (3) The packets just moved out of the queue are passed to the 'process_incoming_packets' method.
  (4) The 'process_incoming_packets' method inspects the packet headers, and converts the packet data
      into SessionPacket instances that are suitable for inserting into the database.
      In the process, it collects packets from the same session. After collecting all
      available packets from the same session, it passed them on to the
      'process_incoming_same_session_packets' method.
  (5) The 'process_incoming_same_session_packets' method makes sure that the appropriate SQLite database file
      is opened (i.e., the one with matching sessionUID), then writes the packets into the 'packets' table.

By decoupling the packet capture and database writing in different threads, we minimize the risk of
dropping UDP packets. This risk is real because SQLite3 database commits can take a considerable time.
"""

import argparse
import sys
import time
import socket
import sqlite3
import threading
import logging
import ctypes
import selectors

from collections import namedtuple

from .threading_utils import WaitConsoleThread, Barrier
from ..packets import PacketHeader, PacketID, HeaderFieldsToPacketType, unpack_udp_packet

# The type used by the PacketReceiverThread to represent incoming telemetry packets, with timestamp.
TimestampedPacket = namedtuple('TimestampedPacket', 'timestamp, packet')

# The type used by the PacketRecorderThread to represent incoming telemetry packets for storage in the SQLite3 database.
SessionPacket = namedtuple('SessionPacket', 'timestamp, packetFormat, gameMajorVersion, gameMinorVersion, packetVersion, packetId, sessionUID, sessionTime, frameIdentifier, playerCarIndex, packet')


class PacketRecorder:
    """The PacketRecorder records incoming packets to SQLite3 database files.

```

(continues on next page)

(continued from previous page)

```
    A single SQLite3 file stores packets from a single session.
    Whenever a new session starts, any open file is closed, and a new database file is created.
    """

    # The SQLite3 query that creates the 'packets' table in the database file.
    _create_packets_table_query = """
        CREATE TABLE packets (
            pkt_id            INTEGER  PRIMARY KEY, -- Alias for SQLite3's 'rowid'.
            timestamp         REAL     NOT NULL,    -- The POSIX time right after capturing the telemetry packet.
            packetFormat      INTEGER  NOT NULL,    -- Header field: packet format.
            gameMajorVersion  INTEGER  NOT NULL,    -- Header field: game major version.
            gameMinorVersion  INTEGER  NOT NULL,    -- Header field: game minor version.
            packetVersion     INTEGER  NOT NULL,    -- Header field: packet version.
            packetId          INTEGER  NOT NULL,    -- Header field: packet type ('packetId' is a bit of a misnomer).
            sessionUID        CHAR(16) NOT NULL,    -- Header field: unique session id as hex string.
            sessionTime       REAL     NOT NULL,    -- Header field: session time.
            frameIdentifier   INTEGER  NOT NULL,    -- Header field: frame identifier.
            playerCarIndex    INTEGER  NOT NULL,    -- Header field: player car index.
            packet            BLOB     NOT NULL     -- The packet itself
        );
        """

    # The SQLite3 query that inserts packet data into the 'packets' table of an open database file.
    _insert_packets_query = """
        INSERT INTO packets(
            timestamp,
            packetFormat, gameMajorVersion, gameMinorVersion, packetVersion, packetId, sessionUID,
            sessionTime, frameIdentifier, playerCarIndex,
            packet) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?);
        """

    def __init__(self):
        self._conn = None
        self._cursor = None
        self._filename = None
        self._sessionUID = None

    def close(self):
        """Make sure that no database remains open."""
        if self._conn is not None:
            self._close_database()

    def _open_database(self, sessionUID: str):
        """Open SQLite3 database file and make sure it has the correct schema."""
        assert self._conn is None
        filename = "F1_2019_{:s}.sqlite3".format(sessionUID)
        logging.info("Opening file {!r}.".format(filename))
        conn = sqlite3.connect(filename)
        cursor = conn.cursor()
```

(continues on next page)

(continued from previous page)

```
114
115         # Get rid of indentation and superfluous newlines in the 'CREATE TABLE'␣
    ↪command.
116         query = "".join(line[8:] + "\n" for line in PacketRecorder._create_packets_
    ↪table_query.split("\n")[1:-1])
117
118         # Try to execute the 'CREATE TABLE' statement. If it already exists, this␣
    ↪will raise an exception.
119         try:
120             cursor.execute(query)
121         except sqlite3.OperationalError:
122             logging.info("    (Appending to existing file.)")
123         else:
124             logging.info("    (Created new file.)")
125
126         self._conn = conn
127         self._cursor = cursor
128         self._filename = filename
129         self._sessionUID = sessionUID
130
131     def _close_database(self):
132         """Close SQLite3 database file."""
133         assert self._conn is not None
134         logging.info("Closing file {!r}.".format(self._filename))
135         self._cursor.close()
136         self._cursor = None
137         self._conn.close()
138         self._conn = None
139         self._filename = None
140         self._sessionUID = None
141
142     def _insert_and_commit_same_session_packets(self, same_session_packets):
143         """Insert session packets to database and commit."""
144         assert self._conn is not None
145         self._cursor.executemany(PacketRecorder._insert_packets_query, same_session_
    ↪packets)
146         self._conn.commit()
147
148     def _process_same_session_packets(self, same_session_packets):
149         """Insert packets from the same session into the 'packets' table of the␣
    ↪appropriate database file.
150
151         Precondition: all packets in 'same_session_packets' are from the same session␣
    ↪(identical 'sessionUID' field).
152
153         We need to handle four different cases:
154
155         (1) 'same_session_packets' is empty:
156
157             --> return (no-op).
158
159         (2) A database file is currently open, but it stores packets with a different␣
    ↪session UID:
160
161             --> Close database;
162             --> Open database with correct session UID;
163             --> Insert 'same_session_packets'.
```

(continues on next page)

(continued from previous page)

```
164
165         (3) No database file is currently open:
166
167             --> Open database with correct session UID;
168             --> Insert 'same_session_packets'.
169
170         (4) A database is currently open, with correct session UID:
171
172             --> Insert 'same_session_packets'.
173         """
174
175         if not same_session_packets:
176             # Nothing to insert.
177             return
178
179         if self._conn is not None and self._sessionUID != same_session_packets[0].
    ↪sessionUID:
180             # Close database if it's recording a different session.
181             self._close_database()
182
183         if self._conn is None:
184             # Open database with the correct sessionID.
185             self._open_database(same_session_packets[0].sessionUID)
186
187         # Write packets.
188         self._insert_and_commit_same_session_packets(same_session_packets)
189
190     def process_incoming_packets(self, timestamped_packets):
191         """Process incoming packets by recording them into the correct database file.
192
193         The incoming 'timestamped_packets' is a list of timestamped raw UDP packets.
194
195         We process them to a variable 'same_session_packets', which is a list of␣
    ↪consecutive
196         packets having the same 'sessionUID' field. In this list, each packet is a 11-
    ↪element tuple
197         that can be inserted into the 'packets' table of the database.
198
199         The 'same_session_packets' are then passed on to the '_process_same_session_
    ↪packets'
200         method that writes them into the appropriate database file.
201         """
202
203         t1 = time.monotonic()
204
205         # Invariant to be guaranteed: all packets in 'same_session_packets' have the␣
    ↪same 'sessionUID' field.
206         same_session_packets = []
207
208         for (timestamp, packet) in timestamped_packets:
209
210             if len(packet) < ctypes.sizeof(PacketHeader):
211                 logging.error("Dropped bad packet of size {} (too short).".
    ↪format(len(packet)))
212                 continue
213
214             header = PacketHeader.from_buffer_copy(packet)
```

(continues on next page)

(continued from previous page)

