Garmin Fleet Management Interface Control Specification

June 10, 2014
Drawing Number: 001-00096-00 Rev. M

Limitation of Warranties and Liability:

Garmin International, Inc. and its affiliates make no warranties, whether express, implied or statutory, to companies or individuals accessing Garmin Fleet Management Interface Control Specification, or any other person, with respect to the Garmin Fleet Management Interface Control Specification, including, without limitation, any warranties of merchantability or fitness for a particular purpose, or arising from course of performance or trade usage, all of which are hereby excluded and disclaimed by Garmin. Garmin does not warrant that the Specification will meet the requirements of potential Fleet Management customers, that the Specification will work for all mobile platforms that potential customer may desire, or that the Specification is error free.

Garmin International, Inc. and its affiliates shall not be liable for any indirect, incidental, consequential, punitive or special damages for any cause of action, whether in contract, tort or otherwise, even if Garmin International, Inc. has been advised of the possibility of such damages.

Warning:

All companies and individuals accessing the Garmin Fleet Management Interface Control Specification are advised to ensure the correctness of their Device software and to avoid the use of undocumented features, particularly with respect to packet ID, command ID, and packet data content. Any software implementation errors or use of undocumented features, whether intentional or not, may result in damage to and/or unsafe operation of the device.

Garmin's Fleet Management solutions:
Please visit Garmin's Fleet Management web page at: http://www.garmin.com/solutions/

Garmin's Fleet Management Technical Support or Feedback:
For technical support or to provide feedback please email: **FleetSupport@garmin.com**

1 Introduction

1.1 Overview

This document describes the Garmin Fleet Management Interface, which is used to communicate with a Garmin device for the purpose of Fleet Management / Enterprise Tracking applications. The Device Interface supports bi-directional transfer of data. In the sections below, detailed descriptions of the interface protocols and data types are given.

Note: It is highly recommended to perform initial prototype testing with the Garmin "Fleet Management Controller" tool (also known as the "FMC" or "PC App"), which simulates Server connectivity when connected to a Garmin FMI Client device. This would allow a developer to observe format and sequence of protocol packet exchanges outlined in this document. This free tool can be found in the Fleet Management Interface Developer Kit at: http://developer.garmin.com/lbs/fleet-management/

1.2 Definition of Terms

- **Client** – Refers to a Garmin-produced device that supports fleet management.
- **Server** – Refers to the device communicating with the Garmin-produced device.

1.3 Serialization of Data

Every data type must be serialized into a stream of bytes to be transferred over a serial data link. Serialization of each data type is accomplished by transmitting the bytes in the order that they would occur in memory given a machine with the following characteristics:

1. Data structure members are stored in memory in the same order as they appear in the type definition.
2. All structures are packed, meaning that there are no unused "pad" bytes between structure members.
3. Multi-byte numeric types are stored in memory using little-endian format, meaning the least-significant byte occurs first in memory followed by increasingly significant bytes in successive memory locations.

1.4 Data Types

The following table contains data types that are used to construct more complex data types in this document. The order of the members in the data matches the order shown in structures in this document and there is no padding not expressed in this document.

Data Type	Description
char	8 bits in size and its value is an ASCII character
uchar_t8	This represents a single byte in the UTF-8 variable length encoding for Unicode characters and is backwards compatible with ASCII characters. The contents of an uchar_t8 array will always be ASCII characters unless both the Server and Client support Unicode. If both the Server and Client support Unicode, then the contents of an uchar_t8 array can have UTF-8 encoded characters. Please see Section 5.1.4 for more information.
uint8	8-bit unsigned integers
uint16	16-bit unsigned integers
uint32	32-bit unsigned integers
sint16	16-bit signed integers
sint32	32-bit signed integers
float32	32-bit IEEE-format floating point data (1 sign bit, 8 exponent bits, and 23 mantissa bits)
float64	64-bit IEEE-format floating point data (1 sign bit, 11 exponent bits, and 52 mantissa bits)
boolean	8-bit integer used to indicate true (non-zero) or false (zero).

time_type	The *time_type* is used in some data structures to indicate an absolute time. It is an unsigned 32-bit integer and its value is the number of seconds since 12:00 am December 31, 1989 UTC. A hex value of 0xFFFFFFFF represents an invalid time, and the Client will ignore the time.

sc_position_type	The *sc_position_type* is used to indicate latitude and longitude in semicircles, where 2^{31} semicircles equal 180 degrees. North latitudes and East longitudes are indicated with positive numbers; South latitudes and West longitudes are indicated with negative numbers. All positions are given in WGS-84.
	
