# Cellocator Safety Application (CSA)

# Integration Manual

Cellocator Division
Pointer Telocation Ltd.

Proprietary and Confidential

Version 1.1

Revised and Updated: March 11, 2013


Pointer Telocation Ltd. 14 Hamelacha St., Rosh Ha'ayin 48091, Israel • Tel: 972-3-5723111 • Fax: 972-3-5723100 • www.pointer.com

# Legal Notices

## IMPORTANT

1. All legal terms and safety and operating instructions should be read thoroughly before the product accompanying this document is installed and operated.
2. This document should be retained for future reference.
3. Attachments, accessories or peripheral devices not supplied or recommended in writing by Pointer Telocation Ltd. May be hazardous and/or may cause damage to the product and should not, in any circumstances, be used or combined with the product.

## General

The product accompanying this document is not designated for and should not be used in life support appliances, devices, machines or other systems of any sort where any malfunction of the product can reasonably be expected to result in injury or death. Customers of Pointer Telocation Ltd. using, integrating, and/or selling the product for use in such applications do so at their own risk and agree to fully indemnify Pointer Telocation Ltd. for any resulting loss or damages.

## Warranty Exceptions and Disclaimers

Pointer Telocation Ltd. Shall bear no responsibility and shall have no obligation under the foregoing limited warranty for any damages resulting from normal wear and tear, the cost of obtaining substitute products, or any defect that is (i) discovered by purchaser during the warranty period but purchaser does not notify Pointer Telocation Ltd. Until after the end of the warranty period, (ii) caused by any accident, force majeure, misuse, abuse, handling or testing, improper installation or unauthorized repair or modification of the product, (iii) caused by use of any software not supplied by Pointer Telocation Ltd., or by use of the product other than in accordance with its documentation, or (iv) the result of electrostatic discharge, electrical surge, fire, flood or similar causes. Unless otherwise provided in a written agreement between the purchaser and Pointer Telocation Ltd., the purchaser shall be solely responsible for the proper configuration, testing and verification of the product prior to deployment in the field.

POINTER TELOCATION LTD.'S SOLE RESPONSIBILITY AND PURCHASER'S SOLE REMEDY UNDER THIS LIMITED WARRANTY SHALL BE TO REPAIR OR REPLACE THE PRODUCT HARDWARE, SOFTWARE OR SOFTWARE MEDIA (OR IF REPAIR OR REPLACEMENT IS NOT POSSIBLE, OBTAIN A REFUND OF THE PURCHASE PRICE) AS PROVIDED ABOVE. POINTER TELOCATION LTD. EXPRESSLY DISCLAIMS ALL OTHER WARRANTIES OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, SATISFACTORY PERFORMANCE AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL POINTER TELOCATION LTD. BE LIABLE FOR ANY INDIRECT, SPECIAL, EXEMPLARY, INCIDENTAL OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOSS OR INTERRUPTION OF USE, DATA, REVENUES OR PROFITS) RESULTING FROM A BREACH OF THIS WARRANTY OR BASED ON ANY OTHER LEGAL THEORY, EVEN IF POINTER TELOCATION LTD. HAS BEEN ADVISED OF THE POSSIBILITY OR LIKELIHOOD OF SUCH DAMAGES.

## Intellectual Property

# Table of Contents

# 1 Introduction

This manual gives instructions and technical information for the system integration of Cello-IQ units with operational fleet management platforms wishing to extend their capabilities to support driver behavior, fuel efficiency and safety applications.

## 1.1 Scope

This document is part of the Cello-IQ documentation set. It is assumed the reader is familiar with Cellocator products, communication protocols and device provisioning capabilities.

## 1.2 Purpose

Using the provided interface the server developer is able to perform the following:

- Configure the safety mechanism attributes.
- Build GW protocol to interconnect with multiple Cello-IQ units.
- Handle CSA files sent via FTP/TFTP protocol (File decompression and parsing).

## 1.3 Target Audience

This manual is intended for server side developers adding Cello-IQ units to their existing Cellocator fleet management system. This manual will support two types of software integrators depending on the level of integration required by the existing server application, as described below.

The first type of integration is intended to address fleet management systems working directly with the Cellocator CSA protocol and thus self-implementing all the low level communication layers. Such integrators need to have good knowledge and understanding of the CSA modular protocol, its versions and data flow. It is also recommended to use the code samples embedded in this document to facilitate the server side development process.

The second type of integration is intended for TSPs and SW integrators who already use Cellocator's integration package or those who are new to Cellocator's APIs and OTA Interfaces and considering their optional integration paths. The Cellocator Integration package is a set of backend gateway components designed to hide the OTA-protocol's internals from the integrator by transforming the integration task from a low level, communication and protocol management task into a database level, explicit and parsed information integration mission. Customers using the Cellocator Gateway are benefitting from a quicker and easier integration process, entitled for software upgrades, technical support and more.

Apart from the protocol level integration described above, the Cello-IQ unit is also capable of sending raw data files via file transfer protocols. The server side application should be able to access the files saved by the FTP server added to the server side solution and parse the files for more detailed information and analysis.

## 1.4 References

| No. | Document Name | Version | Remark |
|---|---|---|---|
| 1 | Cellocator Wireless Protocol | 4.007 | The document holds the complete OTA protocol (Fleet, CSA) and the CSA files structure |
| 2 | CSA Programming Manual | 2.0.0.3 | |
| 3 | Programming Manual Cellocator Cello | 31p | |
| 4 | Cello-IQ Evaluation Manual | 1.1 | |
| 5 | Cellocator Integration Package [Full Edition] v2.1 Manual | 2.1 | |
| 6 | Cello-IQ Product Overview | 2.4 | The document provides high-level information required by service providers regarding integration and operation of Cello-IQ devices with their fleet management applications.<br>The document describes the Cello-IQ content and deliverables. It also briefly describes the features and capabilities of driver behavior monitoring, eco-driving, and the driver safety system, as implemented in the Cello-IQ. |

## 1.5 Revision history

| Version | Date | Description |
|---|---|---|
| 1.0 | 14/01/09 | Original version. |
| 1.1 | 11/03/13 | Updated and edited. |

## 1.6 Abbreviations

| Abbreviation | Definition |
|---|---|
| ACK | Acknowledge |
| CCC | Command and Control Center |
| CSA | Car Safety Application |
| OTA | Over The Air |

# 2 General Description

## 2.1 Overview

Cello-IQ is a new Cellocator product designed to integrate the driver's safety and eco driving analysis into the legacy fleet management core.

Unlike legacy Cello devices, the Cello-IQ supports multiple IP concurrent sessions designed to carry fleet and CSA information to two different servers: one server is associated with the Fleet Management application while the other server manages CSA data. Apart from the two application oriented communication sessions, the Cello-IQ has a built in FTP/TFTP client designed to transfer CSA raw data files and crash reports (Emergency Data recording) to an FTP server.

The CSA uses the onboard accelerometer and GPS as sources for its driver behavior and ECO driving analysis algorithms. The CSA algorithm reports CSA events to the CSA server while maintaining trips and maneuver statistics in its nonvolatile memory for later upload. The CSA events are fully configurable, enabling the user to select the reporting level resolution.

The CSA mechanism is also used as an EDR (Emergency Data Recorder). There are two levels of accident detection and reporting supported (Light and heavy). Upon accident detection, the EDR part of the CSA mechanism sends an event to the CSA server and a file containing the accident time stamps and accelerations via FTP/TFTP to the FTP server.

Cello-IQ also has a Driver Feedback Device (DFD). The DFD is connected serially with the Cello-IQ unit and uses a set of visual icons and audible voice phrases to reflect CSA safety related driver events.

