f1-2020-telemetry

Release 0.2.1

Guillaume Parent

Dec 10, 2020

CONTENTS

The *f1-2020-telemetry* package provides support for interpreting telemetry information as sent out over the network by the F1 2020 game by CodeMasters. It also provides *command line tools* to record, playback, and monitor F1 2020 session data.

CHAPTER

ONE

PROJECT INFORMATION

The *f1-2020-telemetry* package and its documentation are currently at version **0.2.1**.

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 f1-2020-telemetry
```

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

https://gitlab.com/gparent/f1-2020-telemetry/

The pip-installable package is hosted on PyPI:

https://pypi.org/project/f1-2020-telemetry/

The documentation is hosted on Read the Docs:

https://f1-2020-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 f1-2020-telemetry package, and documents the included command-line tools.
- The *F1 2020 Telemetry Packet Specification* is a non-authoritative copy of the CodeMasters telemetry packet specification, with some corrections applied.

2.1 Package Documentation

The *f1-2020-telemetry* package provides support for interpreting telemetry information as sent out over the network by the F1 2020 game by CodeMasters. It also provides *command line tools* to record, playback, and monitor F1 2020 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 2020, the packet format is described here:

> https://forums.codemasters.com/topic/50942-f1-2020-udp-specification/

A formatted version of this specification, with some small issues fixed, is included in the *f1-2020-telemetry* package and can be found *here*.

The *f1-2020-telemetry* package should work on Python 3.6 and above.

2.1.1 Installation

The f1-2020-telemetry package is hosted on PyPI. To install it in your Python 3 environment, type:

```
pip3 install f1-2020-telemetry
```

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

```
import f1_2020_telemetry.packet

help(f1_2020_telemetry.packet)
```

Apart from the *f1_2020_telemetry* package (and its main module *f1_2020_telemetry.packet*), the `pip3 install` command will also install some command-line utilities that can be used to record, playback, and monitor F1 2020 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 2020 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:

```
"""
Listen to telemetry packets and print them to standard output
"""
import socket

from f1_2020_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 2020 game uses to send packages; it then waits for packages and, upon reception, prints their full contents.

To generate some data, start your F1 2020 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 *2020*.

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_2020_telemetry.cli.monitor* and *f1_2020_telemetry.cli.recorder* scripts. Note that those examples are considerably more complicated because they use multi-threading.

2.1.3 Command Line Tools

The f1-2020-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.

f1-2020-telemetry-recorder script

```
usage: f1-2020-telemetry-recorder [-h] [-p PORT] [-i INTERVAL]

Record F1 2020 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)
```

f1-2020-telemetry-player script

```
usage: f1-2020-telemetry-player [-h] [-r REALTIME_FACTOR] [-d DESTINATION] [-p PORT]␣
↪filename

Replay an F1 2020 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)
```

f1-2020-telemetry-monitor script

```
usage: f1-2020-telemetry-monitor [-h] [-p PORT]

Monitor UDP port for incoming F1 2020 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_2020_telemetry.packets

Module *f1_2020_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 2020 UDP Telemetry support package

This package is based on the CodeMasters Forum post documenting the F1 2020 packet format:

    https://forums.codemasters.com/topic/54423-f1%C2%AE-2020-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".
(4) In the 'CarStatusData' structure, tyreVisualCompound was renamed to visualTyreCompound.
"""

import ctypes
import enum
from typing import Dict

##########################################################
#                                                        #
#  ___________  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 field in self._fields_:
            fname = field[0]
```

(continues on next page)

(continued from previous page)

```
            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 = f"{fname}={vstr}"
            fstr_list.append(fstr)
        return "{}({})".format(self.__class__.__name__, ", ".join(fstr_list))


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


class PacketHeader(PackedLittleEndianStructure):
    """The header for each of the UDP telemetry packets."""

    _fields_ = [
        ("packetFormat", ctypes.c_uint16),
        ("gameMajorVersion", ctypes.c_uint8),
        ("gameMinorVersion", ctypes.c_uint8),
        ("packetVersion", ctypes.c_uint8),
        ("packetId", ctypes.c_uint8),
        ("sessionUID", ctypes.c_uint64),
        ("sessionTime", ctypes.c_float),
        ("frameIdentifier", ctypes.c_uint32),
        ("playerCarIndex", ctypes.c_uint8),
        ("secondaryPlayerCarIndex", ctypes.c_uint8),
    ]


@enum.unique
class PacketID(enum.IntEnum):
    """Value as specified in the PacketHeader.packetId header field, used to␣
↪distinguish packet types."""

    _ignore_ = "long_description short_description"

    MOTION = 0
    SESSION = 1
    LAP_DATA = 2
    EVENT = 3
    PARTICIPANTS = 4
    CAR_SETUPS = 5
    CAR_TELEMETRY = 6
    CAR_STATUS = 7
    FINAL_CLASSIFICATION = 8
    LOBBY_INFO = 9
```

(continues on next page)

(continued from previous page)

```
    long_description: Dict[enum.IntEnum, str]
    short_description: Dict[enum.IntEnum, str]


PacketID.short_description = {
    PacketID.MOTION: "Motion",
    PacketID.SESSION: "Session",
    PacketID.LAP_DATA: "Lap Data",
    PacketID.EVENT: "Event",
    PacketID.PARTICIPANTS: "Participants",
    PacketID.CAR_SETUPS: "Car Setups",
    PacketID.CAR_TELEMETRY: "Car Telemetry",
    PacketID.CAR_STATUS: "Car Status",
    PacketID.FINAL_CLASSIFICATION: "Final Classification",
    PacketID.LOBBY_INFO: "Lobby information",
}


PacketID.long_description = {
    PacketID.MOTION: "Contains all motion data for player's car - only sent while␣
↪player is in control",
    PacketID.SESSION: "Data about the session - track, time left",
    PacketID.LAP_DATA: "Data about all the lap times of cars in the session",
    PacketID.EVENT: "Various notable events that happen during a session",
    PacketID.PARTICIPANTS: "List of participants in the session, mostly relevant for␣
↪multiplayer",
    PacketID.CAR_SETUPS: "Packet detailing car setups for cars in the race",
    PacketID.CAR_TELEMETRY: "Telemetry data for all cars",
    PacketID.CAR_STATUS: "Status data for all cars such as damage",
    PacketID.FINAL_CLASSIFICATION: "Final classification confirmation at the end of a␣
↪race",
    PacketID.LOBBY_INFO: "Information about players in a multiplayer lobby",
}

############################################################
#                                                          #
#  ___________  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),
        ("worldPositionY", ctypes.c_float),
        ("worldPositionZ", ctypes.c_float),
        ("worldVelocityX", ctypes.c_float),
        ("worldVelocityY", ctypes.c_float),
        ("worldVelocityZ", ctypes.c_float),
        ("worldForwardDirX", ctypes.c_int16),
        ("worldForwardDirY", ctypes.c_int16),
        ("worldForwardDirZ", ctypes.c_int16),
        ("worldRightDirX", ctypes.c_int16),
```