	`typedef struct`
	`{`
	`sint32 lat; /* latitude in semicircles */`
	`sint32 lon; /* longitude in semicircles */`
	`} sc_position_type;`
	
	The following formulas show how to convert between degrees and semicircles:
	
	degrees = semicircles * (180 / 2^{31})
	semicircles = degrees * (2^{31} / 180)

double_position_type	The *double_position_type* is used to indicate latitude and longitude in radians. North latitudes and East longitudes are indicated with positive numbers; South latitudes and West longitudes are indicated with negative numbers. All positions are given in WGS-84.
	
	`typedef struct`
	`{`
	`float64 lat; /* latitude in radians */`
	`float64 lon; /* longitude in radians */`
	`} double_position_type;`
	
	The following formulas show how to convert between degrees and radians:
	
	degrees = radians * (180 / pi)
	radians = degrees * (pi / 180)

map_symbol	An enumeration that specifies a map symbol. It is an unsigned 16-bit integer. For possible values, see the Garmin Device Interface Specification at http://developer.garmin.com/web-device/device-sdk/.

2 Protocol Layers

The protocols used in the fleet management interface control are arranged in the following layers:

Protocol Layer	
Application	Highest
Physical/Link	Lowest

The Physical layer is based on RS-232. The link layer uses packets with minimal overhead. At the Application layer, there are several protocols used to implement data transfers between a Client and a Server. These protocols are described in more detail later in this document.

3 Physical/Link Protocol

3.1 Serial Protocol

The Serial protocol is RS-232. Other electrical characteristics are full duplex, serial data, 9600 baud, 8 data bits, no parity bits, and 1 stop bit.

3.1.1 Serial Packet Format

All data is transferred in byte-oriented packets. A packet contains a three-byte header (DLE, ID, and Size), followed by a variable number of data bytes, and followed by a three-byte trailer (Checksum, DLE, and ETX). The following table shows the format of a packet:

Byte Number	Byte Description	Notes
0	Data Link Escape	ASCII DLE character (16 decimal)
1	Packet ID	identifies the type of packet (See Appendix 6.1)
2	Size of Application Payload	number of bytes of packet data (bytes 3 to n-4)
3 to n-4	Application Payload	0 to 255 bytes
n-3	Checksum	2's complement of the sum of all bytes from byte 1 to byte n-4 (end of the payload)
n-2	Data Link Escape	ASCII DLE character (16 decimal)
n-1	End of Text	ASCII ETX character (3 decimal)

3.1.2 DLE Stuffing

If any byte in the Size, Packet Data, or Checksum fields is equal to DLE, then a second DLE is inserted immediately following the byte. This extra DLE is not included in the size or checksum calculation. This procedure allows the DLE character to be used to delimit the boundaries of a packet.

3.1.3 ACK/NAK Handshaking

Unless otherwise noted in this document, a device that receives a data packet must send an ACK or NAK packet to the transmitting device to indicate whether the data packet was successfully received. Normally, the transmitting device does not send any additional packets until an ACK or NAK is received (this is sometimes referred to as a "stop and wait" or "blocking" protocol). The following table shows the format of an ACK/NAK packet:

Byte Number	Byte Description	Notes
0	Data Link Escape	ASCII DLE character (16 decimal)
1	Packet ID	ASCII ACK/NAK character (6 or 21 decimal respectively) (See Appendix 6.1)
2	Size of Packet Data	2
3	Packet Data	Packet ID of the acknowledged packet.
4	NULL	0
5	Checksum	2's complement of the sum of all bytes from byte 1 to byte 4
6	Data Link Escape	ASCII DLE character (16 decimal)
7	End of Text	ASCII ETX character (3 decimal)

The ACK packet has a Packet ID equal to 6 decimal (the ASCII ACK character), while the NAK packet has a Packet ID equal to 21 decimal (the ASCII NAK character). Both ACK and NAK packets contain an 8-bit integer in their packet data to indicate the Packet ID of the acknowledged packet.

If an ACK packet is received, the data packet was received correctly and communication may continue. If a NAK packet is received, the data packet was not received correctly and should be sent again. NAKs are used only to indicate errors in the communications link, not errors in any higher-layer protocol.

4 Overview of Application Protocols

4.1 Packet Sequences

Each of the Application protocols is defined in terms of a packet sequence, which defines the order and types of packets exchanged between two devices, including direction of the packet, Packet ID, and packet data type. An example of a packet sequence is shown below:

N	Direction	Packet ID	Packet Data Type
0	Server to Client	First_Packet_ID	First_Data_Type
1	Server to Client	Second_Packet_ID	Second_Data_Type
2	Server to Client	Third_Packet_ID	Third_Data_Type
3	Client to Server	Fourth_Packet_ID	Fourth_Data_Type
4	Client to Server	Fifth_Packet_ID	Fifth_Data_Type

In this example, there are five packets exchanged: three from Server to Client and two from the Client to the Server. Each of these five packets must be acknowledged, but the acknowledgement packets are omitted from the table for clarity.