## 2.2 Network Architecture

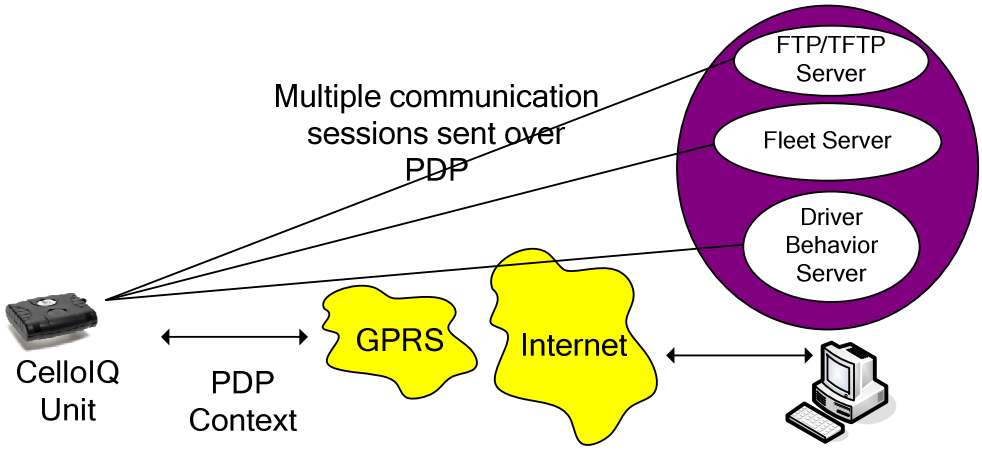


*Figure 1 - Cello-IQ Network Architecture*

Cello-IQ network architecture is built around multiple concurrent IP sessions with the server side. Unlike the single session used by legacy fleet management units, the Cello-IQ architecture supports multiple sessions designed to separate between the fleet management data interchange and the new CSA data interchange. The legacy fleet management protocol has higher priority over the CSA protocol since it handles the basic APN session maintenance and control.

The Cello-IQ also supports file transfer protocol to carry data files representing the vehicle's trips, maneuver statistics and crash data. The Cello-IQ unit has new configuration fields designed to set the new session properties (IP/DNS/port) and the FTP/TFTP client side properties.

The advantage in this multiple socket architecture is the great flexibility obtained in building the back office infrastructure and the possibility to work with distinct SW modules, in charge of a driver behavior management application, independent of the fleet management application. This architecture, as shown below, allows the seamless, transparent integration of a third party's DBM application into another TSP's or SW integrator's fleet management platform.

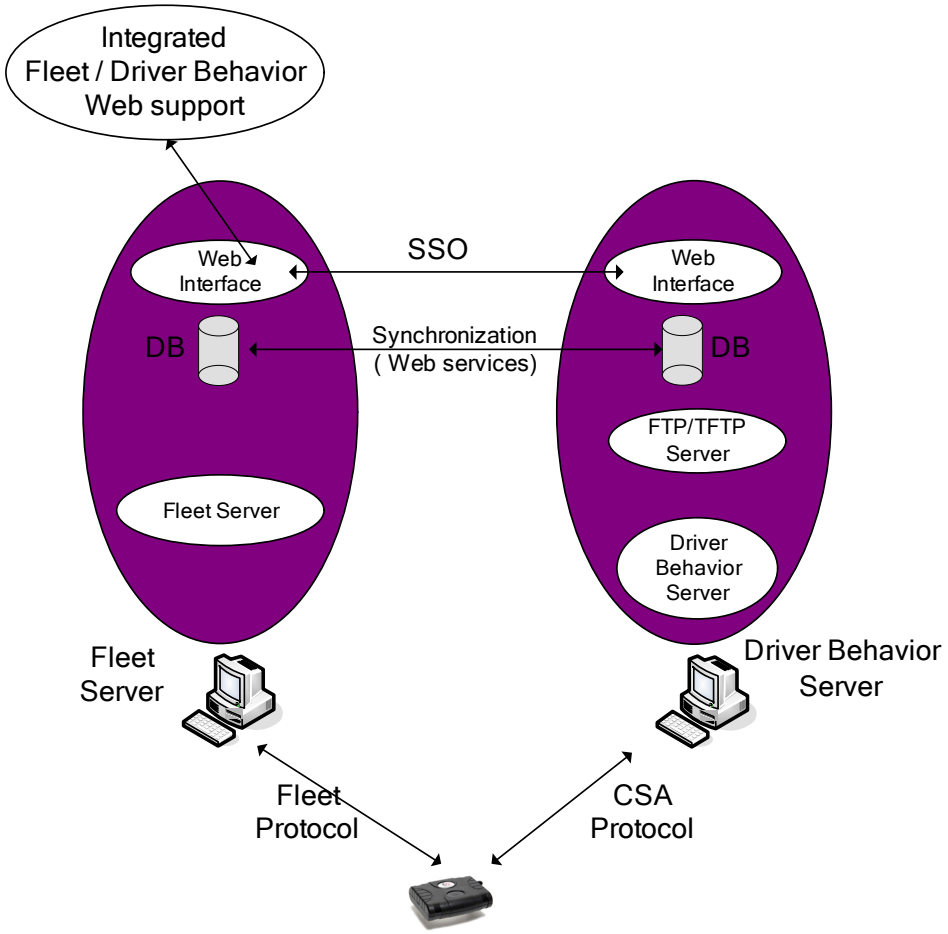


Figure 2 – SSO Network Architecture

**Note:** When the Driver behavior server is hosted by a remote server which is outside the private APN, a SIM with a public APN must be used.

## 2.3 Prerequisites

It is assumed the reader of this Manual is familiar with the following items:

- Cello-IQ Product Overview
- Cellocator Programming Manual
- Cellocator OTA Wireless Protocol
- CSA Programming Manual
- Cellocator Integration Package [Full Edition] Manual

We also assume the user has already setup an evaluation environment and demonstrated communication interchange between the evaluated Cello-IQ unit and the communication center provided by Cellocator.

This integration manual supports two integration procedures:

- Integrators developing their own application server or modifying their server to support the CSA protocol in addition to the existing fleet management protocol. This kind of integration requires full understanding of the CSA wireless protocol and requires server side implementation of CSA protocol message parsing. This document includes some embedded code examples. Please refer to the *Server Side CSA Integration* section for more details.
- Integrators using the Cellocator Integration Package which implements full CSA protocol integration with a database. Integrators using the Cellocator Integration Package will implement database level integration between their application layer and the units represented as database entries. Please refer to the *Cellocator Integration Package* section for more details.

## 2.4 Cello-IQ Integration Objectives

Cello-IQ is designed to communicate with a driving behavior management backend. It continuously sends maneuvers and event reports, event statistics and scores, and if requested, raw data for later analysis, processing and reconstruction.

The system reports a wide range of events and raw data concerned with hazardous or aggressive driving behavior, uneconomic and environmentally-unfriendly driving, and accident events as raw data for later reconstruction on the server side.

The information gathered on the server side can be used in a number of ways, including:

- To report risky events and to analyze the evolution and root cause of those events.
- To score driver behavior performance (Safety, ECO) over time.
- To build a driver profile (DNA) in order to understand weak performance aspects.
- To compare driver skills and behavior with other drivers in the fleet or within other populations.
- To evaluate a driver's risk level and Eco friendliness based on the above, and, as a consequence, to be able to take measures (training and education plan) in order to improve their driving skills and monitor improvement over time.

Cello-IQ is ready for integration with any TSP's SW platform with minimal development and integration effort as it has already analyzed and scored, on-board and in real-time, all maneuvers and trips. Valuable and compact information sets are now being uploaded to the backend.

The only mission left is to build a driver management portal that will allow efficient visibility of fleet performance indicators, exceptions and trends in order to allow an effective training and education plan execution.

This manual attempts to clarify and ease the process of turning Cello-IQ from an in-vehicle physical product into a complete DBM solution.