```
215
216             packet_type_tuple = (header.packetFormat, header.packetVersion, header.
↪packetId)
217
218             packet_type = HeaderFieldsToPacketType.get(packet_type_tuple)
219             if packet_type is None:
220                 logging.error("Dropped unrecognized packet (format, version, id) = {!
↪r}.".format(packet_type_tuple))
221                 continue
222
223             if len(packet) != ctypes.sizeof(packet_type):
224                 logging.error("Dropped packet with unexpected size; "
225                               "(format, version, id) = {!r} packet, size = {},␣
↪expected {}.".format(
226                                   packet_type_tuple, len(packet), ctypes.
↪sizeof(packet_type)))
227                 continue
228
229             if header.packetId == PacketID.EVENT:  # Log Event packets
230                 event_packet = unpack_udp_packet(packet)
231                 logging.info("Recording event packet: {}".format(event_packet.
↪eventStringCode.decode()))
232
233             # NOTE: the sessionUID is not reliable at the start of a session (in F1␣
↪2018, need to check for F1 2019).
234             # See: http://forums.codemasters.com/discussion/138130/bug-f1-2018-pc-v1-
↪0-4-udp-telemetry-bad-session-uid-in-first-few-packets-of-a-session
235
236             # Create an INSERT-able tuple for the data in this packet.
237             #
238             # Note that we convert the sessionUID to a 16-digit hex string here.
239             # SQLite3 can store 64-bit numbers, but only signed ones.
240             # To prevent any issues, we represent the sessionUID as a 16-digit hex␣
↪string instead.
241
242             session_packet = SessionPacket(
243                 timestamp,
244                 header.packetFormat, header.gameMajorVersion, header.gameMinorVersion,
245                 header.packetVersion, header.packetId, "{:016x}".format(header.
↪sessionUID),
246                 header.sessionTime, header.frameIdentifier, header.playerCarIndex,
247                 packet
248             )
249
250             if len(same_session_packets) > 0 and same_session_packets[0].sessionUID !
↪= session_packet.sessionUID:
251                 # Write 'same_session_packets' collected so far to the correct␣
↪session database, then forget about them.
252                 self._process_same_session_packets(same_session_packets)
253                 same_session_packets.clear()
254
255             same_session_packets.append(session_packet)
256
257         # Write 'same_session_packets' to the correct session database, then forget␣
↪about them.
258         # The 'same_session_packets.clear()' is not strictly necessary here, because
↪'same_session_packets' is about to
```

(continues on next page)

(continued from previous page)

```
        #   go out of scope; but we make it explicit for clarity.

        self._process_same_session_packets(same_session_packets)
        same_session_packets.clear()

        t2 = time.monotonic()

        duration = (t2 - t1)

        logging.info("Recorded {} packets in {:.3f} ms.".format(len(timestamped_
↪packets), duration * 1000.0))

    def no_packets_received(self, age: float) -> None:
        """No packets were received for a considerable time. If a database file is␣
↪open, close it."""
        if self._conn is None:
            logging.info("No packets to record for {:.3f} seconds.".format(age))
        else:
            logging.info("No packets to record for {:.3f} seconds; closing file due␣
↪to inactivity.".format(age))
            self._close_database()


class PacketRecorderThread(threading.Thread):
    """The PacketRecorderThread writes telemetry data to SQLite3 files."""

    def __init__(self, record_interval):
        super().__init__(name='recorder')
        self._record_interval = record_interval
        self._packets = []
        self._packets_lock = threading.Lock()
        self._socketpair = socket.socketpair()

    def close(self):
        for sock in self._socketpair:
            sock.close()

    def run(self):
        """Receive incoming packets and hand them over the the PacketRecorder.

        This method runs in its own thread.
        """

        selector = selectors.DefaultSelector()
        key_socketpair = selector.register(self._socketpair[0], selectors.EVENT_READ)

        recorder = PacketRecorder()

        packets = []

        logging.info("Recorder thread started.")

        quitflag = False
        inactivity_timer = time.time()
        while not quitflag:

            # Calculate the timeout value that will bring us in sync with the next␣
↪period.
```

(continues on next page)

(continued from previous page)

```
            timeout = (-time.time()) % self._record_interval
            # If the timeout interval is too short, increase its length by 1 period.
            if timeout < 0.5 * self._record_interval:
                timeout += self._record_interval

            for (key, events) in selector.select(timeout):
                if key == key_socketpair:
                    quitflag = True

            # Swap packets, so the 'record_packet' method can be called uninhibited as soon as possible.
            with self._packets_lock:
                (packets, self._packets) = (self._packets, packets)

            if len(packets) != 0:
                inactivity_timer = packets[-1].timestamp
                recorder.process_incoming_packets(packets)
                packets.clear()
            else:
                t_now = time.time()
                age = t_now - inactivity_timer
                recorder.no_packets_received(age)
                inactivity_timer = t_now

        recorder.close()

        selector.close()

        logging.info("Recorder thread stopped.")

    def request_quit(self):
        """Request termination of the PacketRecorderThread.

        Called from the main thread to request that we quit.
        """
        self._socketpair[1].send(b'\x00')

    def record_packet(self, timestamped_packet):
        """Called from the receiver thread for every UDP packet received."""
        with self._packets_lock:
            self._packets.append(timestamped_packet)


class PacketReceiverThread(threading.Thread):
    """The PacketReceiverThread receives incoming telemetry packets via the network and passes them to the PacketRecorderThread for storage."""

    def __init__(self, udp_port, recorder_thread):
        super().__init__(name='receiver')
        self._udp_port = udp_port
        self._recorder_thread = recorder_thread
        self._socketpair = socket.socketpair()

    def close(self):
        for sock in self._socketpair:
            sock.close()

```

(continues on next page)

(continued from previous page)

```
    def run(self):
        """Receive incoming packets and hand them over to the PacketRecorderThread.