(continues on next page)

(continued from previous page)

```
        ("worldRightDirY", ctypes.c_int16),
        ("worldRightDirZ", ctypes.c_int16),
        ("gForceLateral", ctypes.c_float),
        ("gForceLongitudinal", ctypes.c_float),
        ("gForceVertical", ctypes.c_float),
        ("yaw", ctypes.c_float),
        ("pitch", ctypes.c_float),
        ("roll", ctypes.c_float),
    ]


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: 1464 bytes
    Version: 1
    """

    _fields_ = [
        ("header", PacketHeader),
        ("carMotionData", CarMotionData_V1 * 22),
        # Extra player car ONLY data
        ("suspensionPosition", ctypes.c_float * 4),
        ("suspensionVelocity", ctypes.c_float * 4),
        ("suspensionAcceleration", ctypes.c_float * 4),
        ("wheelSpeed", ctypes.c_float * 4),
        ("wheelSlip", ctypes.c_float * 4),
        ("localVelocityX", ctypes.c_float),
        ("localVelocityY", ctypes.c_float),
        ("localVelocityZ", ctypes.c_float),
        ("angularVelocityX", ctypes.c_float),
        ("angularVelocityY", ctypes.c_float),
        ("angularVelocityZ", ctypes.c_float),
        ("angularAccelerationX", ctypes.c_float),
        ("angularAccelerationY", ctypes.c_float),
        ("angularAccelerationZ", ctypes.c_float),
        ("frontWheelsAngle", ctypes.c_float),
    ]


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


class MarshalZone_V1(PackedLittleEndianStructure):
```

(continues on next page)

(continued from previous page)

```
    """This type is used for the 21-element 'marshalZones' array of the␣
↪PacketSessionData_V1 type, defined below."""

    _fields_ = [("zoneStart", ctypes.c_float), ("zoneFlag", ctypes.c_int8)]


class WeatherForecastSample(PackedLittleEndianStructure):
    """This type is used for the 20-element 'weatherForecastSamples' array of the␣
↪PacketSessionData_V1 type, defined below."""

    _fields_ = [
        ("sessionType", ctypes.c_uint8),
        ("timeOffset", ctypes.c_uint8),
        ("weather", ctypes.c_uint8),
        ("trackTemperature", ctypes.c_int8),
        ("airTemperature", ctypes.c_int8),
    ]


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

    Frequency: 2 per second
    Size: 251 bytes
    Version: 1
    """

    _fields_ = [
        ("header", PacketHeader),
        ("weather", ctypes.c_uint8),
        ("trackTemperature", ctypes.c_int8),
        ("airTemperature", ctypes.c_int8),
        ("totalLaps", ctypes.c_uint8),
        ("trackLength", ctypes.c_uint16),
        ("sessionType", ctypes.c_uint8),
        ("trackId", ctypes.c_int8),
        ("formula", ctypes.c_uint8),
        ("sessionTimeLeft", ctypes.c_uint16),
        ("sessionDuration", ctypes.c_uint16),
        ("pitSpeedLimit", ctypes.c_uint8),
        ("gamePaused", ctypes.c_uint8),
        ("isSpectating", ctypes.c_uint8),
        ("spectatorCarIndex", ctypes.c_uint8),
        ("sliProNativeSupport", ctypes.c_uint8),
        ("numMarshalZones", ctypes.c_uint8),
        ("marshalZones", MarshalZone_V1 * 21),
        ("safetyCarStatus", ctypes.c_uint8),
        ("networkGame", ctypes.c_uint8),
        ("numWeatherForecastSamples", ctypes.c_uint8),
        ("weatherForecastSamples", WeatherForecastSample * 20),
    ]


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

(continues on next page)

(continued from previous page)

```
##############################################################


class LapData_V1(PackedLittleEndianStructure):
    """This type is used for the 22-element 'lapData' array of the PacketLapData_V1 type, defined below."""

    _fields_ = [
        ("lastLapTime", ctypes.c_float),
        ("currentLapTime", ctypes.c_float),
        ("sector1TimeInMS", ctypes.c_uint16),
        ("sector2TimeInMS", ctypes.c_uint16),
        ("bestLapTime", ctypes.c_float),
        ("bestLapNum", ctypes.c_uint8),
        ("bestLapSector1TimeInMS", ctypes.c_uint16),
        ("bestLapSector2TimeInMS", ctypes.c_uint16),
        ("bestLapSector3TimeInMS", ctypes.c_uint16),
        ("bestOverallSector1TimeInMS", ctypes.c_uint16),
        ("bestOverallSector1LapNum", ctypes.c_uint8),
        ("bestOverallSector2TimeInMS", ctypes.c_uint16),
        ("bestOverallSector2LapNum", ctypes.c_uint8),
        ("bestOverallSector3TimeInMS", ctypes.c_uint16),
        ("bestOverallSector3LapNum", ctypes.c_uint8),
        ("lapDistance", ctypes.c_float),
        ("totalDistance", ctypes.c_float),
        ("safetyCarDelta", ctypes.c_float),
        ("carPosition", ctypes.c_uint8),
        ("currentLapNum", ctypes.c_uint8),
        ("pitStatus", ctypes.c_uint8),
        ("sector", ctypes.c_uint8),
        ("currentLapInvalid", ctypes.c_uint8),
        ("penalties", ctypes.c_uint8),
        ("gridPosition", ctypes.c_uint8),
        ("driverStatus", ctypes.c_uint8),
        ("resultStatus", ctypes.c_uint8),
    ]


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

    Frequency: Rate as specified in menus
    Size: 1190 bytes
    Version: 1
    """

    _fields_ = [
        ("header", PacketHeader),  # Header
        ("lapData", LapData_V1 * 22),  # Lap data for all cars on track
    ]


##############################################################
#                                                            #
#  ___________  Packet ID 3 : EVENT PACKET  ___________     #
#                                                            #
##############################################################
```

(continues on next page)

(continued from previous page)

```
class FastestLapData(PackedLittleEndianStructure):
    """Event data for fastest lap (FTLP)"""

    _fields_ = [
        ("vehicleIdx", ctypes.c_uint8),  # Vehicle index of car
        ("lapTime", ctypes.c_float),  # Lap time is in seconds
    ]


class PenaltyData(PackedLittleEndianStructure):
    """Event data for penalty (PENA)"""