# 3 Server Side CSA Integration

## 3.1 Generic CSA Server Block Diagram

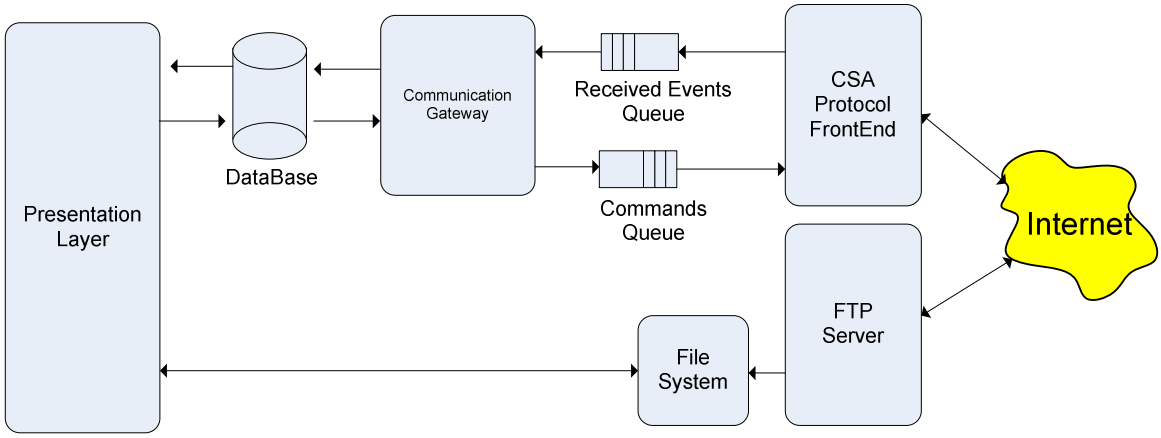


*Figure 3 - Typical CSA Server Building Blocks*

The block diagram describes a generic CSA server application intended to handle multiple Cello-IQ units. The CSA protocol frontend service handles multiple sockets connected with Cello-IQ units.

Cello-IQ units connected with the CSA server for the first time will initiate a CSA IPUP message to register their IP and unit ID in the server's database. CSA Datagrams received via open sockets are first checked to verify reception message integrity. If the integrity check passes, an acknowledgement CSA message will be sent towards the unit to confirm reception. Application messages are parsed, classified and then queued into the Received Events Queue. The Received Events Queue is read by the GW parser and sent from there to the database which holds the history of all CSA events. In the downstream path, actions originated by the presentation layer (or automatic DB routines) are written into the dedicated table in the database. The table update event is sensed by the GW and an event is queued towards the CSA protocol frontend. The CSA protocol frontend will build the protocol datagram and send it to the socket associated with the destination unit ID.

**Note**: It is assumed the fleet management server side is already implemented and running. The user must note that fleet session has a higher priority over the CSA session and it is mandatory to have an open session with the fleet server prior to the establishment of the CSA server.

## 3.2 CSA Uplink Protocol Structure and Data Flow

The CSA protocol is a modular protocol designed to interchange information elements between the Cello-IQ unit and the server application. The generic frame format supports variable length messages starting with fixed header and concatenated modules each holding a sub header defining the module type and length, followed by the module contents. The generic frame is terminated with a frame check sequence implemented as checksum.

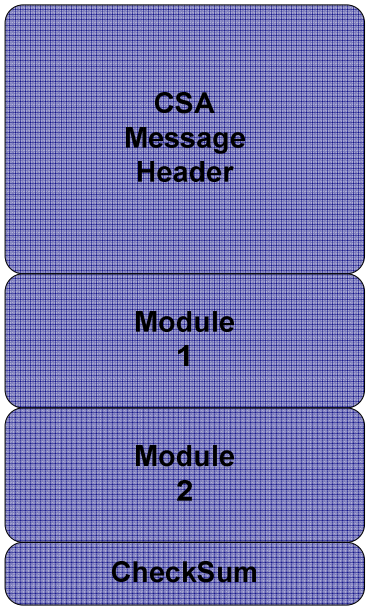


*Figure 4 - CSA Protocol Structure*

The CSA message header has the following format:

<table>
<tr><td>1</td><td colspan="8">System code, byte 1 – ASCII “C”</td></tr>
<tr><td>2</td><td colspan="8">System code, byte 2 – ASCII “S”</td></tr>
<tr><td>3</td><td colspan="8">System code, byte 3 – ASCII “A”</td></tr>
<tr><td>4</td><td colspan="8" rowspan="2">Length of message from byte 6 to CS</td></tr>
<tr><td>5</td></tr>
<tr><td>6</td><td colspan="8" rowspan="2">Message ID (Sequential numerator used by ACK mechanism. In replies contains the numerator of the command.)</td></tr>
<tr><td>7</td></tr>
<tr><td>8</td><td colspan="8">Message Type / Initiator / Direction (0 for CSA event)
<table>
<tr><td>Initiator<br>0 – reply or ACK<br>1 - active</td><td>Direction<br>0 – inbound<br>1 - outbound</td><td colspan="3">Protocol Version<br>1</td><td colspan="3">Message Type</td></tr>
<tr><td>Bit 7</td><td></td><td></td><td></td><td></td><td>Bit 2</td><td>Bit 1</td><td>Bit 0</td></tr>
</table></td></tr>
<tr><td>9</td><td colspan="8" rowspan="3">Cello Unit’s ID (total 32 bits)</td></tr>
<tr><td>10</td></tr>
<tr><td>11</td></tr>
</table>

| | |
|---|---|
| 12 | |

*Table 1 - CSA Header Format*

Parsing CSA messages starts with verifying the message prefix. The prefix for all CSA messages is the 3 ASCII characters "CSA". The parser mechanism then calculates the message checksum by summing the bytes following the length (bytes 4 and 5). The 8 bits sum is then compared with the last message byte holding the received checksum.

After the message integrity is validated the server starts the module level parsing. The detailed modules structure is part of the Cellocator OTA protocol. The GW parser should refer to the Protocol version bits in order to determine the expected modules and the actual structure of these modules, which may change with the evolution of the CSA protocol versions.

Modules are byte arrays in the following structure:

| | |
|---|---|
| | Module Name 30 (**CSA Full Event**) |
| | Length of module - 46 |
| | CSA Event Reason |
| | CSA Event Sub Reason |
| | CSA Event Numerator (or zero in case of reply) |
| | |
| | . |
| | . |
| | . |

*Table 2 - CSA Protocol Module Structure*

The first byte describes the modules type, for example "CSA FULL Event" and the byte following it defines the module length. CSA Full Event has event reason and sub reason bytes describing the event details; for example, IP Up Event reason is 1 and it does not carry a sub reason. CSA Full Event includes additional fields like GPS location and timestamp. For more details please refer to the *Cellocator OTA Protocol*.

## 3.3 Typical CSA Uplink Protocol Modules

The following table provides an overview of the main and most common modules, used by the CSA in order to report maneuvers, scoring and statistics according to the default PL configuration, provided by Cellocator:

| # | Module ID | Content and typical transmission use cases | Back-office typical use / processing |
|---|---|---|---|
| 1. | **Module 30** | **CSA full event** – This is the main report entity used by the CSA to report all sorts of CSA activities and triggering such as start / end of a trip, Ignition on or Off, IPUP, Go / Halt events and of course, any driving behavior event or maneuver detected and processed by the CSA. | Any driver behavior backend should be able to obtain, parse and present all the information provided by Module 30 in order to represent the various activities in time and behavior attributes performed by drivers and vehicles in the field. |
| 2. | **Module 31** | **ABC maneuver statistics** – This module, if enabled, is concatenated to Module 30 of short term maneuvers (Brakes, Turns, Accelerations, etc) and holds many maneuver attributes such as start and stop location of the maneuver, average and max values of speed and accelerations, duration of the maneuver, and so on. | Essential for map pinpointing of start and stop locations and provisioning of additional valuable information which may help in the presentation of the maneuver to the driver / fleet manager. |
| 3. | **Module 32** | **Trip statistics** – This module, if enabled, is sent upon the end of Trip (Ignition off / Driver change) and holds valuable trip related information like time driven, time idling, score of the trip, number of maneuvers of each type and severity, and so on. | This module is critical in order to profile a driver's efficiency and risk level over time through the trip score distance and time reported (refer to Section 5 for further info). Other provided attributes may be used in order to quickly identify trips which need further analysis on the back end due to certain exceptions, such as excessive idling time, severe over speeding, and so on. |
| 4. | **Module 35** | **Crash attributes** – Upon crash detection of a light or heavy crash, this module is generated and delivered through GPRS or SMS on top of an optional voice call triggering. This module holds Minimum Set of Data, crucial in determining via the control center the severity and type of the crash, the attributes and identity of the vehicle, the location of the crash, | This module should be analyzed in the backend in order to determine whether rescue forces should be sent to the location of the crash and to equip rescue forces with information that may help them serve the incident efficiently. |