The first column of the table shows the packet number (used only for reference; this number is not encoded into the packet). The second column shows the direction of each packet transfer. The third column shows the Packet ID value. The last column shows the Packet Data Type.

4.2 Undocumented Application Packets

The Client may transmit application packets containing packet IDs that are not documented in this specification. These packets are used for internal testing purposes by Garmin engineering. Their contents are subject to change at any time and should not be used by third-party applications for any purpose. They should be handled according to the physical/link protocols described in this specification and then discarded.

5 Application Protocols

5.1 Fleet Management Protocols

Note: It is highly recommended to perform initial prototype testing with the Garmin "Fleet Management Controller" tool (also known as the "FMC" or "PC App"), which simulates Server connectivity when connected to a Garmin FMI Client device. This would allow a developer to observe format and sequence of protocol packet exchanges outlined in this document. This free tool can be found in the Fleet Management Interface Developer Kit at: http://developer.garmin.com/lbs/fleet-management/

5.1.1 Protocol Identifier

All packets related to the fleet management protocols will use the packet ID 161. The first 16 bits of data in the application payload will identify the fleet management protocol's specified packet. The remaining data in the application payload will be the fleet management payload. The fleet management data types discussed in this document will not include the fleet management packet ID in their structures. The fleet management packet ID is implied. The following table shows the format of a fleet management packet:

Byte Number	Byte Description	Notes
0	Data Link Escape	16 (decimal)
1	Packet ID	161 (decimal)
2	Size of Packet Data	Size of fleet management Payload + 2
3-4	Fleet Management Packet ID	See Appendix 6.2
5 to n-4	Fleet Management Payload	0 to 253 bytes
n-3	Checksum	2's complement of the sum of all bytes from byte 1 to byte n-4
n-2	Data Link Escape	16 (decimal)
n-1	End of Text	3 (decimal)

5.1.2 Enable Fleet Management Protocol

By default, a Garmin device that supports fleet management will have the fleet management protocol disabled until it receives the following packet from the Server. Only Clients that report A607 as part of their protocol support will support the associated data type.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0000 – Enable Fleet Protocol Request	fleet_features_data_type

Previous versions of this protocol had no associated data and it remains acceptable to provide no data with this packet. In this case, the features will be set to their previous states or default states, as appropriate. The type definition for *fleet_features_data_type* is shown below.

```
typedef struct  /* D607 */
    {
    uint8                   feature_count;
    uint8                   reserved; /* Set to 0 */
    uint16                  features[];
    } fleet_features_data_type;
```

The *feature_count* indicates the number of items in the features array. The *features* array contains an array of up to 126 feature IDs and whether or not the Server supports them. The lower 15 bits of each uint16 contains the *Feature ID* and the high bit of these indicates whether the feature is supported or not. A value of 1 indicates that the feature should be enabled and a 0 indicates that the feature should be disabled.

Note: If a feature is not specified on or off, it will take the value it had at the previous enable. If there was no previous enable, there is a default value for each feature. The following table shows the ID for each available feature and the feature's default value.

Feature ID (decimal)	Feature Name	Default value
1	Unicode support	Enabled
2	A607 Support	Disabled*
10	Driver passwords	Disabled
11	Multiple drivers	Disabled
12	AOBRD	Disabled**

* A607 support will be enabled automatically if either the driver passwords feature or the multiple drivers feature is enabled, as A607 support is required for these features.

** AOBRD functionality is the optional HOS FMI protocol defined in Section 6.5, as A610 (HOS 1.0), A615 (HOS 2.0) and A619 (HOS 2.1) support.

An example packet that enables the multiple driver and driver password features follows:

Byte Number	Byte Description	Notes
0	DLE	16 (decimal)
1	Packet ID	161 (decimal)
2	Size of Enable Fleet Request	8 (Represents bytes 3 to 10 for this example)
3-4	Fleet Management Packet ID	0 (decimal)
5	feature_count	2 (decimal)
6	reserved	0 (decimal)
7-8	Driver passwords (Enabled)	0x800A (hexadecimal)
9-10	Multiple drivers (Disabled)	0x000B (hexadecimal)
11	Checksum	2's complement of the sum of all bytes from byte 1 to byte 10
12	DLE	16 (decimal)
13	ETX	3 (decimal)

The Server is required to enable the Fleet Management interface if any of the following conditions occurs:

- The cable connecting the Server to the Client is disconnected and then reconnected
- Whenever the Client powers on
- Whenever the Server powers on

Note: Any attempt to access the fleet management protocols will be ignored by the Client until the Fleet Management interface is enabled. It is possible that the Client will not be listening to the communication line, so the following procedure is recommended:

1. Server sends the Enable Fleet Protocol Request packet to the Client until an ACK packet is received back from the Client.
2. Server sends the Product ID Request packet to the Client.
3. The Fleet Management Interface is not successfully enabled until the Product ID and Protocol Support Data packets are received from the Client.