    _fields_ = [
        ("penaltyType", ctypes.c_uint8),
        ("infringementType", ctypes.c_uint8),
        ("vehicleIdx", ctypes.c_uint8),
        ("otherVehicleIdx", ctypes.c_uint8),
        ("time", ctypes.c_uint8),
        ("lapNum", ctypes.c_uint8),
        ("placesGained", ctypes.c_uint8),
    ]


class RaceWinnerData(PackedLittleEndianStructure):
    """Event data for race winner (RCWN)"""

    _fields_ = [
        ("vehicleIdx", ctypes.c_uint8),
    ]


class RetirementData(PackedLittleEndianStructure):
    """Event data for retirement (RTMT)"""

    _fields_ = [
        ("vehicleIdx", ctypes.c_uint8),
    ]


class SpeedTrapData(PackedLittleEndianStructure):
    """Event data for speedtrap (SPTP)"""

    _fields_ = [("vehicleIdx", ctypes.c_uint8), ("speed", ctypes.c_float)]


class TeamMateInPitsData(PackedLittleEndianStructure):
    """Event data for teammate in pits (TMPT)"""

    _fields_ = [
        ("vehicleIdx", ctypes.c_uint8),
    ]


class EventDataDetails(ctypes.Union):
    """Union for the different event data types"""
```

(continues on next page)

(continued from previous page)

```
    _fields_ = [
        ("fastestLap", FastestLapData),
        ("penalty", PenaltyData),
        ("raceWinner", RaceWinnerData),
        ("retirement", RetirementData),
        ("speedTrap", SpeedTrapData),
        ("teamMateInPits", TeamMateInPitsData),
    ]


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

    Frequency: When the event occurs
    Size: 35 bytes
    Version: 1
    """

    _fields_ = [
        ("header", PacketHeader),  # Header
        ("eventStringCode", ctypes.c_char * 4),
        (
            "eventDetails",
            EventDataDetails,
        ),
    ]

    def __repr__(self):
        event = self.eventStringCode.decode()

        if event in ["CHQF", "DRSD", "DRSE", "SEND", "SSTA"]:
            end = ")"
        else:
            if event == "FTLP":
                event_details = self.eventDetails.fastestLap
            elif event == "PENA":
                event_details = self.eventDetails.penalty
            elif event == "RCWN":
                event_details = self.eventDetails.raceWinner
            elif event == "RTMT":
                event_details = self.eventDetails.retirement
            elif event == "SPTP":
                event_details = self.eventDetails.speedTrap
            elif event == "TMPT":
                event_details = self.eventDetails.teamMateInPits
            else:
                raise RuntimeError(f"Bad event code {event}")

            end = f", eventDetails={event_details!r})"

        return f"{self.__class__.__name__}(header={self.header!r}, eventStringCode=
↪{self.eventStringCode!r}{end}"


@enum.unique
class EventStringCode(enum.Enum):
```

(continues on next page)

(continued from previous page)

```
    """Value as specified in the PacketEventData_V1.eventStringCode header field, used to distinguish packet types."""

    _ignore_ = "long_description short_description"

    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"
    PENA = b"PENA"
    SPTP = b"SPTP"

    long_description: Dict[enum.Enum, str]
    short_description: Dict[enum.Enum, str]


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.PENA: "Penalty issued",
    EventStringCode.SPTP: "Speed trap triggered",
}


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",
    EventStringCode.PENA: "A penalty has been issued",
    EventStringCode.SPTP: "Speed trap has been triggered",
}

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


```

(continues on next page)

(continued from previous page)

```
class ParticipantData_V1(PackedLittleEndianStructure):
    """This type is used for the 22-element 'participants' array of the␣
↪PacketParticipantsData_V1 type, defined below."""

    _fields_ = [
        ("aiControlled", ctypes.c_uint8),
        ("driverId", ctypes.c_uint8),
        ("teamId", ctypes.c_uint8),
        ("raceNumber", ctypes.c_uint8),
        ("nationality", ctypes.c_uint8),
        ("name", ctypes.c_char * 48),
        ("yourTelemetry", ctypes.c_uint8),
    ]


class PacketParticipantsData_V1(PackedLittleEndianStructure):
    """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.
    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.

    Frequency: Every 5 seconds
    Size: 1213 bytes
    Version: 1
    """

    _fields_ = [
        ("header", PacketHeader),
        ("numActiveCars", ctypes.c_uint8),
        ("participants", ParticipantData_V1 * 22),
    ]


##############################################################
#                                                            #
#  ___________  Packet ID 5 : CAR SETUPS PACKET  ___________  #
#                                                            #
##############################################################


class CarSetupData_V1(PackedLittleEndianStructure):
    """This type is used for the 22-element 'carSetups' array of the␣
↪PacketCarSetupData_V1 type, defined below."""

    _fields_ = [
        ("frontWing", ctypes.c_uint8),
        ("rearWing", ctypes.c_uint8),
        ("onThrottle", ctypes.c_uint8),
        ("offThrottle", ctypes.c_uint8),
        ("frontCamber", ctypes.c_float),
        ("rearCamber", ctypes.c_float),
        ("frontToe", ctypes.c_float),
        ("rearToe", ctypes.c_float),
```

(continues on next page)

(continued from previous page)

```
        ("frontSuspension", ctypes.c_uint8),
        ("rearSuspension", ctypes.c_uint8),
        ("frontAntiRollBar", ctypes.c_uint8),
        ("rearAntiRollBar", ctypes.c_uint8),
        ("frontSuspensionHeight", ctypes.c_uint8),
        ("rearSuspensionHeight", ctypes.c_uint8),
        ("brakePressure", ctypes.c_uint8),
        ("brakeBias", ctypes.c_uint8),
        ("rearLeftTyrePressure", ctypes.c_float),
        ("rearRightTyrePressure", ctypes.c_float),
        ("frontLeftTyrePressure", ctypes.c_float),
        ("frontRightTyrePressure", ctypes.c_float),
        ("ballast", ctypes.c_uint8),
        ("fuelLoad", ctypes.c_float),
    ]


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: 1102 bytes
    Version: 1
    """

    _fields_ = [("header", PacketHeader), ("carSetups", CarSetupData_V1 * 22)]


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


class CarTelemetryData_V1(PackedLittleEndianStructure):
    """This type is used for the 22-element 'carTelemetryData' array of the PacketCarTelemetryData_V1 type, defined below."""