        This method runs in its own thread.
        """

        udp_socket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)

        # Allow multiple receiving endpoints.
        if sys.platform in ['darwin']:
            udp_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)
        elif sys.platform in ['linux', 'win32']:
            udp_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

        # Accept UDP packets from any host.
        address = ('', self._udp_port)
        udp_socket.bind(address)

        selector = selectors.DefaultSelector()

        key_udp_socket = selector.register(udp_socket, selectors.EVENT_READ)
        key_socketpair = selector.register(self._socketpair[0], selectors.EVENT_READ)

        logging.info("Receiver thread started, reading UDP packets from port {}.".
↪format(self._udp_port))

        quitflag = False
        while not quitflag:
            for (key, events) in selector.select():
                timestamp = time.time()
                if key == key_udp_socket:
                    # All telemetry UDP packets fit in 2048 bytes with room to spare.
                    packet = udp_socket.recv(2048)
                    timestamped_packet = TimestampedPacket(timestamp, packet)
                    self._recorder_thread.record_packet(timestamped_packet)
                elif key == key_socketpair:
                    quitflag = True

        selector.close()
        udp_socket.close()
        for sock in self._socketpair:
            sock.close()

        logging.info("Receiver thread stopped.")

    def request_quit(self):
        """Request termination of the PacketReceiverThread.

        Called from the main thread to request that we quit.
        """
        self._socketpair[1].send(b'\x00')


def main():
    """Record incoming telemetry data until the user presses enter."""

    # Configure logging.
```

(continues on next page)

(continued from previous page)

```
424
425     logging.basicConfig(level=logging.DEBUG, format="%(asctime)-23s | %(threadName)-
↪10s | %(levelname)-5s | %(message)s")
426     logging.Formatter.default_msec_format = '%s.%03d'
427
428     # Parse command line arguments.
429
430     parser = argparse.ArgumentParser(description="Record F1 2019 telemetry data to␣
↪SQLite3 files.")
431
432     parser.add_argument("-p", "--port", default=20777, type=int, help="UDP port to␣
↪listen to (default: 20777)", dest='port')
433     parser.add_argument("-i", "--interval", default=1.0, type=float, help="interval␣
↪for writing incoming data to SQLite3 file, in seconds (default: 1.0)", dest=
↪'interval')
434
435     args = parser.parse_args()
436
437     # Start recorder thread first, then receiver thread.
438
439     quit_barrier = Barrier()
440
441     recorder_thread = PacketRecorderThread(args.interval)
442     recorder_thread.start()
443
444     receiver_thread = PacketReceiverThread(args.port, recorder_thread)
445     receiver_thread.start()
446
447     wait_console_thread = WaitConsoleThread(quit_barrier)
448     wait_console_thread.start()
449
450     # Recorder, receiver, and wait_console threads are now active. Run until we're␣
↪asked to quit.
451
452     quit_barrier.wait()
453
454     # Stop threads.
455
456     wait_console_thread.request_quit()
457     wait_console_thread.join()
458     wait_console_thread.close()
459
460     receiver_thread.request_quit()
461     receiver_thread.join()
462     receiver_thread.close()
463
464     recorder_thread.request_quit()
465     recorder_thread.join()
466     recorder_thread.close()
467
468     # All done.
469
470     logging.info("All done.")
471
472
473 if __name__ == "__main__":
474     main()
```

Module: f1_2019_telemetry.cli.player

Module *f1_2019_telemetry.cli.player* is a script that implements session data playback functionality.

The script starts a thread to read session data packets stored in a SQLite3 database file, and plays them back as UDP network packets. The speed at which playback happens can be changed by a command-line parameter.

```
#! /usr/bin/env python3

"""This script reads F1 2019 telemetry packets stored in a SQLite3 database file and sends them out over UDP, effectively replaying a session of the F1 2019 game."""

import sys
import logging
import threading
import argparse
import time
import sqlite3
import socket
import selectors

from .threading_utils import WaitConsoleThread, Barrier
from ..packets import HeaderFieldsToPacketType


class PacketPlaybackThread(threading.Thread):
    """The PacketPlaybackThread reads telemetry data from an SQLite3 file and plays it back as UDP packets."""

    def __init__(self, filename, destination, port, realtime_factor, quit_barrier):
        super().__init__(name='playback')
        self._filename = filename
        self._destination = destination
        self._port = port
        self._realtime_factor = realtime_factor
        self._quit_barrier = quit_barrier

        self._packets = []
        self._packets_lock = threading.Lock()
        self._socketpair = socket.socketpair()

    def close(self):
        for sock in self._socketpair:
            sock.close()

    def run(self):
        """Read packets from database and replay them as UDP packets.

        The run method executes in its own thread.
        """
        selector = selectors.DefaultSelector()
        key_socketpair = selector.register(self._socketpair[0], selectors.EVENT_READ)

        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        #sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

        if self._destination is None:
            sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
```

(continues on next page)

(continued from previous page)

```
            sock.connect(('<broadcast>', self._port))
        else:
            sock.connect((self._destination, self._port))

        conn = sqlite3.connect(self._filename)
        cursor = conn.cursor()

        query = "SELECT timestamp, packet FROM packets ORDER BY pkt_id;"

        cursor.execute(query)

        logging.info("Playback thread started.")

        packet_count = 0
        quitflag = False

        t_first_packet = None
        t_start_playback = time.monotonic()
        while not quitflag:
            timestamped_packet = cursor.fetchone()
            if timestamped_packet is None:
                quitflag = True
                continue

            (timestamp, packet) = timestamped_packet
            if t_first_packet is None:
                t_first_packet = timestamp
            t_playback = t_start_playback + (timestamp - t_first_packet) / self._
↪realtime_factor

            while True:
                t_sleep = max(0.0, t_playback - time.monotonic())
                for (key, events) in selector.select(t_sleep):
                    if key == key_socketpair:
                        quitflag = True

                if quitflag:
                    break

                delay = time.monotonic() - t_playback

                if delay >= 0:
                    sock.send(packet)
                    packet_count += 1
                    if packet_count % 500 == 0:
                        logging.info("{} packages sent, delay: {:.3f} ms".
↪format(packet_count, 1000.0 * delay))
                    break


        cursor.close()
        conn.close()

        sock.close()

        self._quit_barrier.proceed()

```

(continues on next page)

(continued from previous page)

```
        logging.info("playback thread stopped.")

    def request_quit(self):
        """Called from the main thread to request that we quit."""
        self._socketpair[1].send(b'\x00')


def main():

    # Configure logging.

    logging.basicConfig(level=logging.DEBUG, format="%(asctime)-23s | %(threadName)-10s | %(levelname)-5s | %(message)s")
    logging.Formatter.default_msec_format = '%s.%03d'

    # Parse command line arguments.

    parser = argparse.ArgumentParser(description="Replay an F1 2019 session as UDP packets.")

    parser.add_argument("-r", "--rtf", dest='realtime_factor', type=float, default=1.0, help="playback real-time factor (higher is faster, default=1.0)")
    parser.add_argument("-d", "--destination", type=str, default=None, help="destination UDP address; omit to use broadcast (default)")
    parser.add_argument("-p", "--port", type=int, default=20777, help="destination UDP port (default: 20777)")
    parser.add_argument("filename", type=str, help="SQLite3 file to replay packets from")

    args = parser.parse_args()

    # Start threads.

    quit_barrier = Barrier()

    playback_thread = PacketPlaybackThread(args.filename, args.destination, args.port, args.realtime_factor, quit_barrier)
    playback_thread.start()

    wait_console_thread = WaitConsoleThread(quit_barrier)
    wait_console_thread.start()

    # Playback and wait_console threads are now active. Run until we're asked to quit.

    quit_barrier.wait()

    # Stop threads.

    wait_console_thread.request_quit()
    wait_console_thread.join()
    wait_console_thread.close()

    playback_thread.request_quit()
    playback_thread.join()
    playback_thread.close()

    # All done.
```

(continues on next page)

(continued from previous page)

```
    logging.info("All done.")


if __name__ == "__main__":
    main()
```

Module: f1_2019_telemetry.cli.monitor

Module *f1_2019_telemetry.cli.monitor* is a script that prints live session data.

The script starts a thread to capture incoming UDP packets, and outputs a summary of incoming packets.