| # | Module ID | Content and typical transmission use cases | Back-office typical use / processing |
|---|---|---|---|
| | | and so on. | |
| **5.** | **Modules 58-61** | **Continuous events statistics** – This module reports various attributes of long term maneuvers such as over speeding, wrong gear (RPM), idling and off-road sessions. Valuable information such as the start / end location of the event, its duration, its extreme values and averages, and of course, the score are reported. | Analysis and presentation layer of information pieces of these modules provides additional credibility, explanation and rationale to the event score which was assigned by the CSA logic and may help dramatically in the training and coaching process of drivers in the fleet. It may also be a source of information for independent scoring algorithm of the event which may be carried out on the server side. |

*Table 3 - Typical CSA Uplink Modules*

## 3.4 Typical CSA Activity and Reporting Scenarios

### 3.4.1 *IPUP*

After the Cello-IQ unit is restarted due to a reset, Power-Up or wake up from hibernation, the unit starts network registration and an IP is allocated for the unit. The unit opens a link towards the server and then an IPUP event is sent to associate the unit ID with the IP allocated to the unit by the network. The diagram below shows the message sequence between the unit and the server. The server's CSA front-end checks the datagram integrity and sends an acknowledge message toward the unit.

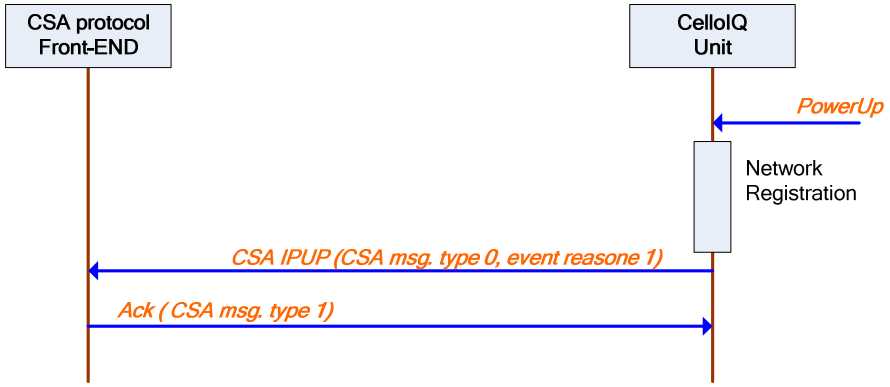


*Figure 5 - IPUP Information Flow*

### 3.4.2 *Calibration*

Cello-IQ units initially installed in cars are considered to be un-calibrated. The calibration process requires a few driving hours to converge. When converged, an event will be sent to the server to notify the server the unit is ready for driver monitoring. The server side should receive the message, check its reception integrity and acknowledge the unit. The server should change the unit's state to "Calibrated" in the server's database.

There are two calibration processes in the unit: the **orientation calibration**, which calculates and compensates the units' orientation (rotation angle, elevation angle etc.), and the **off-road calibration**, which adapt the unit's off-road thresholds in order to detect off-road events.

Calibration process monitoring is an essential process that should be carried out by the backend for two main reasons:

- Non-calibrated units do not generate maneuver reports and thus are only partially operative.
- Units which do not converge for a long period of time may indicate installation issues that must be addressed.

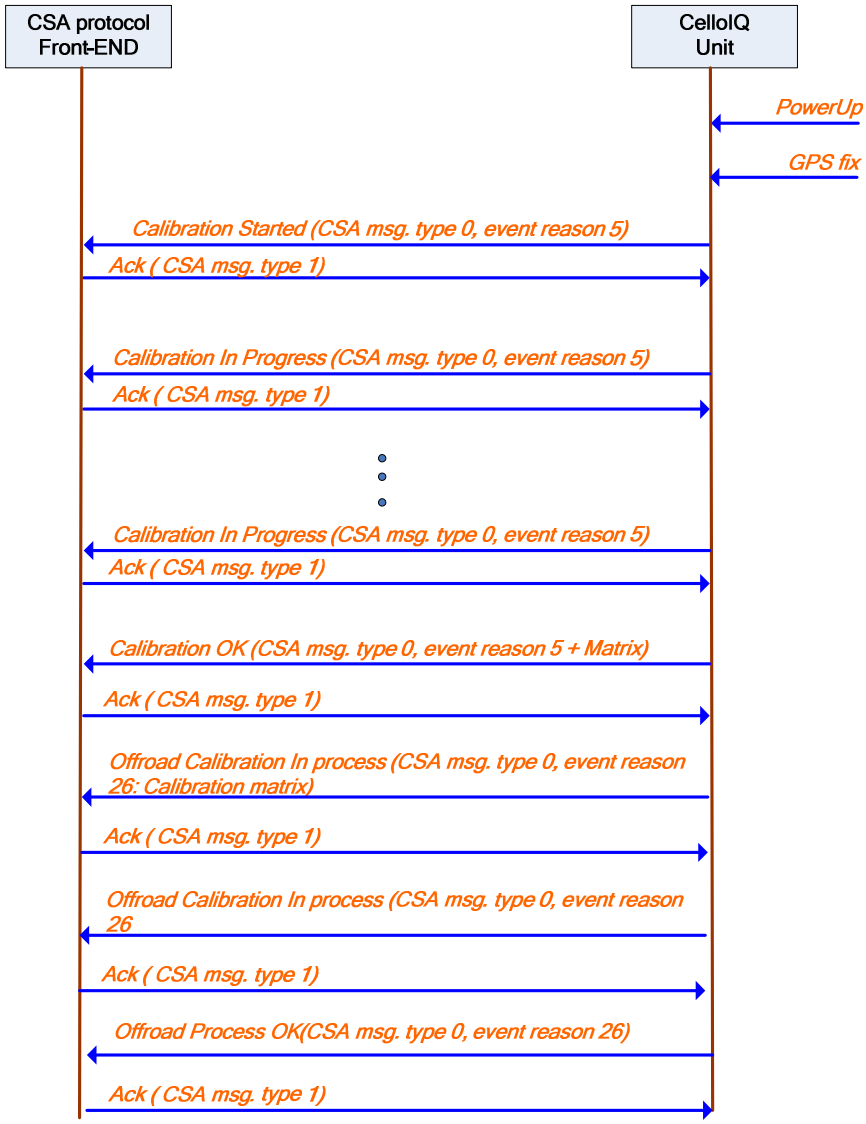


*Figure 6 - Calibration Information Exchange*

### 3.4.3 *Files Upload*

File uploading is an optional feature of the Cello-IQ unit. Files are usually uploaded to the server as they contain more detailed information about the trips accumulated by the unit. There are two types of files generated and uploaded by the CSA through the FTP channel:

- Crash data file for EDR (emergency data reconstruction)
- Raw data trip files

**Crash data files** are created when an accident is detected by the unit and the EDR feature is enabled. The unit continuously logs the acceleration and GPS stamps and when an accident is detected a higher resolution sampling starts until the buffer is filled. The accident "Story" captured is sent to the FTP server and a CSA event is sent.