    _fields_ = [
        ("speed", ctypes.c_uint16),
        ("throttle", ctypes.c_float),
        ("steer", ctypes.c_float),
        ("brake", ctypes.c_float),
        ("clutch", ctypes.c_uint8),
        ("gear", ctypes.c_int8),
        ("engineRPM", ctypes.c_uint16),
        ("drs", ctypes.c_uint8),
        ("revLightsPercent", ctypes.c_uint8),
        ("brakesTemperature", ctypes.c_uint16 * 4),
        ("tyresSurfaceTemperature", ctypes.c_uint8 * 4),
        ("tyresInnerTemperature", ctypes.c_uint8 * 4),
        ("engineTemperature", ctypes.c_uint16),
        ("tyresPressure", ctypes.c_float * 4),
```

(continues on next page)

(continued from previous page)

```
        ("surfaceType", ctypes.c_uint8 * 4),
    ]


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: 1307 bytes
    Version: 1
    """

    _fields_ = [
        ("header", PacketHeader),
        ("carTelemetryData", CarTelemetryData_V1 * 22),
        ("buttonStatus", ctypes.c_uint32),
        ("mfdPanelIndex", ctypes.c_uint8),
        ("mfdPanelIndexSecondaryPlayer", ctypes.c_uint8),
        ("suggestedGear", ctypes.c_int8),
    ]


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


class CarStatusData_V1(PackedLittleEndianStructure):
    """This type is used for the 22-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.

    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
```

(continues on next page)

(continued from previous page)

```
        frontRightWingDamage
        rearWingDamage
        engineDamage
        gearBoxDamage
        tyresWear (All four wheels)
        tyresDamage (All four wheels)
        ersDeployMode
        ersStoreEnergy
        ersDeployedThisLap
        ersHarvestedThisLapMGUK
        ersHarvestedThisLapMGUH
        tyresAgeLaps
    """

    _fields_ = [
        ("tractionControl", ctypes.c_uint8),
        ("antiLockBrakes", ctypes.c_uint8),
        ("fuelMix", ctypes.c_uint8),
        ("frontBrakeBias", ctypes.c_uint8),
        ("pitLimiterStatus", ctypes.c_uint8),
        ("fuelInTank", ctypes.c_float),
        ("fuelCapacity", ctypes.c_float),
        ("fuelRemainingLaps", ctypes.c_float),
        ("maxRPM", ctypes.c_uint16),
        ("idleRPM", ctypes.c_uint16),
        ("maxGears", ctypes.c_uint8),
        ("drsAllowed", ctypes.c_uint8),
        ("drsActivationDistance", ctypes.c_uint16),
        ("tyresWear", ctypes.c_uint8 * 4),
        ("actualTyreCompound", ctypes.c_uint8),
        ("visualTyreCompound", ctypes.c_uint8),
        ("tyresAgeLaps", ctypes.c_uint8),
        ("tyresDamage", ctypes.c_uint8 * 4),
        ("frontLeftWingDamage", ctypes.c_uint8),
        ("frontRightWingDamage", ctypes.c_uint8),
        ("rearWingDamage", ctypes.c_uint8),
        ("drsFault", ctypes.c_uint8),
        ("engineDamage", ctypes.c_uint8),
        ("gearBoxDamage", ctypes.c_uint8),
        ("vehicleFiaFlags", ctypes.c_int8),
        ("ersStoreEnergy", ctypes.c_float),
        ("ersDeployMode", ctypes.c_uint8),
        ("ersHarvestedThisLapMGUK", ctypes.c_float),
        ("ersHarvestedThisLapMGUH", ctypes.c_float),
        ("ersDeployedThisLap", ctypes.c_float),
    ]


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: 1344 bytes
    Version: 1
    """
```

(continues on next page)

(continued from previous page)

```
    _fields_ = [
        ("header", PacketHeader),  # Header
        ("carStatusData", CarStatusData_V1 * 22),
    ]


##############################################################
#                                                            #
#  _______  Packet ID 8 : FINAL CLASSIFICATION PACKET _____  #
#                                                            #
##############################################################


class FinalClassificationData_V1(PackedLittleEndianStructure):
    """
    This type is used for the 22-element 'classificationData' array of the PacketFinalClassificationData_V1 type, defined below.
    """

    _fields_ = [
        ("position", ctypes.c_uint8),
        ("numLaps", ctypes.c_uint8),
        ("gridPosition", ctypes.c_uint8),
        ("points", ctypes.c_uint8),
        ("numPitStops", ctypes.c_uint8),
        ("resultStatus", ctypes.c_uint8),
        ("bestLapTime", ctypes.c_float),
        ("totalRaceTime", ctypes.c_double),
        ("penaltiesTime", ctypes.c_uint8),
        ("numPenalties", ctypes.c_uint8),
        ("numTyreStints", ctypes.c_uint8),
        ("tyreStintsActual", ctypes.c_uint8 * 8),
        ("tyreStintsVisual", ctypes.c_uint8 * 8),
    ]


class PacketFinalClassificationData_V1(PackedLittleEndianStructure):
    """This packet details the final classification at the end of the race.

    This data will match with the post race results screen.

    Frequency: Once at the end of the race
    Size: 839 bytes
    Version: 1
    """

    _fields_ = [
        ("header", PacketHeader),  # Header
        (
            "numCars",
            ctypes.c_uint8,
        ),  # Number of cars in the final classification
        ("classificationData", FinalClassificationData_V1 * 22),
    ]


```

(continues on next page)

(continued from previous page)

```
################################################################
#                                                              #
#  ____________  Packet ID 9 : LOBBY INFO PACKET  ____________  #
#                                                              #
################################################################


class LobbyInfoData_V1(PackedLittleEndianStructure):
    """This type is used for the 22-element 'lobbyPlayers' array of the␣
↪PacketLobbyInfoData_V1 type, defined below."""

    _fields_ = [
        ("aiControlled", ctypes.c_uint8),
        ("teamId", ctypes.c_uint8),
        ("nationality", ctypes.c_uint8),
        ("name", ctypes.c_char * 48),
        ("readyStatus", ctypes.c_uint8),
    ]


class PacketLobbyInfoData_V1(PackedLittleEndianStructure):
    """This is a list of players in a multiplayer lobby.