```
#! /usr/bin/env python3

"""This script monitors a UDP port for F1 2019 telemetry packets and prints useful info upon reception."""

import argparse
import sys
import socket
import threading
import logging
import selectors
import math

from .threading_utils import WaitConsoleThread, Barrier
from ..packets import PacketID, unpack_udp_packet


class PacketMonitorThread(threading.Thread):
    """The PacketMonitorThread receives incoming telemetry packets via the network and shows interesting information."""

    def __init__(self, udp_port):
        super().__init__(name='monitor')
        self._udp_port = udp_port
        self._socketpair = socket.socketpair()

        self._current_frame = None
        self._current_frame_data = {}

    def close(self):
        for sock in self._socketpair:
            sock.close()

    def run(self):
        """Receive incoming packets and print info about them.

        This method runs in its own thread.
        """

        udp_socket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)

        # Allow multiple receiving endpoints.
```

(continues on next page)

(continued from previous page)

```
        if sys.platform in ['darwin']:
            udp_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)
        elif sys.platform in ['linux', 'win32']:
            udp_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

        # Accept UDP packets from any host.
        address = ('', self._udp_port)
        udp_socket.bind(address)

        selector = selectors.DefaultSelector()

        key_udp_socket = selector.register(udp_socket, selectors.EVENT_READ)
        key_socketpair = selector.register(self._socketpair[0], selectors.EVENT_READ)

        logging.info("Monitor thread started, reading UDP packets from port {}.".
↪format(self._udp_port))

        quitflag = False
        while not quitflag:
            for (key, events) in selector.select():
                if key == key_udp_socket:
                    # All telemetry UDP packets fit in 2048 bytes with room to spare.
                    udp_packet = udp_socket.recv(2048)
                    packet = unpack_udp_packet(udp_packet)
                    self.process(packet)
                elif key == key_socketpair:
                    quitflag = True

        self.report()

        selector.close()
        udp_socket.close()
        for sock in self._socketpair:
            sock.close()

        logging.info("Monitor thread stopped.")

    def process(self, packet):

        if packet.header.frameIdentifier != self._current_frame:
            self.report()
            self._current_frame = packet.header.frameIdentifier
            self._current_frame_data = {}

        self._current_frame_data[PacketID(packet.header.packetId)] = packet


    def report(self):
        if self._current_frame is None:
            return

        any_packet = next(iter(self._current_frame_data.values()))

        player_car = any_packet.header.playerCarIndex

        try:
            distance = self._current_frame_data[PacketID.LAP_DATA].lapData[player_
↪car].totalDistance
```

(continues on next page)

(continued from previous page)

```
97          except:
98              distance = math.nan
99
100         message = "frame {:6d} distance {:10.3f}".format(self._current_frame,␣
    ↪distance)
101
102         if message is not None:
103             logging.info(message)
104
105     def request_quit(self):
106         """Request termination of the PacketMonitorThread.
107
108         Called from the main thread to request that we quit.
109         """
110         self._socketpair[1].send(b'\x00')
111
112
113 def main():
114     """Record incoming telemetry data until the user presses enter."""
115
116     # Configure logging.
117
118     logging.basicConfig(level=logging.DEBUG, format="%(asctime)-23s | %(threadName)-
    ↪10s | %(levelname)-5s | %(message)s")
119     logging.Formatter.default_msec_format = '%s.%03d'
120
121     # Parse command line arguments.
122
123     parser = argparse.ArgumentParser(description="Monitor UDP port for incoming F1␣
    ↪2019 telemetry data and print information.")
124
125     parser.add_argument("-p", "--port", default=20777, type=int, help="UDP port to␣
    ↪listen to (default: 20777)", dest='port')
126
127     args = parser.parse_args()
128
129     # Start recorder thread first, then receiver thread.
130
131     quit_barrier = Barrier()
132
133     monitor_thread = PacketMonitorThread(args.port)
134     monitor_thread.start()
135
136     wait_console_thread = WaitConsoleThread(quit_barrier)
137     wait_console_thread.start()
138
139     # Monitor and wait_console threads are now active. Run until we're asked to quit.
140
141     quit_barrier.wait()
142
143     # Stop threads.
144
145     wait_console_thread.request_quit()
146     wait_console_thread.join()
147     wait_console_thread.close()
148
149     monitor_thread.request_quit()
```

(continues on next page)

(continued from previous page)

```
150     monitor_thread.join()
151     monitor_thread.close()
152
153     # All done.
154
155     logging.info("All done.")
156
157
158 if __name__ == "__main__":
159     main()
```

2.2 F1 2019 Telemetry Packet Specification

Note: This specification was copied (with the minor changes listed below) from the CodeMasters forum topic describing the F1 2019 telemetry UDP packet specification, as found here:

https://forums.codemasters.com/topic/38920-f1-2019-udp-specification/

The forum post has one post detailing packet formats, followed by a post with Frequently Asked Questions, followed by a post with appendices, giving a number of lookup tables. The package format and appendices have been reproduced here; for the FAQ, please refer to the original forum topic.

The following changes were made in the process of copying the specification:

- Added suffix '_t' to all integer types, bringing the type names in lines with the types declared in the standard C header file `<stdint.h>` (equivalent to `<cstdint>` in C++). This change also improves the syntax highlighting of the struct definitions below.
- Added the *uint32_t* type to the *Packet Types* table;
- Changed the type of field *m_frameIdentifier* in the *PacketHeader* struct from *uint* to *uint32_t*;
- In struct *PacketMotionData*: corrected comments of the fields *m_angularAccelerationX*, *m_angularAccelerationY*, and *m_angularAccelerationZ* to reflect that the values represent accelerations rather than velocities;
- In struct *CarSetupData*: corrected comment of field *m_rearAntiRollBar* to refer to *rear* instead of *front*;
- In the Driver IDs appendix: corrected the name of driver 34: *Wilheim Kaufmann* to *Wilhelm Kaufmann*.

The F1 series of games support the output of certain game data across UDP connections. This data can be used supply race information to external applications, or to drive certain hardware (e.g. motion platforms, force feedback steering wheels and LED devices).

The following information summarise this data structures so that developers of supporting hardware or software are able to configure these to work correctly with the F1 game.

If you cannot find the information that you require then please contact community@codemasters.com and a member of the dev team will respond to your query as soon as possible.

2.2.1 Packet Information

Note: The structure definitions given below are specified in the syntax of the C programming language.

The Python versions of the structures provided by the *f1-telemetry-packet* package are very similar to the C versions, with the notable exception that for all field names, the 'm_' prefix is omitted. For example, the header field *m_packetFormat* is just called *packetFormat* in the Python version.

Packet Types

Each packet can now carry different types of data rather than having one packet which contains everything. A header has been added to each packet as well so that versioning can be tracked and it will be easier for applications to check they are interpreting the incoming data in the correct way. Please note that all values are encoded using Little Endian format. All data is packed.

The following data types are used in the structures:

Type	Description
uint8_t	Unsigned 8-bit integer
int8_t	Signed 8-bit integer
uint16_t	Unsigned 16-bit integer
int16_t	Signed 16-bit integer
uint32_t	Unsigned 32-bit integer
float	Floating point (32-bit)
uint64_t	Unsigned 64-bit integer

Packet Header

Each packet has the following header:

```
struct PacketHeader
{
    uint16_t  m_packetFormat;          // 2019
    uint8_t   m_gameMajorVersion;      // Game major version - "X.00"
    uint8_t   m_gameMinorVersion;      // Game minor version - "1.XX"
    uint8_t   m_packetVersion;         // Version of this packet type, all start from 1
    uint8_t   m_packetId;              // Identifier for the packet type, see below
    uint64_t  m_sessionUID;            // Unique identifier for the session
    float     m_sessionTime;           // Session timestamp
    uint32_t  m_frameIdentifier;       // Identifier for the frame the data was␣
↪retrieved on
    uint8_t   m_playerCarIndex;        // Index of player's car in the array
};
```

Packet IDs

The packets IDs are as follows:

Packet Name	Value	Description
Motion	0	Contains all motion data for player's car – only sent while player is in control
Session	1	Data about the session – track, time left
Lap Data	2	Data about all the lap times of cars in the session
Event	3	Various notable events that happen during a session
Participants	4	List of participants in the session, mostly relevant for multiplayer
Car Setups	5	Packet detailing car setups for cars in the race
Car Telemetry	6	Telemetry data for all cars
Car Status	7	Status data for all cars such as damage

Motion Packet

The motion packet gives physics data for all the cars being driven. There is additional data for the car being driven with the goal of being able to drive a motion platform setup.

N.B. For the normalised vectors below, to convert to float values divide by 32767.0f – 16-bit signed values are used to pack the data and on the assumption that direction values are always between -1.0f and 1.0f.

Frequency: Rate as specified in menus
Size: 1343 bytes
Version: 1

```
struct CarMotionData
{
    float         m_worldPositionX;           // World space X position
    float         m_worldPositionY;           // World space Y position
    float         m_worldPositionZ;           // World space Z position
    float         m_worldVelocityX;           // Velocity in world space X
    float         m_worldVelocityY;           // Velocity in world space Y
    float         m_worldVelocityZ;           // Velocity in world space Z
    int16_t       m_worldForwardDirX;         // World space forward X direction␣
↪(normalised)
    int16_t       m_worldForwardDirY;         // World space forward Y direction␣
↪(normalised)
    int16_t       m_worldForwardDirZ;         // World space forward Z direction␣
↪(normalised)
    int16_t       m_worldRightDirX;           // World space right X direction␣
↪(normalised)
    int16_t       m_worldRightDirY;           // World space right Y direction␣
↪(normalised)
    int16_t       m_worldRightDirZ;           // World space right Z direction␣
↪(normalised)
    float         m_gForceLateral;            // Lateral G-Force component
    float         m_gForceLongitudinal;       // Longitudinal G-Force component
    float         m_gForceVertical;           // Vertical G-Force component
    float         m_yaw;                      // Yaw angle in radians
    float         m_pitch;                    // Pitch angle in radians
    float         m_roll;                     // Roll angle in radians
};

struct PacketMotionData
```

(continues on next page)

(continued from previous page)

```
{
    PacketHeader    m_header;                 // Header

    CarMotionData   m_carMotionData[20];      // Data for all cars on track

    // Extra player car ONLY data
    float         m_suspensionPosition[4];       // Note: All wheel arrays have the␣
↪following order:
    float         m_suspensionVelocity[4];       // RL, RR, FL, FR
    float         m_suspensionAcceleration[4];   // RL, RR, FL, FR
    float         m_wheelSpeed[4];               // Speed of each wheel
    float         m_wheelSlip[4];                // Slip ratio for each wheel
    float         m_localVelocityX;              // Velocity in local space
    float         m_localVelocityY;              // Velocity in local space
    float         m_localVelocityZ;              // Velocity in local space
    float         m_angularVelocityX;            // Angular velocity x-component
    float         m_angularVelocityY;            // Angular velocity y-component
    float         m_angularVelocityZ;            // Angular velocity z-component
    float         m_angularAccelerationX;        // Angular acceleration x-component
    float         m_angularAccelerationY;        // Angular acceleration y-component
    float         m_angularAccelerationZ;        // Angular acceleration z-component
    float         m_frontWheelsAngle;            // Current front wheels angle in␣
↪radians
};
```

Session Packet

The session packet includes details about the current session in progress.

Frequency: 2 per second
Size: 149 bytes
Version: 1

```
struct MarshalZone
{
    float  m_zoneStart;   // Fraction (0..1) of way through the lap the marshal zone␣
↪starts
    int8   m_zoneFlag;    // -1 = invalid/unknown, 0 = none, 1 = green, 2 = blue, 3 =␣
↪yellow, 4 = red
};

struct PacketSessionData
{
    PacketHeader    m_header;                 // Header

    uint8_t         m_weather;                // Weather - 0 = clear, 1 = light cloud,␣
↪2 = overcast
                                              // 3 = light rain, 4 = heavy rain, 5 =␣
↪storm
    int8_t          m_trackTemperature;       // Track temp. in degrees celsius
    int8_t          m_airTemperature;         // Air temp. in degrees celsius
    uint8_t         m_totalLaps;              // Total number of laps in this race
```

(continues on next page)

(continued from previous page)