Note: Repeat steps 1 and 2 until step 3 is successful.

Sending the enable fleet management protocol request to a Garmin device that supports fleet management will cause the following to occur on the Garmin device:

1. The Garmin device user interface will change so that the user can now access fleet management options on the device like text messages and Stop list. This change will be permanent across power cycles so that the user will have a consistent experience with the device.
2. The Garmin device will start to send PVT (position, velocity, and time) packets to the Server. PVT packets will be discussed later in Section 5.2.4 of this document.

Note: See Section 5.1.14 for information on disabling the fleet management interface after it has been enabled.

5.1.3 Product ID and Support Protocol

The Product ID and support protocol is used to query the Client to find out its Product ID, software version number, and supported protocols and data types. The packet sequence for the Product ID and support protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0001 – Product ID Request	None
1	Client to Server	0x0002 – Product ID	product_id_data_type

2	Client to Server	0x0003 – Protocol Support Data	protocol_array_data_type

The type definition for the *product_id_data_type* is shown below.

```
typedef struct
    {
    uint16                  product_id;
    sint16                  software_version;
    } product_id_data_type;
```

The *product_id* is a unique number given to each type of Garmin device. It should not be confused with an ESN, which is unique to each device regardless of type. The *software_version* is the software version number multiplied by 100 (e.g. version 3.11 will be indicated by 311 decimal).

The type definition for the *protocol_support_data_type* is shown below. The *protocol_array_data_type* is an array of the *protocol_support_data_type*. The number of *protocol_support_data_type* contained in the array is determined by observing the size of the received packet data.

```
typedef struct
    {
    char                    tag;
    sint16                  data;
    } protocol_support_data_type;
```

The *tag* member can have different values. The A tag describes an "application" protocol. For example, if a Client reports a *tag* with an "A" and data of 602 (A602), then it supports some of the fleet management protocol defined in this document. Each section in this document that describes a protocol will state its support A tag.

The D (data type) tags that are listed immediately after the A (application protocol) tag describe the specific data types used by that application protocol. This allows for future growth of both the data types and the protocol. For example, if a Client reports a *tag* with a "D" and data of 602, then it supports some of the fleet management data types defined in this document. Each section in this document that describes a data type will state its support D tag.

The Server is expected to communicate with the Client using the Client's stated application protocol and data types. In this way, it is possible to have different generations of products in the field and still have a workable system.

5.1.4 Unicode Support Protocol

This protocol is used by the Client to determine if the Server supports Unicode characters or just regular ASCII characters. This protocol is only supported on Clients that report A604 as part of their protocol support data. The packet sequence for the Unicode support protocol is shown below.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Client to Server	0x0004 – Unicode Support Request Packet ID	None
1	Server to Client	0x0005 – Unicode Support Response Packet ID	None

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0005 – Unicode Support Packet ID	None

The Server should respond to the Unicode support request to indicate that it supports Unicode and it is okay for the Client to send Unicode texts to it. If the Server does not respond to the Unicode support request, the Client will assume the Server does not support Unicode and it is not okay to send Unicode text to the Server.

Clients that don't report A604 as part of their protocol support data do not support Unicode and the Server should not send Unicode texts to them since they will be interpreted as ASCII text.

5.1.5 Text Message Protocols

5.1.5.1 Server to Client Text Message Protocols

There are numerous protocols available to the Server for sending text messages to the Client. The Server should pick a text message protocol based on the type of functionality it is trying to achieve on the Client. The different types of Server to Client text message protocols are described in detail below.

5.1.5.1.1 A604 Server to Client Open Text Message Protocol

This text message protocol is used to send a simple text message from the Server to the Client. When the Client receives this message, it either displays the message immediately, or presents a notification that a message was received, depending on the options specified in the message. The receipt indicates whether the message was accepted by the device; it does not imply that the message has been displayed.

Note: The Canned Message protocol can be used to give the driver a list of response choices by sending a matching id member value. See the Server to Client Canned Response Text Message Protocol Section 5.1.5.1.3 for details.