**Raw data trip files** are sent at the end of the trip when ignition off is detected. The raw data files consist of maneuver accelerations and GPS stamps according to the severity requested by the user in the CSA PL. Maneuver raw data may be used by the backend in order to apply further / different scoring techniques or to playback an event in order to manage a training and education plan with a driver. The accumulated files are sent to the FTP server until all the files are uploaded or a time limit has been reached.

For more detailed information of the file structure please refer to the [1] Wireless communication Protocol *Files Structure* section.

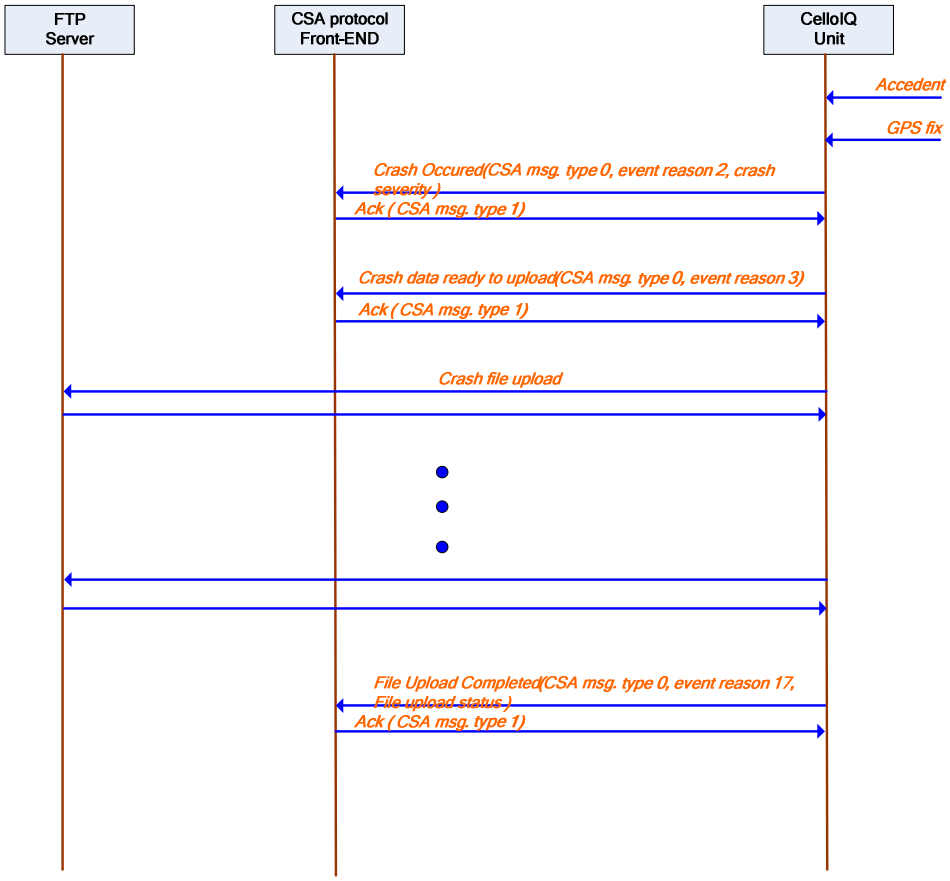


*Figure 7 - File Transfer to FTP Server*

## 3.5 Downlink Protocol Management Principles

### 3.5.1 *How to Manage CSA Configuration*

The CSA configuration memory is part of the fleet configuration memory space. The CSA programming parameters reflect the user's ability to control the driver behavior severity, user feedback, communication channel, and more.

The fleet configuration infrastructure enables the server side to upload and download the configuration memory using OTA, serial and SMS protocol. Apart from the fleet programming infrastructure, the CSA server can control the CSA programming memory using dedicated CSA commands, as described in the following tables.

**Programming Frame**

| | |
|---|---|
| | Module's ID (**10 - Programming Frame**) |
| | Length of module |
| | Programming command numerator |
| | |
| | • Action byte (Read/Write/Lock/Unlock)<br>• 0 for Read command<br>• 1 for Write command<br>• 2 for Lock command (an infrastructure - currently not used)<br>• 3 for Unlock command (an infrastructure - currently not used) |
| | The first address |
| | |
| | |
| | Length of data |
| | The data (in case of Read programming - single byte of Zero) |
| | |

*Table 4 - Programming Module Structure*

**Reply Programming Frame**

| | |
|---|---|
| | Module's ID 11-Reply Programming Frame |
| | Length of module |
| | Reply to Programming command numerator |
| | |
| | Status |

| Status (Success – 0, Failure – 1) | Failure ID | | | | | | |
|---|---|---|---|---|---|---|---|
| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |

| | |
|---|---|
| | Spare |
| | The first address |
| | |
| | |
| | Length of data (Zero - In all cases except Reply to Read Programming) |
| | The data (only in case of Reply to Read) |
| | |

*Table 5 - Reply Programming Module Structure*

Programming commands sent via CSA protocol will be saved into the CSA space of the nonvolatile configuration memory. The configuration will be effective as soon as the unit is restarted via the fleet reset OTA command. The same CSA protocol interface can request loading the CSA configuration for server side storage.

### 3.5.2 *How to Manage Preset Files*

Preset files are lower level configuration parameters, consisting of thresholds, weights, filter variables, etc. which are saved in a predefined location / range in the CSA memory and are used by the CSA algorithm throughout the analysis and scoring process of events, maneuvers and trips. These parameters are divided into four different sections representing different vehicle categories supported by the Cello-IQ. Upon installation, the right vehicle category file is used according to the setting in the PL.

Cellocator invested a significant amount of resources in order to tune the preset values to fit the supported vehicle categories and to deliver credible, qualitative and validated maneuver detection and scoring performance which does not need any user interference beyond manipulation of parameters available in the PL.

However, if you need to meet special vehicle / application environment conditions which enforce modification of these presets, they may be modified through our well-defined API which may be sent by Cellocator's support and engineering teams upon request.

### 3.5.3 *How to Manage Calibration Matrices*

As explained in section 3.4.2 there are 2 calibration processes in the Cello-IQ unit. The first calibration adapts the unit's orientation and the second, the off-road calibration, builds the criteria needed to detect off-road driving conditions. CSA protocol supports calibration matrix manipulation commands for exceptional situations where the calibration data is not correct or special debugging and evaluation procedures are required. Usually the calibration process might not converge if the unit installation is not firmly fixed to the car chassis. After the user corrects and verifies the installation is correct, a "Leave Calibration" command will be sent to the unit using CSA protocol to reinitiate the orientation and off-road calibration algorithms. The following tables show the CSA calibration command capabilities.

**Calibration/ Off-road Ready Mode**

This command initiates and finishes accelerometer Calibrating Mode as well as requesting Calibration Status. The reply to this command mandatorily includes Module 37 (Calibration Status).

| Byte number | Description | Content |
|---|---|---|
| N | Module ID-20 | |
| N+1 | Module length | 2 |
| N+2 | Option byte | 0 - Request Calibration Status<br>1 - Leave calibrated mode (forget calibration and off-road setup)<br>2 - Enter calibrated mode (use existing calibration matrix)<br>3 - Enter off-road ready mode (use existing off-road setup data) |
| N+3 | Spare | 0 |

*Table 6 - Calibration Management Command*

The above command has the ability to force calibration with predefined matrix downloaded via CSA command. The CSA commands supporting matrices upload and download are listed below:

| | | | | | |
|---|---|---|---|---|---|
| 37 | Calibration Status | | ☒ | | 0 |
| 42 | Calibration Matrix Set command | ☒ | | | 2 |
| 43 | Response to Calibration Matrix Set command | | ☒ | | 3 |
| 52 | Offroad Set command | ☒ | | | 2 |
| 53 | Response to Offroad Set command | | ☒ | | 3 |

*Table 7 - CSA Calibration Command Options*

Calibration status can be queried by the CSA command "**Calibration/ Off-road Ready Mode"** mentioned above. When the command is issued with the option byte set to 0, the unit will respond with Module 37. Module 37 is described below and reflects the current calibration status:

<table>
<tr><td></td><td>Module ID – 37 Calibration status</td></tr>
<tr><td></td><td>Module Length -3</td></tr>
<tr><td></td><td>Calibration<br>Not calibrated<br>Calibrated (Done)<br>Bad installations<br>255 - reserved</td></tr>
<tr><td></td><td>Calibration stage/step
<table>
<tr><td colspan="4">Phase number</td><td colspan="4">Step number</td></tr>
<tr><td>Bit 7</td><td>Bit 6</td><td>Bit 5</td><td>Bit 4</td><td>Bit 3</td><td>Bit2</td><td>Bit 1</td><td>Bit 0</td></tr>
</table></td></tr>
</table>

| | |
|---|---|
| | Off-road stage<br>0 – Not calibrated<br>1– calibrated: 100 samples done<br>2- calibrated: 1000 samples done |

*Table 8 - Calibration Status Report*

## 3.6 Decode CSA Structure

Units registered in the system can be accessed anytime by the server using their Unit ID.