    Frequency: Two every second when in the lobby
    Size: 1169 bytes
    Version: 1
    """

    _fields_ = [
        ("header", PacketHeader),  # Header
        ("numPlayers", ctypes.c_uint8),
        ("lobbyPlayers", LobbyInfoData_V1 * 22),
    ]


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

# Map from (packetFormat, packetVersion, packetId) to a specific packet type.
HeaderFieldsToPacketType = {
    (2020, 1, 0): PacketMotionData_V1,
    (2020, 1, 1): PacketSessionData_V1,
    (2020, 1, 2): PacketLapData_V1,
    (2020, 1, 3): PacketEventData_V1,
    (2020, 1, 4): PacketParticipantsData_V1,
    (2020, 1, 5): PacketCarSetupData_V1,
    (2020, 1, 6): PacketCarTelemetryData_V1,
    (2020, 1, 7): PacketCarStatusData_V1,
    (2020, 1, 8): PacketFinalClassificationData_V1,
    (2020, 1, 9): PacketLobbyInfoData_V1,
}


class UnpackError(Exception):
```

(continues on next page)

(continued from previous page)

```
    """Exception for packets that cannot be unpacked"""


def unpack_udp_packet(packet: bytes) -> PacketLittleEndianStructure:
    """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(
            f"Bad telemetry packet: too short ({actual_packet_size} bytes)."
        )

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

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

    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(
                packet_type.__name__, expected_packet_size, actual_packet_size
            )
        )

    return packet_type.from_buffer_copy(packet)
```

Module: f1_2020_telemetry.cli.recorder

Module *f1_2020_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!
  (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.
"""
```

(continues on next page)

(continued from previous page)

```
39
40  import argparse
41  import sys
42  import time
43  import socket
44  import sqlite3
45  import threading
46  import logging
47  import ctypes
48  import selectors
49
50  from collections import namedtuple
51
52  from .threading_utils import WaitConsoleThread, Barrier
53  from ..packets import (
54      PacketHeader,
55      PacketID,
56      HeaderFieldsToPacketType,
57      unpack_udp_packet,
58  )
59
60  # The type used by the PacketReceiverThread to represent incoming telemetry packets,␣
    ↪with timestamp.
61  TimestampedPacket = namedtuple("TimestampedPacket", "timestamp, packet")
62
63  # The type used by the PacketRecorderThread to represent incoming telemetry packets␣
    ↪for storage in the SQLite3 database.
64  SessionPacket = namedtuple(
65      "SessionPacket",
66      "timestamp, packetFormat, gameMajorVersion, gameMinorVersion, packetVersion,␣
    ↪packetId, sessionUID, sessionTime, frameIdentifier, playerCarIndex, packet",
67  )
68
69
70  class PacketRecorder:
71      """The PacketRecorder records incoming packets to SQLite3 database files.
72
73      A single SQLite3 file stores packets from a single session.
74      Whenever a new session starts, any open file is closed, and a new database file␣
    ↪is created.
75      """
76
77      # The SQLite3 query that creates the 'packets' table in the database file.
78      _create_packets_table_query = """
79          CREATE TABLE packets (
80              pkt_id            INTEGER  PRIMARY KEY, -- Alias for SQLite3's 'rowid'.
81              timestamp         REAL     NOT NULL,    -- The POSIX time right after␣
    ↪capturing the telemetry packet.
82              packetFormat      INTEGER  NOT NULL,    -- Header field: packet format.
83              gameMajorVersion  INTEGER  NOT NULL,    -- Header field: game major␣
    ↪version.
84              gameMinorVersion  INTEGER  NOT NULL,    -- Header field: game minor␣
    ↪version.
85              packetVersion     INTEGER  NOT NULL,    -- Header field: packet version.
86              packetId          INTEGER  NOT NULL,    -- Header field: packet type (
    ↪'packetId' is a bit of a misnomer).
87              sessionUID        CHAR(16) NOT NULL,    -- Header field: unique session␣
    ↪id as hex string.
```

(continues on next page)

(continued from previous page)

```
            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 = f"F1_2019_{sessionUID}.sqlite3"
        logging.info("Opening file %s", filename)
        conn = sqlite3.connect(filename)
        cursor = conn.cursor()

        # Get rid of indentation and superfluous newlines in the 'CREATE TABLE' command.
        query = "".join(
            line[8:] + "\n"
            for line in PacketRecorder._create_packets_table_query.split("\n")[
                1:-1
            ]
        )

        # Try to execute the 'CREATE TABLE' statement. If it already exists, this will raise an exception.
        try:
            cursor.execute(query)
        except sqlite3.OperationalError:
            logging.info("    (Appending to existing file.)")
        else:
            logging.info("    (Created new file.)")

        self._conn = conn
        self._cursor = cursor
```

(continues on next page)

(continued from previous page)

```
        self._filename = filename
        self._sessionUID = sessionUID

    def _close_database(self):
        """Close SQLite3 database file."""
        assert self._conn is not None
        logging.info("Closing file %s", self._filename)
        self._cursor.close()
        self._cursor = None
        self._conn.close()
        self._conn = None
        self._filename = None
        self._sessionUID = None

    def _insert_and_commit_same_session_packets(self, same_session_packets):
        """Insert session packets to database and commit."""
        assert self._conn is not None
        self._cursor.executemany(
            PacketRecorder._insert_packets_query, same_session_packets
        )
        self._conn.commit()

    def _process_same_session_packets(self, same_session_packets):
        """Insert packets from the same session into the 'packets' table of the␣
↪appropriate database file.

        Precondition: all packets in 'same_session_packets' are from the same session␣
↪(identical 'sessionUID' field).

        We need to handle four different cases:

        (1) 'same_session_packets' is empty:

            --> return (no-op).

        (2) A database file is currently open, but it stores packets with a different␣
↪session UID:

            --> Close database;
            --> Open database with correct session UID;
            --> Insert 'same_session_packets'.

        (3) No database file is currently open:

            --> Open database with correct session UID;
            --> Insert 'same_session_packets'.

        (4) A database is currently open, with correct session UID:

            --> Insert 'same_session_packets'.
        """

        if not same_session_packets:
            # Nothing to insert.
            return

        if (
```

(continues on next page)

(continued from previous page)

```
            self._conn is not None
            and self._sessionUID != same_session_packets[0].sessionUID
        ):
            # Close database if it's recording a different session.
            self._close_database()

        if self._conn is None:
            # Open database with the correct sessionID.
            self._open_database(same_session_packets[0].sessionUID)

        # Write packets.
        self._insert_and_commit_same_session_packets(same_session_packets)

    def process_incoming_packets(self, timestamped_packets):
        """Process incoming packets by recording them into the correct database file.

        The incoming 'timestamped_packets' is a list of timestamped raw UDP packets.

        We process them to a variable 'same_session_packets', which is a list of consecutive
        packets having the same 'sessionUID' field. In this list, each packet is a 11-element tuple
        that can be inserted into the 'packets' table of the database.

        The 'same_session_packets' are then passed on to the '_process_same_session_packets'
        method that writes them into the appropriate database file.
        """

        t1 = time.monotonic()

        # Invariant to be guaranteed: all packets in 'same_session_packets' have the same 'sessionUID' field.
        same_session_packets = []

        for (timestamp, packet) in timestamped_packets:

            if len(packet) < ctypes.sizeof(PacketHeader):
                logging.error(
                    "Dropped bad packet of size {} (too short).".format(
                        len(packet)
                    )
                )
                continue

            header = PacketHeader.from_buffer_copy(packet)

            packet_type_tuple = (
                header.packetFormat,
                header.packetVersion,
                header.packetId,
            )

            packet_type = HeaderFieldsToPacketType.get(packet_type_tuple)
            if packet_type is None:
                logging.error(
                    "Dropped unrecognized packet (format, version, id) = %r.",
```