The packet sequence for the Server to Client open text message is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x002a – A604 Server to Client Open Text Message Packet ID	A604_server_to_client_open_text_msg_data_type
1	Client to Server	0x002b – Server to Client Open Text Message Receipt Packet ID	server_to_client_text_msg_receipt_data_type

The type definition for the *A604_server_to_client_open_text_msg_data_type* is shown below.

```
typedef struct  /* D604 */
    {
    time_type   origination_time;
    uint8       id_size;
    uint8       message_type;
    uint16      reserved;    /* set to 0 */
    uint8       id[/* 16 bytes */];
    uchar_t8    text_message[/* variable length, null-terminated string, 200 bytes max */];
    } A604_server_to_client_open_text_msg_data_type;
```

The *origination_time* is the time that the text message was sent from the Server. The *id_size* determines the number of characters used in the id member. An id size of zero indicates that there is no message id. The *id* member is an array of 8-bit integers that could represent any type of data. A message ID is required to use the Message Status Protocol (see Section 5.1.5.2). The *message_type* indicates how the message should be handled on the Client device. The allowed values for *message_type* are:

Value (Decimal)	Behavior
0	Add message to inbox, and display a floating button indicating that a message was received. (This is same behavior used for the A602 and A603 Server to Client text messages).
1	Display the message on the device immediately.

The type definition for the *open_text_msg_receipt_data_type* is shown below.

```
typedef struct  /* D604 */
    {
    time_type   origination_time;
    uint8       id_size;
    boolean     result_code;
    uint16      reserved;    /* set to 0 */
    uint8       id[/* 16 bytes */];
    } server_to_client_text_msg_receipt_data_type;
```

The *origination_time*, *id_size*, *message_type*, and *id* will be the same as the corresponding A604 Server to Client Open Text Message packet. The *result_code* indicates whether the message was received and stored on the Client device; it will be true if the message was accepted, or false if an error occurred (for example, there is already a message with the same ID).

5.1.5.1.2 A611 Server to Client Long Text Message Protocol

The Long Text Message Protocol allows the Server to send text messages up to 2,000 bytes in length to the Client. Text messages greater than 200 bytes will need to be segmented into additional packets by the Server. These packets will be reassembled and stored by the Client as a single text message. The Server should send an entire text message before attempting to send a different text message (i.e., no interleaving of segments for multiple text messages).

The Client will acknowledge each segment using an error *result_code*, and send a final receipt to indicate the message was accepted by the device (this does not imply that the message has been displayed).

Note: The Canned Message protocol can be used to give the driver a list of response choices by sending a matching id member value. See the Server to Client Canned Response Text Message Protocol Section 5.1.5.1.3 for details.

The *origination_time* field is the time the text message was sent from the Server. The *id_size* field indicates the number of characters of the *id* field. The *message_type* determines if the text message is displayed immediately or if a notification icon is displayed. Either display mode occurs after the complete text message is received.

The *finished_flag* field indicates the last segment of a long text message. The *sequence_number* field starts with a zero value for the first segment of each text message, and increments for each additional packet segment of that text message. Every packet segment of a long text message is required to have the same *id* field, which differentiates each complete text message from all other text messages.

Each packet segment is expected to fill the entire 200 byte *text_message* field with non-NULL data, except the last packet, which terminates the entire data set with a NULL terminator character. The current maximum complete Long Text Message size is 2,000 bytes (which does include the NULL terminator character).

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0..n-1	Server to Client	0x0055 – A611 Server to Client Long Text Message Packet ID	A611_server_to_client_long_text_msg_data_type
1..n	Client to Server	0x0056 – Server to Client Long Text Message Receipt Packet ID	server_to_client_long_text_msg_receipt_data_type

The type definition for the A611*_server_to_client_long_text_msg_data_type* is shown below.

```
typedef struct  /* D611 */
    {
    time_type   origination_time;    /* Time sent from Server */
    uint8       id_size;             /* Number of characters used in id member */
    uint8       message_type ;       /* 0 = Notify with icon, 1 = Display immediately */
    boolean     finished_flag;       /* 0 = Indicates additional packet segment, */
                                     /* 1 = Indicates final packet segment */
```

```
    uint8       sequence_number;     /* Indicates packet segment number, 0 = First packet */
    uint8       id[ 16 ];            /* id_size, 16 bytes */
    uchar_t8    text_message[ 200 ]; /* 200 bytes, or less variable length in final segment */
    } A611_server_to_client_long_text_msg_data_type;
```

The type definition for the *server_to_client_long_text_msg_receipt_data_type* is shown below.

```
typedef struct  /* D611 */
    {
    time_type   origination_time;    /* Original time sent from Server */
    uint8       id_size;             /* Number of characters used in the id member */
    uint8       result_code;         /* 0 = Message accepted, non-zero = Error occurred, */
                                     /* see code list below */
    boolean     finished_flag;       /* 0 = Indicates additional packet, 1 = Final packet */
    uint8       sequence_number;     /* Indicates packet segment number of a long message, */
                                     /* 0 = First packet segment */
    uint8       id[ 16 ];            /* id_size, 16 bytes */
    } server_to_client_long_text_msg_receipt_data_type;
```

Note: The protocol should not continue if the result_code is nonzero.