This section provides Cello-IQ code samples that will enable you to structure and analyze the data communicated between the CSA server and the Cello-IQ units.

The **decode** samples are relevant for messages that come from the Cello-IQ unit and are sent to the CSA server. The **encode** and **encapsulate** samples are relevant for messages in the opposite direction (from the CSA server to the unit).

The code samples in this document are for reference purposes, and can be modified according to your requirements. Most samples are specific to a certain module within the message sent between the server and the unit.

The samples were created in C# / C++ and Microsoft Visual Studio .NET.

This code sample is relevant for the header of the message sent from the Cello-IQ unit to the CSA server.

The method shown in the sample below receives a byte array and looks for the CSA message structure. When found, it validates the checksum, creates a buffer of the CSA message and calls the `GenerateData` method, which generates the appropriate event.

### 3.6.1 *CSA Validation Code Sample*

The following code sample was developed in C#.

```
// define a cyclic buffer – accumulating the incoming data.
public CircularBuffer<byte> m_IncomingBuffer = new CircularBuffer<byte>(MAX_BUFFER);

/// <summary>
/// Decode a raw data to M2C messages, fires a OnData event.
/// </summary>
/// <param name="_Data"></param>
public void Decode(byte[] _Data)
{
    int iBufPos = 0;

    // loop until reaching the data end..
    while (iBufPos < _Data.Length)
    {
       // copy the incoming data to the accumulation buffer
```

```
        int iDataBlockSize = Math.Min(m_IncomingBuffer.Capacity / 2, _Data.Length -
iBufPos);
        byte[] DataBlock = new byte[iDataBlockSize];
        Array.Copy(_Data, iBufPos, DataBlock, 0, iDataBlockSize);
        m_IncomingBuffer.Put(DataBlock);

        iBufPos += iDataBlockSize;

        // define internal variables
        byte iData;
        byte iRemoveBytes;
        int m_BeforeChecksum = 0;

        // state machine looking for CSA message structure
        while (m_Offset < m_IncomingBuffer.Size)
        {
            iData = m_IncomingBuffer[m_Offset];
            iRemoveBytes = 0;

            switch (m_State)
            {
                case States.SearchC:
                    if (iData == 0x43)
                    {
                        m_StartPos = m_IncomingBuffer.Head;
                        m_State++;
                    }
                    Else
                        iRemoveBytes = 1;
                    break;
                case States.SearchS:
                    if (iData == 0x53)
                        m_State++;
                    else
                        iRemoveBytes = 1;
                    break;
                case States.SearchA:
                    if (iData == 0x41)
                        m_State++;
                    else
                        iRemoveBytes = 2;
                    break;
                case States.GetLengthByte1:
                    m_MessageLength = iData;
```

```
                    m_State++;
                    break;
                case States.GetLengthByte2:
                    m_MessageLength = (iData << 8) + m_MessageLength;
                    if (m_MessageLength - 5 > MAX_BUFFER)
                    {
                       LoggerClass.Instance.AddLine(LoggerClass.FLAG_PARSE_ERR, "Prob
lem, size is bigger then maximum size allowed!");
                        iRemoveBytes = 3;
                    }
                    else
                        m_State++;
                    break;
                case States.FindEnd:
                    if (m_Offset == m_MessageLength + 4)
                    {
                        m_State = States.ValidateChecksum;
                        m_BeforeChecksum = m_IncomingBuffer.Head;
                        if (ValidateChecksum(iData))
                        {
        // call a function that generate an event with the buffer found and cutted
                            GenerateData();
                            m_IncomingBuffer.Delete(m_Offset + 1);
                            m_State = States.SearchC;
                            m_Offset = -1;
                            m_StartPos = -2;
                            if (((m_Buffer[7] & 0x80) > 0)
&& (SettingsClass.Instance.SendImmediateAck))
            // call function to generate generic CSA ACK
                                GenerateAck();
                            else
                            {
            // Checksum failed!
                             byte[] CurrentBuffer = new byte[m_Offset + 1];
                              Array.Copy(m_IncomingBuffer.ToArray()
, 0, CurrentBuffer, 0, m_Offset + 1);
                              iRemoveBytes = 3;
                            }
                        }
                        else
                        {
                            if (m_Offset > m_MessageLength + 4)
                                iRemoveBytes = 3;
                        }
```

```
                    }
                    break;
            }  // switch

          // state machine state control – roll and delete data if structure doesn't
match CSA message.
            if (iRemoveBytes > 0)
            {
                m_IncomingBuffer.Delete(iRemoveBytes);
                m_State = States.SearchC;
                m_StartPos = -1;
                m_Offset = 0;
            }
            else
                m_Offset++;
        }  // while (m_Offset < m_IncomingBuffer.Size)

    } // while (iBufPos < _Data.Length)
} // Decode end

/// <summary>
/// Validate the checksum of CSA message
/// </summary>
/// <param name="_Checksum">Checksum from data (last byte)</param>
/// <returns>true if the checksum is the expected one</returns>
public bool ValidateChecksum(byte _Checksum)
{
    byte iChecksum = 0;
    m_Buffer = m_IncomingBuffer.ToArray(0, m_Offset + 1);

    for (int iPos = 3; iPos < m_Buffer.Length - 1; iPos++)
    {
        iChecksum += m_Buffer[iPos];
    }
    return iChecksum == _Checksum;
}

/// <summary>
/// Generates event data for CSA message and fires a OnData event.
/// </summary>
public void GenerateData()
{
    if (OnData != null)
    {
```

```
        DataInfo Info = new DataInfo(m_Buffer);
        OnData(Info, m_Buffer);
    }
}
```