(continues on next page)

(continued from previous page)

```
                    packet_type_tuple,
                )
                continue

            if len(packet) != ctypes.sizeof(packet_type):
                logging.error(
                    "Dropped packet with unexpected size; "
                    "(format, version, id) = %r packet, size = %d, expected %d.",
                    packet_type_tuple,
                    len(packet),
                    ctypes.sizeof(packet_type),
                )
                continue

            if header.packetId == PacketID.EVENT:  # Log Event packets
                event_packet = unpack_udp_packet(packet)
                logging.info(
                    "Recording event packet: %s",
                    event_packet.eventStringCode.decode(),
                )

            # NOTE: the sessionUID is not reliable at the start of a session (in F1 2018, need to check for F1 2019).
            # 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

            # Create an INSERT-able tuple for the data in this packet.
            #
            # Note that we convert the sessionUID to a 16-digit hex string here.
            # SQLite3 can store 64-bit numbers, but only signed ones.
            # To prevent any issues, we represent the sessionUID as a 16-digit hex string instead.

            session_packet = SessionPacket(
                timestamp,
                header.packetFormat,
                header.gameMajorVersion,
                header.gameMinorVersion,
                header.packetVersion,
                header.packetId,
                f"{header.sessionUID:016x}",
                header.sessionTime,
                header.frameIdentifier,
                header.playerCarIndex,
                packet,
            )

            if (
                len(same_session_packets) > 0
                and same_session_packets[0].sessionUID
                != session_packet.sessionUID
            ):
                # Write 'same_session_packets' collected so far to the correct session database, then forget about them.
                self._process_same_session_packets(same_session_packets)
                same_session_packets.clear()

```

(continues on next page)

(continued from previous page)

```
            same_session_packets.append(session_packet)

        # Write 'same_session_packets' to the correct session database, then forget about them.
        # The 'same_session_packets.clear()' is not strictly necessary here, because 'same_session_packets' is about to
        #   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 %d packets in %.3f ms.",
            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.", age)
        else:
            logging.info(
                "No packets to record for %.3f seconds; closing file due to inactivity.",
                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(
```

(continues on next page)

(continued from previous page)

```
            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.
            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)

```

(continues on next page)

(continued from previous page)

```
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()

    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 %d",
            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:
```

(continues on next page)

(continued from previous page)

```
                    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.

    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="Record F1 2019 telemetry data to SQLite3 files."
    )

    parser.add_argument(
        "-p",
        "--port",
        default=20777,
        type=int,
        help="UDP port to listen to (default: 20777)",
        dest="port",
    )
    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",
    )

    args = parser.parse_args()

    # Start recorder thread first, then receiver thread.

    quit_barrier = Barrier()
```

(continues on next page)

(continued from previous page)

```
    recorder_thread = PacketRecorderThread(args.interval)
    recorder_thread.start()

    receiver_thread = PacketReceiverThread(args.port, recorder_thread)
    receiver_thread.start()

    wait_console_thread = WaitConsoleThread(quit_barrier)
    wait_console_thread.start()

    # Recorder, receiver, 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()

    receiver_thread.request_quit()
    receiver_thread.join()
    receiver_thread.close()

    recorder_thread.request_quit()
    recorder_thread.join()
    recorder_thread.close()

    # All done.

    logging.info("All done.")


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

Module: f1_2020_telemetry.cli.player

Module *f1_2020_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
```

(continues on next page)

(continued from previous page)

```
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)
            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.")

```

(continues on next page)

(continued from previous page)

```
        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(
                            "%d packages sent, delay: %.3f ms",
                            packet_count,
                            1000.0 * delay,
                        )
                    break

        cursor.close()
        conn.close()

        sock.close()

        self._quit_barrier.proceed()

        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():

```

(continues on next page)

(continued from previous page)

```
    # 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()
```

(continues on next page)

(continued from previous page)

```
    # 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.

    logging.info("All done.")


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

Module: f1_2020_telemetry.cli.monitor

Module *f1_2020_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()

```

(continues on next page)

(continued from previous page)

```
        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.
        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 %d",
            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()

```

(continues on next page)

(continued from previous page)

```
        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
            )
        except:
            distance = math.nan

        logging.info("frame %6d distance %10.3f", self._current_frame, distance)

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

        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.

    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="Monitor UDP port for incoming F1 2019 telemetry data and print␣
↪information."
    )

    parser.add_argument(
```

(continues on next page)

(continued from previous page)

```
        "-p",
        "--port",
        default=20777,
        type=int,
        help="UDP port to listen to (default: 20777)",
        dest="port",
    )

    args = parser.parse_args()

    # Start recorder thread first, then receiver thread.

    quit_barrier = Barrier()

    monitor_thread = PacketMonitorThread(args.port)
    monitor_thread.start()

    wait_console_thread = WaitConsoleThread(quit_barrier)
    wait_console_thread.start()

    # Monitor 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()

    monitor_thread.request_quit()
    monitor_thread.join()
    monitor_thread.close()

    # All done.

    logging.info("All done.")


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

2.2 F1 2020 Telemetry Packet Specification

Note: This specification was copied (with the minor changes listed below) from the CodeMasters forum topic describing the F1 2020 telemetry UDP packet specification, as found here:

https://forums.codemasters.com/topic/50942-f1-2020-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;
- 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*;
- In struct *PacketSessionData*: documented value 3 for *m_safetyCarStatus* (formation lap).

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)
double	Floating point (64-bit)
uint64_t	Unsigned 64-bit integer

Packet Header

Each packet has the following header:

```
struct PacketHeader
{
    uint16_t  m_packetFormat;             // 2020
    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
    uint8_t   m_secondaryPlayerCarIndex; // Index of secondary player's car in the␣
↪array (splitscreen)
                                          // 255 if no second player
};
```

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
Final Classification	8	Final classification confirmation at the end of a race
Lobby information	9	Information about players in a multiplayer lobby

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: 1464 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
{
    PacketHeader    m_header;                 // Header

    CarMotionData   m_carMotionData[22];      // 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: 251 bytes
Version: 1

```
struct MarshalZone
{
    float  m_zoneStart;   // Fraction (0..1) of way through the lap the marshal zone␣
↪starts
    int8_t m_zoneFlag;    // -1 = invalid/unknown, 0 = none, 1 = green, 2 = blue, 3 =␣
↪yellow, 4 = red
};

struct WeatherForecastSample
{
    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
    uint8_t   m_timeOffset;                   // Time in minutes the forecast is␣
↪for
    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
};