The values for *result_code* are listed below:

Value (Decimal)	Result Code
0	No Error
1	Invalid ID size Error
2	Non-Zero Sequence Number Error
3	Storage Error 0
4	ID In Use Error 1
5	Storage Error 1
6	Wrong ID size Error
7	Wrong ID Error
8	Wrong Sequence Number Error
9	Wrong Origination Time Error
10	Wrong Message Type
11	Exceeded 2,000 bytes of text
12	Storage Error 2
13	Storage Error 3
14	Storage Error 4
15	Storage Error 5

5.1.5.1.3 Server to Client Canned Response Text Message Protocol

This text message protocol is used to send a text message from the Server to the Client which requires a response to be selected from a list.

Note: The A604 Server to Client Open Text Message protocol described in Section 5.1.5.1.1, and A611 Server to Client Long Text Message Protocol described in Section 5.1.5.1.2 can utilize this protocol capability.

When the text message is displayed, the Client will also display a Reply button. When the Reply button is pressed, the Client will display a list of the allowed responses. Once the user selects one of the responses, the Client will send an acknowledgement message to the Server indicating which response was selected.

Prior to using this protocol, the Server must send the text for allowed responses to the Client using the Canned Response List Protocols described in Section 5.1.5.4.

The packet sequence for the *Server to Client Canned Response Text Message* protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0028 –Canned Response List Packet ID	canned_response_list_data_type
1	Client to Server	0x0029 – Canned Response List Receipt Packet ID	canned_response_list_receipt_data_type
2	Server to Client	0x002a –A604 Server to Client Open Text Message Packet ID or 0x0055 –A611 Server to Client Long Text Message Packet ID	A604_server_to_client_open_text_msg_data_type A611_server_to_client_long_text_msg_data_type
3	Client to Server	0x002b – Client to Server Open Text Message Receipt Packet ID or 0x0056 – Client to Server Long Text Message Receipt Packet ID	server_to_client_text_msg_receipt_data_type server_to_client_long_text_msg_receipt_data_type
4	Client to Server	0x0020 – Text Message Acknowledgment Packet ID	text_msg_ack_data_type
5	Server to Client	0x002c – Text Message Ack Receipt Packet ID	text_msg_id_data_type

The type definition for the *canned_response_list_data_type* is shown below.

```
typedef struct  /*  D604/D611 */
    {
    uint8       id_size;
    uint8       response_count;
    uint16      reserved; /* set to 0 */
    uint8       id[/* 16 bytes */];
    uint32      response_id[];
    } canned_response_list_data_type;
```

The *id_size* and *id* are used to correlate the canned response list to the A604 Server to Client Open Text Message or A611 Server to Client Long Text Message that follows; the *id* must be unique for each message sent to the device, and cannot have a length of zero. The *response_id* array contains the IDs for up to 50 canned responses that the user may select from when responding to the message. The *response_count* indicates the number of items in the *response_id* array.

The type definition for the *canned_response_list_receipt_data_type* is shown below.

```
typedef struct  /*  D604/D611 */
    {
    uint8       id_size;
    uint8       result_code;
    uint16      reserved; /* set to 0 */
    uint8       id[/* 16 bytes */];
    } canned_response_list_receipt_data_type;
```

The *id_size* and *id* will be identical to those received using the Canned Response List packet. A *result_code* of zero indicates success, a nonzero *result_code* means that an error occurred, according to the table below.

Note: The protocol should not continue if the result_code is nonzero.

Result Code (Decimal)	Meaning	Suggested Response
0	Success	Send the text message packet.
1	Invalid response_count	Send a Canned Response List packet containing between 1 and 50 response_id entries.
2	Invalid response_id	Use the Set Canned Response protocol (Section 5.1.5.4.1) to ensure all of the canned responses are on the

		Client, then resend the Canned Response List packet.
3	Invalid or duplicate message ID	Send a Canned Response List packet using a message ID not on the Client.

The A604 Server to Client Open Text Message is described in Section 5.1.5.1.1, the A611 Server to Client Long Text Message is described in Section 5.1.5.1.2.

The type definition for the *text_msg_ack_data_type* is shown below.

```
typedef struct  /* D602/D604/D611 */
    {
    time_type   origination_time;
    uint8       id_size;
    uint8       reserved[3];       /* set to 0 */
    uint8       id[/* 16 bytes */];
    uint32      msg_ack_type;
    } text_msg_ack_data_type;
```

The *origination_time* is the time that the text message was acknowledged on the Client. The *id_size* and *id* will match the *id_size* and id of the applicable text message. The *msg_ack_type* will identify the *response_id* corresponding to the response selected by the user.

The type definition for the *text_msg_id_data_type* is shown below.

```
typedef struct  /* D604/D611 */
    {
    uint8       id_size;
    uint8       reserved[3];        /* set to 0 */
    uint8       id[/* 16 bytes */];
    } text_msg_id_data_type;
```

The *id_size* and *id* will match the *id_size* and *id* of the applicable text message.