### 3.6.2 *Decoding CSA Full Event Module (Module 30)*

The following code sample shows an object wrapping the **Full Event** module. It includes internal parameters, decoding function and the public variable for each parameter.

```
public class MMFullEvent : MMMessage
{
    // Define variable for each parameter of the module
    public byte m_EventReason;
    public byte m_EventSubReason;
    public UInt16 m_EventNumerator;
    public byte m_OperationMode;
    public byte m_Spare1;
    public Int64 m_DriverID; // 6bytes
    public Int32 m_TripID;  // 3 bytes
    public Int32 m_ManeuverID; // 3bytes
    public byte m_ManeuversDataUsage;
    public byte m_AccidentBufferStatusBitmask;
    public byte m_Spare2;
    public byte m_HDOP;
    public byte m_Mode1;
    public byte m_Mode2;
    public byte m_SatellitesUsed;
    public UInt32 m_Longitude;
    public UInt32 m_Latitude;
    public UInt32 m_Altitude;
    public byte m_GroundSpeed;
    public UInt16 m_SpeedDirection;
    public byte m_Seconds;
    public byte m_Minutes;
    public byte m_Hours;
    public byte m_Day;
    public byte m_Month;
    public byte m_Year;

    // Decode a byte array into the different fields of the module
    // The byte array is the module payload without the two first
```

```
    // bytes (module ID and module length)
    public  void Decode(byte[] _Data)
    {
      if (_Data.Length < 46)
                return;
      m_EventReason = _Data[0];
      m_EventSubReason = _Data[1];
      m_EventNumerator = BitConverter.ToUInt16(_Data, 2);
      m_OperationMode = _Data[4];
      m_Spare1 = _Data[5];
      m_DriverID = (Int64)((BitConverter.ToUInt64(_Data, 6) & 0x0000FFFFFFFFFFFF));
      m_TripID = (Int32)((BitConverter.ToUInt32(_Data, 12) & 0x00FFFFFF));
      m_ManeuverID = (Int32)((BitConverter.ToUInt32(_Data, 15) & 0x00FFFFFF));
      m_ManeuversDataUsage = _Data[18];
      m_AccidentBufferStatusBitmask = _Data[19];
      m_Spare2 = _Data[20];
      m_HDOP = _Data[21];
      m_Mode1 = _Data[22];
      m_Mode2 = _Data[23];
      m_SatellitesUsed = _Data[24];
      m_Longitude = BitConverter.ToUInt32(_Data, 25);
      m_Latitude = BitConverter.ToUInt32(_Data, 29);
      m_Altitude = BitConverter.ToUInt32(_Data, 33);
      m_GroundSpeed = _Data[37];
      m_SpeedDirection = BitConverter.ToUInt16(_Data, 38);
      m_Seconds = _Data[40];
      m_Minutes = _Data[41];
      m_Hours = _Data[42];
      m_Day = _Data[43];
      m_Month = _Data[44];
      m_Year = _Data[45];
    }
}
```

### 3.6.3 *Decoding CSA Maneuver Statistics (Module 31)*

The following code sample shows an object wrapping the **Maneuver Statistics** module. It includes internal parameters, decoding function and the public variable for each parameter.

```
public byte[] AddHeader(byte[] _Data)
{
    byte[] Result = new byte[_Data.Length + 2];
    Result[0] = ModuleNumber;
    Result[1] = (byte)_Data.Length;
    Array.Copy(_Data, 0, Result, 2, _Data.Length);
    return Result;
}

public class MMManeuverStatistics
{
    public Int32 m_TripID;  // 3 bytes
    public Int32 m_ManeuverID; // 3bytes
    public byte m_ManeuverType;
    public UInt64 m_StartLocation;
    public UInt64 m_EndLocation;
    public Int64 m_StartTime; // 7bytes
    public UInt16 m_ManeuverDuration;
    public UInt16 m_XAverage;
    public UInt16 m_YAverage;
    public UInt16 m_MaxX;
    public UInt16 m_MaxY;
    public UInt16 m_MaxZ;
    public byte m_SpeedAverage;
    public byte m_SpeedMax;
    public byte m_SpeedDelta;
    public UInt16 m_MaxRPM;
    public UInt16 m_MaxFuelFlow;
    public UInt16 m_FuelConsumed;
    public byte m_ABSState;
    public byte m_RiskScore;
    public byte m_NumOfInitFrames;

    public override void Decode(byte[] _Data)
    {
       if (_Data.Length < 54)
           return;
```

```
        m_TripID = _Data[0] | _Data[1] << 8 | _Data[2] << 16;
        m_ManeuverID = _Data[3] | _Data[4] << 8 | _Data[5] << 16;
        m_ManeuverType = _Data[6];
        m_StartLocation = BitConverter.ToUInt64(_Data, 7);
        m_EndLocation = BitConverter.ToUInt64(_Data, 15);
        m_StartTime = _Data[23] | _Data[24] << 8 | _Data[25] << 16 | _Data[26]
<< 24 | _Data[27] << 32 | _Data[28] << 40 | _Data[29] << 48;
        m_ManeuverDuration = BitConverter.ToUInt16(_Data, 30);
        m_XAverage = BitConverter.ToUInt16(_Data, 32);
        m_YAverage = BitConverter.ToUInt16(_Data, 34);
        m_MaxX = BitConverter.ToUInt16(_Data, 36);
        m_MaxY = BitConverter.ToUInt16(_Data, 38);
        m_MaxZ = BitConverter.ToUInt16(_Data, 40);
        m_SpeedAverage = _Data[42];
        m_SpeedMax = _Data[43];
        m_SpeedDelta = _Data[44];
        m_MaxRPM = BitConverter.ToUInt16(_Data, 45);
        m_MaxFuelFlow = BitConverter.ToUInt16(_Data, 47);
        m_FuelConsumed = BitConverter.ToUInt16(_Data, 49);
        m_ABSState = _Data[51];
        m_RiskScore = _Data[52];
        m_NumOfInitFrames = _Data[53];
    }
 }
```

### 3.6.4 *Decoding CSA File Structure (FTP)*

The FTP file structure is thoroughly described in the *Wireless Communication Manual* with relevant example files.

### 3.6.5 *Encoding CSA Request Maneuver/Trip Statistics (Module 15)*

The following code sample shows how to transmit data from the CSA server to the Cello-IQ unit. Specifically, it shows an object wrapping the **Request Maneuver/Trip Statistics** module, and includes public variables for each parameter and an encoding function which creates a byte array of the module according to the parameters.

```
public byte[] AddHeader(byte[] _Data)
{
    byte[] Result = new byte[_Data.Length + 2];
    Result[0] = ModuleNumber;
    Result[1] = (byte)_Data.Length;
    Array.Copy(_Data, 0, Result, 2, _Data.Length);
    return Result;
}

public class MMRequestStatistics
{
    public byte m_Spare;
    public Int32 m_TripID;  // 3 bytes
    public Int32 m_ManeuverID; // 3bytes

    // Generate a byte array with the module data:
    // id + length + parameters
    public override byte[] Encode()
    {
        Byte[] Result = { 0x15, 7
           m_Spare,
            (Byte)((m_TripID & 0xFF)),
           (Byte)((m_TripID & 0xFF00) >> 8),
           (Byte)((m_TripID & 0xFF0000) >> 16),
            (Byte)((m_ManeuverID & 0xFF)),
           (Byte)((m_ManeuverID & 0xFF00) >> 8),
           (Byte)((m_ManeuverID & 0xFF0000) >> 16
         )};

      return Result;
   }
}
```

### 3.6.6 *Encapsulate Module Data in CSA Message*

The following code sample shows how to add a header to the data/message that is sent from the CSA server to the Cello-IQ unit. Specifically, it shows how to encapsulate a CSA module (as generated by the module encode method in the previous code sample) into a full CSA message. The `PostData` method encapsulates the module data and sends an `OnSend` event with the CSA message.

```
public enum MsgType
{   CSAEvent_or_Reply = 0, Ack = 1, Programming_Inbound = 2,
    Programming_Outbound = 3, Command = 4 }

/// <summary>
/// Holds the outgoing numerator
/// </summary>
private UInt16 m_Numerator = 0;

private const byte PROTOCOL_VERSION = 0x08;     // protocol version 1 << 3

/// <summary>
/// Holds the unit ID (as byte array for easy insertion in the buffer)
/// </summary>
private byte[] m_UnitIDBuffer = new byte[4];

// <summary>
// Generates a CSA Message -
// when ready fires the OnSend event to post the CSA message to the socket.
/// The Post does not wait for an ACK!
/// </summary>
/// <param name="_Data">Payload part of the CSA message</param>
/// <param name="_MsgType">CSA Message type:</param>
public void PostData(byte[] _Data, MsgType _MsgType)
{
    if (OnSend != null)
    {
        int iDataLength = _Data.Length;
        byte[] SendBuffer = new byte[iDataLength + 13];
        // set CSA header
        SendBuffer[0] = 0x43;
        SendBuffer[1] = 0x53;
        SendBuffer[2] = 0x41;
        // set length
        SendBuffer[3] = (byte)(iDataLength + 8);
```

```
        SendBuffer[4] = (byte)((iDataLength + 8) >> 8);
        // set numerator
        SendBuffer[5] = (byte)m_Numerator;
        SendBuffer[6] = (byte)(m_Numerator >> 8);
        // set message type
         SendBuffer[7] = (byte)(0x80 | PROTOCOL_VERSION | (byte)_MsgType);

        // set unit ID
        Array.Copy(m_UnitIDBuffer, 0, SendBuffer, 8, 4);
        // set payload
        Array.Copy(_Data, 0, SendBuffer, 12, _Data.Length);
        // generate checksum
        byte iChecksum = 0;
        for (int iPos = 3; iPos < SendBuffer.Length - 1; iPos++)
        {
            iChecksum += SendBuffer[iPos];
        }
        SendBuffer[SendBuffer.Length - 1] = iChecksum;
        // fires event
        OnSend(SendBuffer, true, m_Numerator);
        m_Numerator++;
    }
}
```