```
    uint16_t        m_trackLength;          // Track length in metres
    uint8_t         m_sessionType;          // 0 = unknown, 1 = P1, 2 = P2, 3 = P3,␣
↪4 = Short P
                                            // 5 = Q1, 6 = Q2, 7 = Q3, 8 = Short Q,␣
↪9 = OSQ
                                            // 10 = R, 11 = R2, 12 = Time Trial
    int8_t          m_trackId;              // -1 for unknown, 0-21 for tracks, see␣
↪appendix
    uint8_t         m_formula;              // Formula, 0 = F1 Modern, 1 = F1␣
↪Classic, 2 = F2,
                                            // 3 = F1 Generic
    uint16_t        m_sessionTimeLeft;      // Time left in session in seconds
    uint16_t        m_sessionDuration;      // Session duration in seconds
    uint8_t         m_pitSpeedLimit;        // Pit speed limit in kilometres per hour
    uint8_t         m_gamePaused;           // Whether the game is paused
    uint8_t         m_isSpectating;         // Whether the player is spectating
    uint8_t         m_spectatorCarIndex;    // Index of the car being spectated
    uint8_t         m_sliProNativeSupport;  // SLI Pro support, 0 = inactive, 1 =␣
↪active
    uint8_t         m_numMarshalZones;      // Number of marshal zones to follow
    MarshalZone     m_marshalZones[21];     // List of marshal zones - max 21
    uint8_t         m_safetyCarStatus;      // 0 = no safety car, 1 = full safety car
                                            // 2 = virtual safety car
    uint8_t         m_networkGame;          // 0 = offline, 1 = online
};
```

Lap Data Packet

The lap data packet gives details of all the cars in the session.

Frequency: Rate as specified in menus
Size: 843 bytes
Version: 1

```
struct LapData
{
    float       m_lastLapTime;              // Last lap time in seconds
    float       m_currentLapTime;           // Current time around the lap in seconds
    float       m_bestLapTime;              // Best lap time of the session in␣
↪seconds
    float       m_sector1Time;              // Sector 1 time in seconds
    float       m_sector2Time;              // Sector 2 time in seconds
    float       m_lapDistance;              // Distance vehicle is around current␣
↪lap in metres - could
                                            // be negative if line hasn't been␣
↪crossed yet
    float       m_totalDistance;            // Total distance travelled in session␣
↪in metres - could
                                            // be negative if line hasn't been␣
↪crossed yet
    float       m_safetyCarDelta;           // Delta in seconds for safety car
    uint8_t     m_carPosition;              // Car race position
```

(continues on next page)

(continued from previous page)

```
    uint8_t      m_currentLapNum;               // Current lap number
    uint8_t      m_pitStatus;                   // 0 = none, 1 = pitting, 2 = in pit area
    uint8_t      m_sector;                      // 0 = sector1, 1 = sector2, 2 = sector3
    uint8_t      m_currentLapInvalid;           // Current lap invalid - 0 = valid, 1 =␣
↪invalid
    uint8_t      m_penalties;                   // Accumulated time penalties in seconds␣
↪to be added
    uint8_t      m_gridPosition;                // Grid position the vehicle started the␣
↪race in
    uint8_t      m_driverStatus;                // Status of driver - 0 = in garage, 1 =␣
↪flying lap
                                                // 2 = in lap, 3 = out lap, 4 = on track
    uint8_t      m_resultStatus;                // Result status - 0 = invalid, 1 =␣
↪inactive, 2 = active
                                                // 3 = finished, 4 = disqualified, 5 =␣
↪not classified
                                                // 6 = retired
};

struct PacketLapData
{
    PacketHeader    m_header;               // Header

    LapData         m_lapData[20];          // Lap data for all cars on track
};
```

Event Packet

This packet gives details of events that happen during the course of a session.

Frequency: When the event occurs
Size: 32 bytes
Version: 1

```
// The event details packet is different for each type of event.
// Make sure only the correct type is interpreted.

union EventDataDetails
{
    struct
    {
        uint8_t      vehicleIdx; // Vehicle index of car achieving fastest lap
        float        lapTime;    // Lap time is in seconds
    } FastestLap;

    struct
    {
        uint8_t vehicleIdx; // Vehicle index of car retiring
    } Retirement;

    struct
    {
```

(continues on next page)

(continued from previous page)

```
        uint8_t vehicleIdx; // Vehicle index of team mate
    } TeamMateInPits;

    struct
    {
        uint8_t vehicleIdx; // Vehicle index of the race winner
    } RaceWinner;
};

struct PacketEventData
{
    PacketHeader     m_header;               // Header

    uint8_t          m_eventStringCode[4];   // Event string code, see below
    EventDataDetails m_eventDetails;         // Event details - should be interpreted␣
↪differently
                                             // for each type
};
```

Event String Codes

Event	Code	Description
Session Started	“SSTA”	Sent when the session starts
Session Ended	“SEND”	Sent when the session ends
Fastest Lap	“FTLP”	When a driver achieves the fastest lap
Retirement	“RTMT”	When a driver retires
DRS enabled	“DRSE”	Race control have enabled DRS
DRS disabled	“DRSD”	Race control have disabled DRS
Team mate in pits	“TMPT”	Your team mate has entered the pits
Chequered flag	“CHQF”	The chequered flag has been waved
Race Winner	“RCWN”	The race winner is announced

Participants Packet

This is a list of participants in the race. If the vehicle is controlled by AI, then the name will be the driver name. If this is a multiplayer game, the names will be the Steam Id on PC, or the LAN name if appropriate.

N.B. on Xbox One, the names will always be the driver name, on PS4 the name will be the LAN name if playing a LAN game, otherwise it will be the driver name.

The array should be indexed by vehicle index.

Frequency: Every 5 seconds
Size: 1104 bytes
Version: 1

```
struct ParticipantData
{
```

(continues on next page)

(continued from previous page)

```
    uint8_t     m_aiControlled;           // Whether the vehicle is AI (1) or Human␣
↪(0) controlled
    uint8_t     m_driverId;               // Driver id - see appendix
    uint8_t     m_teamId;                 // Team id - see appendix
    uint8_t     m_raceNumber;             // Race number of the car
    uint8_t     m_nationality;            // Nationality of the driver
    char        m_name[48];               // Name of participant in UTF-8 format -␣
↪null terminated
                                          // Will be truncated with ... (U+2026) if␣
↪too long
    uint8_t     m_yourTelemetry;          // The player's UDP setting, 0 = restricted,␣
↪1 = public
};

struct PacketParticipantsData
{
    PacketHeader    m_header;             // Header

    uint8           m_numActiveCars;      // Number of active cars in the data -␣
↪should match number of
                                          // cars on HUD
    ParticipantData m_participants[20];
};
```

Car Setups Packet

This packet details the car setups for each vehicle in the session. Note that in multiplayer games, other player cars will appear as blank, you will only be able to see your car setup and AI cars.

Frequency: 2 per second
Size: 843 bytes
Version: 1