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
    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,
```

(continues on next page)

(continued from previous page)

```
                                                // 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
                                                // 3 = formation lap safety car
    uint8_t         m_networkGame;              // 0 = offline, 1 = online
    uint8_t         m_numWeatherForecastSamples; // Number of weather samples to␣
↪follow
    WeatherForecastSample m_weatherForecastSamples[20]; // Array of weather forecast␣
↪samples
};
```

Lap Data Packet

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

Frequency: Rate as specified in menus
Size: 1190 bytes
Version: 1

```
struct LapData
{
    float       m_lastLapTime;              // Last lap time in seconds
    float       m_currentLapTime;           // Current time around the lap in seconds
    uint16_t    m_sector1TimeInMS;          // Sector 1 time in milliseconds
    uint16_t    m_sector2TimeInMS;          // Sector 2 time in milliseconds
    float       m_bestLapTime;              // Best lap time of the session in␣
↪seconds
    uint8_t     m_bestLapNum;               // Lap number best time achieved on
    uint16_t    m_bestLapSector1TimeInMS;   // Sector 1 time of best lap in the␣
↪session in milliseconds
    uint16_t    m_bestLapSector2TimeInMS;   // Sector 2 time of best lap in the␣
↪session in milliseconds
    uint16_t    m_bestLapSector3TimeInMS;   // Sector 3 time of best lap in the␣
↪session in milliseconds
    uint16_t    m_bestOverallSector1TimeInMS;// Best overall sector 1 time of the␣
↪session in milliseconds
    uint8_t     m_bestOverallSector1LapNum;  // Lap number best overall sector 1 time␣
↪achieved on
    uint16_t    m_bestOverallSector2TimeInMS;// Best overall sector 2 time of the␣
↪session in milliseconds
```

(continues on next page)

(continued from previous page)

```
    uint8_t      m_bestOverallSector2LapNum;  // Lap number best overall sector 2 time␣
↪achieved on
    uint16_t     m_bestOverallSector3TimeInMS;// Best overall sector 3 time of the␣
↪session in milliseconds
    uint8_t      m_bestOverallSector3LapNum;  // Lap number best overall sector 3 time␣
↪achieved on
    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
    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[22];          // 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: 35 bytes
Version: 1

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

(continues on next page)

(continued from previous page)

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

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

    struct
    {
        uint8_t penaltyType;      // Penalty type - see Appendices
        uint8_t infringementType; // Infringement type - see Appendices
        uint8_t vehicleIdx;       // Vehicle index of the car the penalty is applied␣
↪to
        uint8_t otherVehicleIdx;  // Vehicle index of the other car involved
        uint8_t time;             // Time gained, or time spent doing action in␣
↪seconds
        uint8_t lapNum;           // Lap the penalty occurred on
        uint8_t placesGained;     // Number of places gained by this
    } Penalty;

    struct
    {
        uint8_t vehicleIdx; // Vehicle index of the vehicle triggering speed trap
        float speed;        // Top speed achieved in kilometres per hour
    } SpeedTrap;
};

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
Penalty issued	PENA	A penalty has been issued
Speed trap triggered	SPTP	Speed trap has been triggered

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: 1213 bytes
Version: 1

```
struct ParticipantData
{
    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
```

(continues on next page)

(continued from previous page)

```
                                        // cars on HUD
    ParticipantData m_participants[22];
};
```

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: 1102 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)
    uint8_t    m_brakeBias;                // Brake bias (percentage)
    float      m_rearLeftTyrePressure;     // Rear left tyre pressure (PSI)
    float      m_rearRightTyrePressure;    // Rear right tyre pressure (PSI)
    float      m_frontLeftTyrePressure;    // Front left tyre pressure (PSI)
    float      m_frontRightTyrePressure;   // Front right tyre pressure (PSI)
    uint8_t    m_ballast;                  // Ballast
    float      m_fuelLoad;                 // Fuel load
};

struct PacketCarSetupData
{
    PacketHeader    m_header;              // Header

    CarSetupData    m_carSetups[22];
};
```

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: 1307 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)
    uint8_t   m_tyresSurfaceTemperature[4]; // Tyres surface temperature (celsius)
    uint8_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[22];

    uint32_t         m_buttonStatus;        // Bit flags specifying which buttons are␣
↪being pressed
                                            // currently - see appendices
    uint8_t          m_mfdPanelIndex;       // Index of MFD panel open - 255 = MFD␣
↪closed
                                            // Single player, race - 0 = Car setup, 1␣
↪= Pits
                                            // 2 = Damage, 3 =  Engine, 4 =␣
↪Temperatures
                                            // May vary depending on game mode
    uint8_t          m_mfdPanelIndexSecondaryPlayer;   // See above
    int8_t           m_suggestedGear;       // Suggested gear for the player (1-8)
                                            // 0 if no gear suggested
};
```

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: 1344 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
    uint16_t     m_drsActivationDistance;    // 0 = DRS not available, non-zero - DRS␣
↪will be available
                                             // in [X] metres
    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 - 19 = super soft, 20 = soft, 21 =␣
↪medium, 22 = hard
                                             // 15 = wet
    uint8_t      m_tyresAgeLaps;             // Age in laps of the current set of tyres
    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_drsFault;                 // Indicator for DRS fault, 0 = OK, 1 =␣
↪fault
    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
```

(continues on next page)

(continued from previous page)

```
                                            // 2 = blue, 3 = yellow, 4 = red
    float       m_ersStoreEnergy;           // ERS energy store in Joules
    uint8_t     m_ersDeployMode;            // ERS deployment mode, 0 = none, 1 =␣
↪medium
                                            // 2 = overtake, 3 = 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[22];
};
```

Final Classification Packet

This packet details the final classification at the end of the race, and the data will match with the post race results screen. This is especially useful for multiplayer games where it is not always possible to send lap times on the final frame because of network delay.

Frequency: Once at the end of a race
Size: 839 bytes
Version: 1

```
struct FinalClassificationData
{
    uint8_t    m_position;              // Finishing position
    uint8_t    m_numLaps;               // Number of laps completed
    uint8_t    m_gridPosition;          // Grid position of the car
    uint8_t    m_points;                // Number of points scored
    uint8_t    m_numPitStops;           // Number of pit stops made
    uint8_t    m_resultStatus;          // Result status - 0 = invalid, 1 = inactive,␣
↪2 = active
                                        // 3 = finished, 4 = disqualified, 5 = not␣
↪classified
                                        // 6 = retired
    float      m_bestLapTime;           // Best lap time of the session in seconds
    double     m_totalRaceTime;         // Total race time in seconds without penalties
    uint8_t    m_penaltiesTime;         // Total penalties accumulated in seconds
    uint8_t    m_numPenalties;          // Number of penalties applied to this driver
    uint8_t    m_numTyreStints;         // Number of tyres stints up to maximum
    uint8_t    m_tyreStintsActual[8];   // Actual tyres used by this driver
    uint8_t    m_tyreStintsVisual[8];   // Visual tyres used by this driver
};

struct PacketFinalClassificationData
{
    PacketHeader               m_header;                  // Header
```

(continues on next page)

(continued from previous page)