5.1.5.2 Message Status Protocol

The Message Status Protocol is used to notify the Server of the status of a text message previously sent from the Server to the Client. The Client will send a message status packet whenever the status changes. If the protocol is throttled (see Section 5.1.17), the Client will only send the message status of a text message when it is requested by the Server.

The packet sequence for the *Client to Server Open Text Message* is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0040 – Message Status Request Packet ID	message_status_request_data_type
1	Client to Server	0x0041 – Message Status Packet ID	message_status_data_type

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Client to Server	0x0041 – Message Status Packet ID	message_status_data_type

The type definition for the *message_status_request_data_type* is shown below.

```
typedef struct  /*  D604  */
    {
    uint8 id_size;
    uint8 reserved[3];        /* set to 0 */
    uint8 id[/* 16 bytes */];
    } message_status_request_data_type;
```

The *id_size* and *id* correspond to those of the message being queried; all must match for the message to be found.

The type definition for the *message_status_data_type* is shown below.

```
typedef struct  /*  D604  */
    {
    uint8 id_size;
    uint8 status_code;
    uint16 reserved;          /* set to 0 */
    uint8 id[/* 16 bytes */];
    } message_status_data_type;
```

The *id_size* and *id* will be identical to those of the corresponding Message Status Request. The *status_code* indicates the status of the message on the device. The following table shows the possible values for *status_code*, along with the meaning of each value:

Status Code (Decimal)	Meaning
0	Message is unread
1	Message is read
2	Message is not found (e.g., deleted)

5.1.5.3 Message Delete Protocol

This protocol allows the Server to delete text messages stored on the Client. The packet sequence for deleting a text message is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x002D – Delete Text Message Packet ID	message_delete_data_type
1	Client to Server	0x002E – Delete Text Message Response Packet ID	message_delete_response_data_type

The type definition for *message_delete_data_type* is shown below.

```
typedef struct  /* D607 */
    {
    uint8       id_size;
    uint8       reserved[3];        /* set to 0 */
    uint8       id[/* 16 bytes */];
    } message_delete_data_type;
```

The *id_size* and *id* are used to specify the *id* of the message to be deleted.

The type definition for the *message_delete_response_data_type* is shown below.

```
typedef struct  /* D607 */
    {
    uint8       id_size;
    boolean     status_code;
    uint16      reserved;           /* set to 0 */
    uint8       id[/* 16 bytes */];
    } message_delete_response_data_type;
```

The *id_size* and *id* confirm the *id* of the message deleted. The *status_code* is false if the message was found but could not be deleted and true otherwise.

5.1.5.4 Canned Response List Protocols

These protocols are used to maintain the list of canned responses used in the Server to Client Canned Response Text Message Protocol (Section 5.1.5.1.3).

Up to 200 canned responses may be stored on the Client, and up to 50 of these responses may be specified as allowed for each text message. Canned responses are stored permanently across power cycles.

5.1.5.4.1 Set Canned Response Protocol

This protocol is used to set (add or update) a response in the canned response list.

The packet sequence for the Set Canned Response Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0030 – Set Canned Response Text Packet ID	canned_response_data_type
1	Client to Server	0x0032 – Set Canned Response Receipt Packet ID	canned_response_receipt_data_type

The type definition for the *canned_response_data_type* is described below.

```
typedef struct  /*  D604  */
    {
    uint32   response_id;
    uchar_t8 response_text[];    /* variable length, null terminated, 50 bytes max */
    } canned_response_data_type;
```

The *response_id* is a number identifying this response. If a response with the specified *response_id* already exists on the device, the response text will be replaced with the *response_text* in this packet.

The type definition for the *canned_response_receipt_data_type* is shown below.

```
typedef struct  /*  D604  */
    {
    uint32   response_id;
    boolean  result_code;
    uint8    reserved[3];               /* Set to 0 */
    } canned_response_receipt_data_type;
```

The *response_id* will be the same as that of the corresponding *canned_response_data_type*. The *result_code* indicates whether the add/update operation was successful.

5.1.5.4.2 Delete Canned Response Protocol

This protocol is used to remove a canned response text from the Client device. When a canned response is deleted, it is also removed from the list of allowed responses for any canned response text messages that the Client has not replied to. If all allowed responses are removed from a message, the message is treated as a Server to Client Open Text Message.

The packet sequence for the Delete Canned Response Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0031 – Delete Canned Response Packet ID	canned_response_delete_data_type
1	Client to Server	0x0033 – Delete Canned Response Receipt Packet ID	canned_response_receipt_data_type

The type definition for the *canned_response_delete_data_type* is shown below.

```
typedef struct  /*  D604  */
    {
    uint32   response_id;
    } canned_response_delete_data_type;
```

The *response_id* identifies the response to be deleted.