```
struct CarSetupData
{
    uint8_t    m_frontWing;               // Front wing aero
    uint8_t    m_rearWing;                // Rear wing aero
    uint8_t    m_onThrottle;              // Differential adjustment on throttle␣
↪(percentage)
    uint8_t    m_offThrottle;             // Differential adjustment off throttle␣
↪(percentage)
    float      m_frontCamber;             // Front camber angle (suspension geometry)
    float      m_rearCamber;              // Rear camber angle (suspension geometry)
    float      m_frontToe;                // Front toe angle (suspension geometry)
    float      m_rearToe;                 // Rear toe angle (suspension geometry)
    uint8_t    m_frontSuspension;         // Front suspension
    uint8_t    m_rearSuspension;          // Rear suspension
    uint8_t    m_frontAntiRollBar;        // Front anti-roll bar
    uint8_t    m_rearAntiRollBar;         // Rear anti-roll bar
    uint8_t    m_frontSuspensionHeight;   // Front ride height
    uint8_t    m_rearSuspensionHeight;    // Rear ride height
    uint8_t    m_brakePressure;           // Brake pressure (percentage)
```

(continues on next page)

(continued from previous page)

```
    uint8_t    m_brakeBias;                // Brake bias (percentage)
    float      m_frontTyrePressure;        // Front tyre pressure (PSI)
    float      m_rearTyrePressure;         // Rear tyre pressure (PSI)
    uint8_t    m_ballast;                  // Ballast
    float      m_fuelLoad;                 // Fuel load
};

struct PacketCarSetupData
{
    PacketHeader    m_header;            // Header

    CarSetupData    m_carSetups[20];
};
```

Car Telemetry Packet

This packet details telemetry for all the cars in the race. It details various values that would be recorded on the car such as speed, throttle application, DRS etc.

Frequency: Rate as specified in menus
Size: 1347 bytes
Version: 1

```
struct CarTelemetryData
{
    uint16_t  m_speed;                    // Speed of car in kilometres per hour
    float     m_throttle;                 // Amount of throttle applied (0.0 to 1.0)
    float     m_steer;                    // Steering (-1.0 (full lock left) to 1.0␣
↪(full lock right))
    float     m_brake;                    // Amount of brake applied (0.0 to 1.0)
    uint8_t   m_clutch;                   // Amount of clutch applied (0 to 100)
    int8_t    m_gear;                     // Gear selected (1-8, N=0, R=-1)
    uint16_t  m_engineRPM;                // Engine RPM
    uint8_t   m_drs;                      // 0 = off, 1 = on
    uint8_t   m_revLightsPercent;         // Rev lights indicator (percentage)
    uint16_t  m_brakesTemperature[4];     // Brakes temperature (celsius)
    uint16_t  m_tyresSurfaceTemperature[4]; // Tyres surface temperature (celsius)
    uint16_t  m_tyresInnerTemperature[4]; // Tyres inner temperature (celsius)
    uint16_t  m_engineTemperature;        // Engine temperature (celsius)
    float     m_tyresPressure[4];         // Tyres pressure (PSI)
    uint8_t   m_surfaceType[4];           // Driving surface, see appendices
};

struct PacketCarTelemetryData
{
    PacketHeader     m_header;             // Header

    CarTelemetryData m_carTelemetryData[20];

    uint32_t         m_buttonStatus;        // Bit flags specifying which buttons are␣
↪being pressed
                                            // currently - see appendices
```

(continues on next page)

(continued from previous page)

```
};
```

Car Status Packet

This packet details car statuses for all the cars in the race. It includes values such as the damage readings on the car.

Frequency: Rate as specified in menus
Size: 1143 bytes
Version: 1

```
struct CarStatusData
{
    uint8_t      m_tractionControl;          // 0 (off) - 2 (high)
    uint8_t      m_antiLockBrakes;           // 0 (off) - 1 (on)
    uint8_t      m_fuelMix;                  // Fuel mix - 0 = lean, 1 = standard, 2 =␣
↪rich, 3 = max
    uint8_t      m_frontBrakeBias;           // Front brake bias (percentage)
    uint8_t      m_pitLimiterStatus;         // Pit limiter status - 0 = off, 1 = on
    float        m_fuelInTank;               // Current fuel mass
    float        m_fuelCapacity;             // Fuel capacity
    float        m_fuelRemainingLaps;        // Fuel remaining in terms of laps (value␣
↪on MFD)
    uint16_t     m_maxRPM;                   // Cars max RPM, point of rev limiter
    uint16_t     m_idleRPM;                  // Cars idle RPM
    uint8_t      m_maxGears;                 // Maximum number of gears
    uint8_t      m_drsAllowed;               // 0 = not allowed, 1 = allowed, -1 =␣
↪unknown
    uint8_t      m_tyresWear[4];             // Tyre wear percentage
    uint8_t      m_actualTyreCompound;       // F1 Modern - 16 = C5, 17 = C4, 18 = C3,␣
↪19 = C2, 20 = C1
                                             // 7 = inter, 8 = wet
                                             // F1 Classic - 9 = dry, 10 = wet
                                             // F2 - 11 = super soft, 12 = soft, 13 =␣
↪medium, 14 = hard
                                             // 15 = wet
    uint8_t      m_tyreVisualCompound;       // F1 visual (can be different from␣
↪actual compound)
                                             // 16 = soft, 17 = medium, 18 = hard, 7 =␣
↪inter, 8 = wet
                                             // F1 Classic - same as above
                                             // F2 - same as above
    uint8_t      m_tyresDamage[4];           // Tyre damage (percentage)
    uint8_t      m_frontLeftWingDamage;      // Front left wing damage (percentage)
    uint8_t      m_frontRightWingDamage;     // Front right wing damage (percentage)
    uint8_t      m_rearWingDamage;           // Rear wing damage (percentage)
    uint8_t      m_engineDamage;             // Engine damage (percentage)
    uint8_t      m_gearBoxDamage;            // Gear box damage (percentage)
    int8_t       m_vehicleFiaFlags;          // -1 = invalid/unknown, 0 = none, 1 =␣
↪green
                                             // 2 = blue, 3 = yellow, 4 = red
    float        m_ersStoreEnergy;           // ERS energy store in Joules
    uint8_t      m_ersDeployMode;            // ERS deployment mode, 0 = none, 1 = low,
↪ 2 = medium
```

(continues on next page)

(continued from previous page)

```
                                                // 3 = high, 4 = overtake, 5 = hotlap
    float       m_ersHarvestedThisLapMGUK;  // ERS energy harvested this lap by MGU-K
    float       m_ersHarvestedThisLapMGUH;  // ERS energy harvested this lap by MGU-H
    float       m_ersDeployedThisLap;       // ERS energy deployed this lap
};