```
    uint8                    m_numCars;                // Number of cars in the␣
↪final classification
    FinalClassificationData  m_classificationData[22];
};
```

Lobby Info Packet

This packet details the players currently in a multiplayer lobby. It details each player's selected car, any AI involved in the game and also the ready status of each of the participants.

Frequency: Two every second when in the lobby
Size: 1169 bytes
Version: 1

```
struct LobbyInfoData
{
    uint8_t    m_aiControlled; // Whether the vehicle is AI (1) or Human (0) controlled
    uint8_t    m_teamId;       // Team id - see appendix (255 if no team currently␣
↪selected)
    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_readyStatus;  // 0 = not ready, 1 = ready, 2 = spectating
};

struct PacketLobbyInfoData
{
    PacketHeader        m_header;
    uint8               m_numPlayers; // Number of players in the lobby data
    LobbyInfoData       m_lobbyPlayers[22];
};
```

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
- m_tyresAgeLaps

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	20	McLaren 2008	63	Ferrari 1990
1	Ferrari	21	Red Bull 2010	64	McLaren 2010
2	Red Bull Racing	31	McLaren 1990	65	Ferrari 2010
3	Williams	38	Williams 2003	255	My Team
4	Racing Point	39	Brawn 2009		
5	Renault	41	F1 Generic car		
6	AlphaTauri	42	ART Grand Prix		
7	Haas	43	Campos Racing		
8	McLaren	44	Carlin		
9	Alfa Romeo	45	Sauber Junior Team by Charouz		
10	McLaren 1988	46	DAMS		
11	McLaren 1991	47	UNI-Virtuosi		
12	Williams 1992	48	MP Motorsport		
13	Ferrari 1995	49	PREMA Racing		
14	Williams 1996	50	Trident		
15	McLaren 1998	51	BWT Arden		
16	Ferrari 2002	53	Benetton 1994		
17	Ferrari 2004	54	Benetton 1995		
18	Renault 2006	55	Ferrari 2000		
19	Ferrari 2007	56	Jordan 1991		

Driver IDs

ID	Driver	ID	Driver	ID	Driver
0	Carlos Sainz	36	Flavio Nieves	68	Alessio Lorandi
1	Daniil Kvyat	37	Peter Belousov	69	Ruben Meijer
2	Daniel Ricciardo	38	Klimek Michalski	70	Rashid Nair
6	Kimi Räikkönen	39	Santiago Moreno	71	Jack Tremblay
7	Lewis Hamilton	40	Benjamin Coppens	74	Antonio Giovinazzi
9	Max Verstappen	41	Noah Visser	75	Robert Kubica
10	Nico Hulkenberg	42	Gert Waldmuller	78	Nobuharu Matsushita
11	Kevin Magnussen	43	Julian Quesada	79	Nikita Mazepin
12	Romain Grosjean	44	Daniel Jones	80	Guanyu Zhou
13	Sebastian Vettel	45	Artem Markelov	81	Mick Schumacher
14	Sergio Perez	46	Tadasuke Makino	82	Callum Ilott
15	Valtteri Bottas	47	Sean Gelael	83	Juan Manuel Correa
17	Esteban Ocon	48	Nyck De Vries	84	Jordan King
19	Lance Stroll	49	Jack Aitken	85	Mahaveer Raghunathan
20	Arron Barnes	50	George Russell	86	Tatiana Calderón
21	Martin Giles	51	Maximilian Günther	87	Anthoine Hubert
22	Alex Murray	52	Nirei Fukuzumi	88	Giuliano Alesi
23	Lucas Roth	53	Luca Ghiotto	89	Ralph Boschung
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		

continues on next page

Table 1 – continued from previous page

ID	Driver	ID	Driver	ID	Driver
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		

Track IDs

ID	Track
0	Melbourne
1	Paul Ricard
2	Shanghai
3	Sakhir (Bahrain)
4	Catalunya
5	Monaco
6	Montreal
7	Silverstone
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
25	Hanoi
26	Zandvoort

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	87	Barbadian
28	French	58	Norwegian	88	Vietnamese
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

Penalty types

ID	Penalty meaning
0	Drive through
1	Stop Go
2	Grid penalty
3	Penalty reminder
4	Time penalty
5	Warning
6	Disqualified
7	Removed from formation lap
8	Parked too long timer
9	Tyre regulations
10	This lap invalidated
11	This and next lap invalidated
12	This lap invalidated without reason
13	This and next lap invalidated without reason
14	This and previous lap invalidated
15	This and previous lap invalidated without reason
16	Retired
17	Black flag timer

Infringement types

ID	Infringement meaning
0	Blocking by slow driving
1	Blocking by wrong way driving
2	Reversing off the start line
3	Big Collision
4	Small Collision
5	Collision failed to hand back position single
6	Collision failed to hand back position multiple
7	Corner cutting gained time
8	Corner cutting overtake single
9	Corner cutting overtake multiple
10	Crossed pit exit lane
11	Ignoring blue flags
12	Ignoring yellow flags
13	Ignoring drive through
14	Too many drive throughs
15	Drive through reminder serve within n laps
16	Drive through reminder serve this lap
17	Pit lane speeding
18	Parked for too long
19	Ignoring tyre regulations
20	Too many penalties
21	Multiple warnings
22	Approaching disqualification
23	Tyre regulations select single

continues on next page

Table 3 – continued from previous page

ID	Infringement meaning
24	Tyre regulations select multiple
25	Lap invalidated corner cutting
26	Lap invalidated running wide
27	Corner cutting ran wide gained time minor
28	Corner cutting ran wide gained time significant
29	Corner cutting ran wide gained time extreme
30	Lap invalidated wall riding
31	Lap invalidated flashback used
32	Lap invalidated reset to track
33	Blocking the pitlane
34	Jump start
35	Safety car to car collision
36	Safety car illegal overtake
37	Safety car exceeding allowed pace
38	Virtual safety car exceeding allowed pace
39	Formation lap below allowed speed
40	Retired mechanical failure
41	Retired terminally damaged
42	Safety car falling too far back
43	Black flag timer
44	Unserved stop go penalty
45	Unserved drive through penalty
46	Engine component change
47	Gearbox change
48	League grid penalty
49	Retry penalty
50	Illegal time gain
51	Mandatory pitstop