# 4 Cellocator Integration Package

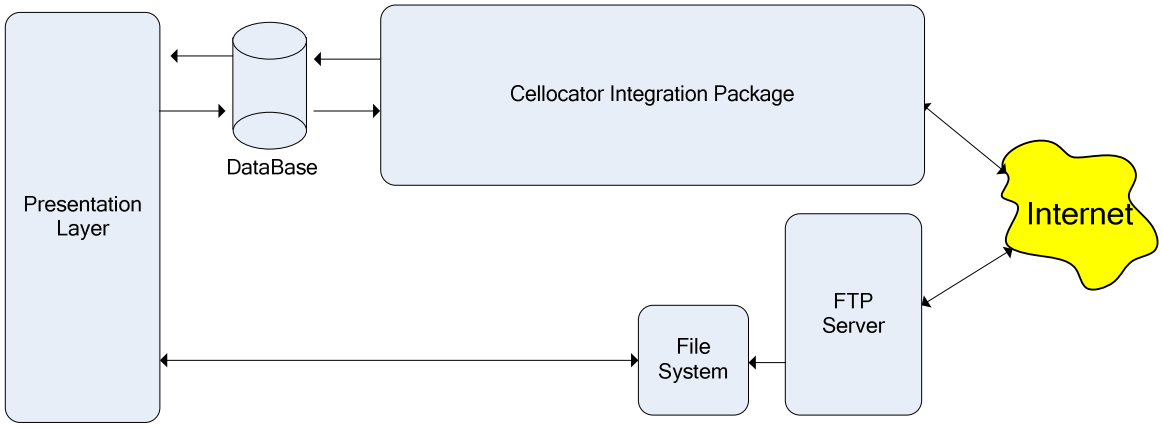


*Figure 8 - Cellocator Integration Package Roll*

The Cellocator Integration Package is a SW solution created by Pointer for Server side developers wishing to achieve database integration between their application layer and Cello family units. The Cellocator Integration Package defines the database tables' structure and commands needed to manage multiple Cello family units in a balanced, modular and expandable method.

The Package was updated to support CSA application on top of the legacy fleet management. For more detailed information about database level integration using Cellocator's Integration Package, please refer to *[5] Cellocator Integration Package [Full Edition] v2.0 Manual*.

Only in cases where Emergency data reconstruction (EDR) and/or backend maneuver analysis applications are implemented, should a standard FTP server be used in order to allow the automatic upload of raw data in addition to the CSA communication support of the integration package.

# 5 Application Layer Integration

## 5.1 Server Side Driver Ranking

As mentioned above, the Cello-IQ delivers on-board processing via maneuver and trip scoring logics. However, for the driver scoring process, the integrator should take into account the relative distance (recommended) / time driven by a specific driver, comparing to another driver or a certain population of drivers (group, the whole fleet, the whole monitored population, etc) within the same vehicle category.

This weighting process is essential in order to reflect the frequency of a driver's incorrect behavior over traveled distance or over time. Otherwise, short trips with bad scoring will have the same impact on a driver's total score as a very long trip with an excellent score.

The following guidelines are recommended for the calculation of a driver's relative score, which also takes into account the distance driven within a given population:

1. All calculations should be done on a given period of time and a given vehicle category (Private/LCV/MCV/HCV).
2. In order to comply with the law of large numbers, driver scoring should be presented only after approximately 30-50 trips were registered and stored for a driver.

   Before such a prerequisite is fulfilled, the statistics which may be provided to the user are only partially accurate and are expected to change as additional data is aggregated.
3. In order to calculate the distance-weighted average trip-score of a specific driver use a simple weighted average formula:

   $$\widetilde{S_m} = \frac{\sum_{i=0}^{n} D_i * S_i}{\sum_{i=0}^{n} D_i}$$

   Where D is the distance of a trip and S is the score of the trip.
4. Perform the same calculation for the reference population or a driver with which you want to compare, for the same period of time and the same vehicle category.
5. An optional ranking presentation may be to project the final relative driver's score as a division of the accumulated mileage during the compared period and the weighted score or:

   $$R_D = \frac{(\sum_{i=0}^{n} D_i)^2}{\sum_{i=0}^{n} D_i * S_i}$$

   Where $R_D$ is a relative, distance-compensated driver ranking.

Another way to compare between two drivers or driver groups is to base the occurrence rate of unsafe or dangerous maneuvers as a function of traveled distance or trip time. Such comparisons may provide an additional value angle of comparison and may be weighted with the safety score ranking. The backend logic may assign negative or positive score points to a driver or a group of drivers as a function of the frequency of incoming dangerous maneuvers compared to the average frequency of all groups in scope.

**Note:** The profile of traveled roads via different driver populations may significantly influence the driver's weighted score and the rate of driving events occurrence. For example, an urban delivery truck which spends most of the day on urban roads cannot be compared evenly with an urban truck which drives a significant portion of its mileage on non-urban roads, for example, from the delivery center and back. The integrator should take this into consideration when implementing driver scoring logic on the server side.

## 5.2 Server Side Over-Speeding Monitoring

For fleet safety operations in which the TSP wishes to implement backend over-speeding events management based on the speeding profile provided by Cello-IQ together with a cross-reference with road-specific speed limit values obtained from a GIS database, the driving behavior backend should alter the speeding-free trip scores received from the Cello-IQ (Module 32 in CSA protocol) in order to combine it with the over-speeding event scores generated by the server side.

The following is the proposed logic to be executed on the backend upon the "end of trip", whenever the speeding profile is enabled in the PL (and thus speeding events are not taken into calculation in the trip safety score):

In order to be able to implement the following logic, Module ID 34 (Trip log) in the CSA protocol should be enabled.

$M = A\ weight\ Multiplier\ per\ sevirity\ (green, yellow, red - From\ PL)$

$R = Risk\ weight\ per\ type\ of\ Maneuver\ (From\ PL)$

$S_i = Score\ for\ trip\ i\ (From\ Modle\ 32)$

$V_{sj} = Score\ for\ speeding\ maneuver\ j\ in\ the\ corresponding\ trip\ (Backend\ calculations)$

$X_{ts} = \#\ of\ maneuvers\ of\ type\ t\ and\ severity\ s\ processed\ during\ the\ corresponding\ trip\ (From\ 34)$

$\widetilde{S_i} = Overspeeding\ weighted\ score\ per\ trip\ i$

$$\widetilde{S_i} = \frac{S_i * \sum_{t=1}^{6}\sum_{s=1}^{3} X_{ts} * M(s) * R(t) + \sum_{j=1}^{m} V_{sj} * R_{Speeding} * M(V_{sj})}{\sum_{t=1}^{6}\sum_{s=1}^{3} X_{ts} * M(s) * R(t) + \sum_{j=1}^{m} R_{Speeding} * M(V_{sj})}$$

**Note**: In the equation above, *t* goes from 1 to 6, representing 6 different maneuver types (acceleration, braking, cornering, lane departure, accelerations + turn, and brake + turn) affecting the risk score, excluding over-speeding.