struct PacketCarStatusData
{
    PacketHeader     m_header;          // Header

    CarStatusData    m_carStatusData[20];
};
```

Restricted data (Your Telemetry setting)

There is some data in the UDP that you may not want other players seeing if you are in a multiplayer game. This is controlled by the "Your Telemetry" setting in the Telemetry options. The options are:

- Restricted (Default) – other players viewing the UDP data will not see values for your car
- Public – all other players can see all the data for your car

Note: You can always see the data for the car you are driving regardless of the setting.

The following data items are set to zero if the player driving the car in question has their "Your Telemetry" set to "Restricted":

Car status packet

- m_fuelInTank
- m_fuelCapacity
- m_fuelMix
- m_fuelRemainingLaps
- m_frontBrakeBias
- m_frontLeftWingDamage
- m_frontRightWingDamage
- m_rearWingDamage
- m_engineDamage
- m_gearBoxDamage
- m_tyresWear (All four wheels)
- m_tyresDamage (All four wheels)
- m_ersDeployMode
- m_ersStoreEnergy
- m_ersDeployedThisLap
- m_ersHarvestedThisLapMGUK

- m_ersHarvestedThisLapMGUH

2.2.2 Appendices

Here are the values used for the team ID, driver ID and track ID parameters.

N.B. Driver IDs in network games differ from the actual driver IDs. All the IDs of human players start at 100 and are unique within the game session, but don't directly correlate to the player.

Team IDs

ID	Team	ID	Team	ID	Team
0	Mercedes	21	Red Bull 2010	63	Ferrari 1990
1	Ferrari	22	Ferrari 1976	64	McLaren 2010
2	Red Bull Racing	23	ART Grand Prix	65	Ferrari 2010
3	Williams	24	Campos Vexatec Racing		
4	Racing Point	25	Carlin		
5	Renault	26	Charouz Racing System		
6	Toro Rosso	27	DAMS		
7	Haas	28	Russian Time		
8	McLaren	29	MP Motorsport		
9	Alfa Romeo	30	Pertamina		
10	McLaren 1988	31	McLaren 1990		
11	McLaren 1991	32	Trident		
12	Williams 1992	33	BWT Arden		
13	Ferrari 1995	34	McLaren 1976		
14	Williams 1996	35	Lotus 1972		
15	McLaren 1998	36	Ferrari 1979		
16	Ferrari 2002	37	McLaren 1982		
17	Ferrari 2004	38	Williams 2003		
18	Renault 2006	39	Brawn 2009		
19	Ferrari 2007	40	Lotus 1978		

Driver IDs

ID	Driver	ID	Driver	ID	Driver
0	Carlos Sainz	37	Peter Belousov	69	Ruben Meijer
1	Daniil Kvyat	38	Klimek Michalski	70	Rashid Nair
2	Daniel Ricciardo	39	Santiago Moreno	71	Jack Tremblay
6	Kimi Räikkönen	40	Benjamin Coppens	74	Antonio Giovinazzi
7	Lewis Hamilton	41	Noah Visser	75	Robert Kubica
9	Max Verstappen	42	Gert Waldmuller		
10	Nico Hulkenberg	43	Julian Quesada		
11	Kevin Magnussen	44	Daniel Jones		
12	Romain Grosjean	45	Artem Markelov		
13	Sebastian Vettel	46	Tadasuke Makino		
14	Sergio Perez	47	Sean Gelael		
15	Valtteri Bottas	48	Nyck De Vries		

Continued on next page

Table 1 – continued from previous page

ID	Driver	ID	Driver	ID	Driver
19	Lance Stroll	49	Jack Aitken		
20	Arron Barnes	50	George Russell		
21	Martin Giles	51	Maximilian Günther		
22	Alex Murray	52	Nirei Fukuzumi		
23	Lucas Roth	53	Luca Ghiotto		
24	Igor Correia	54	Lando Norris		
25	Sophie Levasseur	55	Sérgio Sette Câmara		
26	Jonas Schiffer	56	Louis Delétraz		
27	Alain Forest	57	Antonio Fuoco		
28	Jay Letourneau	58	Charles Leclerc		
29	Esto Saari	59	Pierre Gasly		
30	Yasar Atiyeh	62	Alexander Albon		
31	Callisto Calabresi	63	Nicholas Latifi		
32	Naota Izum	64	Dorian Boccolacci		
33	Howard Clarke	65	Niko Kari		
34	Wilhelm Kaufmann	66	Roberto Merhi		
35	Marie Laursen	67	Arjun Maini		
36	Flavio Nieves	68	Alessio Lorandi		

Track IDs

ID	Track
0	Melbourne
1	Paul Ricard
2	Shanghai
3	Sakhir (Bahrain)
4	Catalunya
5	Monaco
6	Montreal
7	Silverstone
8	Hockenheim
9	Hungaroring
10	Spa
11	Monza
12	Singapore
13	Suzuka
14	Abu Dhabi
15	Texas
16	Brazil
17	Austria
18	Sochi
19	Mexico
20	Baku (Azerbaijan)
21	Sakhir Short
22	Silverstone Short
23	Texas Short
24	Suzuka Short

Nationality IDs

ID	Nationality	ID	Nationality	ID	Nationality
1	American	31	Greek	61	Panamanian
2	Argentinian	32	Guatemalan	62	Paraguayan
3	Australian	33	Honduran	63	Peruvian
4	Austrian	34	Hong Konger	64	Polish
5	Azerbaijani	35	Hungarian	65	Portuguese
6	Bahraini	36	Icelander	66	Qatari
7	Belgian	37	Indian	67	Romanian
8	Bolivian	38	Indonesian	68	Russian
9	Brazilian	39	Irish	69	Salvadoran
10	British	40	Israeli	70	Saudi
11	Bulgarian	41	Italian	71	Scottish
12	Cameroonian	42	Jamaican	72	Serbian
13	Canadian	43	Japanese	73	Singaporean
14	Chilean	44	Jordanian	74	Slovakian
15	Chinese	45	Kuwaiti	75	Slovenian
16	Colombian	46	Latvian	76	South Korean
17	Costa Rican	47	Lebanese	77	South African
18	Croatian	48	Lithuanian	78	Spanish
19	Cypriot	49	Luxembourger	79	Swedish
20	Czech	50	Malaysian	80	Swiss
21	Danish	51	Maltese	81	Thai
22	Dutch	52	Mexican	82	Turkish
23	Ecuadorian	53	Monegasque	83	Uruguayan
24	English	54	New Zealander	84	Ukrainian
25	Emirian	55	Nicaraguan	85	Venezuelan
26	Estonian	56	North Korean	86	Welsh
27	Finnish	57	Northern Irish		
28	French	58	Norwegian		
29	German	59	Omani		
30	Ghanaian	60	Pakistani		

Surface types

These types are from physics data and show what type of contact each wheel is experiencing.

ID	Surface
0	Tarmac
1	Rumble strip
2	Concrete
3	Rock
4	Gravel
5	Mud
6	Sand
7	Grass
8	Water
9	Cobblestone
10	Metal
11	Ridged

Button flags

These flags are used in the telemetry packet to determine if any buttons are being held on the controlling device. If the value below logical ANDed with the button status is set then the corresponding button is being held.

Bit flags	Button
0x0001	Cross or A
0x0002	Triangle or Y
0x0004	Circle or B
0x0008	Square or X
0x0010	D-pad Left
0x0020	D-pad Right
0x0040	D-pad Up
0x0080	D-pad Down
0x0100	Options or Menu
0x0200	L1 or LB
0x0400	R1 or RB
0x0800	L2 or LT
0x1000	R2 or RT
0x2000	Left Stick Click
0x4000	Right Stick Click