The type definition for the *canned_response_receipt_data_type* is described in Section 5.1.5.4.1 and repeated below.

```
typedef struct  /*  D604  */
    {
    uint32   response_id;
    boolean  result_code;
    uint8    reserved[3];                /* Set to 0 */
    } canned_response_receipt_data_type;
```

The *response_id* will be the same as that of the corresponding *canned_response_delete_data_type*. The *result_code* indicates whether the delete operation was successful. This will be *true* if the response was removed or the response did not exist prior to the operation; it will be *false* if the canned response cannot be removed.

5.1.5.4.3 Refresh Canned Response Text Protocol

This protocol is initiated by the Client to request updated response text for a particular message, or for all messages.

This protocol is only supported on Clients that report A604 as part of their protocol support data, and is throttled by default on Clients that report A605 as part of their protocol support data. See the Message Throttling Protocols (Section 5.1.17) to enable the Refresh Canned Response Text Protocol on these Clients.

The packet sequence for the Refresh Canned Response Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Client to Server	0x0034 – Request Response Text Refresh Packet ID	request_canned_response_list_refresh_data_type
1..n		(Set Canned Response protocols)	

The *request_canned_response_list_refresh_data_type* is shown below.

```
typedef struct  /*  D604  */
    {
    uint32 response_count;
    uint32 response_id[];
    } request_canned_response_list_refresh_data_type;
```

The *response_count* indicates the number of responses that are in the *response_id* array. This will always be between 0 and 50. The *response_id* array contains the response IDs that should be sent by the Server using the Set Canned Response protocol.

If *response_count* is zero, the Server should initiate a Set Canned Response protocol for all valid response IDs. If *response_count* is greater than zero, the Server should initiate a Set Canned Response protocol for each response ID in the array, so long as the response ID is still valid.

5.1.5.5 A607 Client to Server Open Text Message Protocol

This text message protocol is used to send a simple text message from the Client to the Server. When the Server receives this message, it is required to send a message receipt back to the Client. The Client will only send this protocol if the Server has enabled A607 features in the enable protocol. The packet sequence for the Server to Client open text message is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Client to Server	0x0026 – A607 Client to Server Open Text Message Packet ID	a607_client_to_server_open_text_msg_data_type
1	Server to Client	0x0025 – Client to Server Text Message Receipt Packet ID	client_to_server_text_msg_receipt_data_type

The type definition for the *message_link_data_type* is shown below.

```
typedef struct  /* D607 */
    {
    time_type         origination_time;
    sc_position_type  scposn;
    uint32            unique_id;
    uint8             id_size;
    uint8             reserved[3];        /* set to 0 */
    uint8             id[/* 16 bytes */];
    uchar_t8          text_message[/* variable length, null-terminated string, 200 bytes max */];
    } a607_client_to_server_open_text_msg_data_type;
```

The *origination_time* is the time that the text message was sent from the Client. The *scposn* is the semi-circle position at the time the message was created. If the Client did not have a GPS signal at the time the message was created, both the *lat* and *lon* will be 0x80000000. The *unique_id* is the unsigned 32-bit unique identifier for the message. The *id* is the ID of the Server to Client text message that this text message is responding to, if any. The *id_size* is the size of the *id*. If the text message is not in response to any Server to Client message or no ID is available, *id_size* will be 0 and the contents of *id* will be all 0.

5.1.5.6 Canned Message (Quick Message) List Protocols

The canned message list maintenance protocols are used to maintain the list of canned (predefined) text messages that a Client device may send to the Server using the Quick Message feature. When sending a canned message, the user of the Client device has the option to modify the text before sending it. The message is sent from the Client to the Server using the A607 **Client to Server** Open Text Message Protocol described in Section 5.1.5.5.

Up to 120 canned messages may be stored on the Client. Canned messages are stored permanently across power cycles.

5.1.5.6.1 Set Canned Message Protocol

The Set Canned Message Protocol is used to add or update the text of a canned message on the Client.

The packet sequence for the Set Canned Message Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0050 – Set Canned Message Packet ID	canned_message_data_type
1	Client to Server	0x0051 – Set Canned Message Receipt Packet ID	canned_message_receipt_data_type

The type definition for the *canned_message_data_type* is shown below.

```
typedef struct  /*  D604  */
    {
    uint32     message_id;      /* message identifier  */
    uchar_t8   message[];       /* message text, null terminated (50 bytes max) */
    } canned_message_data_type;
```

The *message_id* is a number identifying this message. The *message_id* is not displayed on the Client, but it is used to control the order in which the messages are displayed: messages are sorted in ascending order by *id*. If a message with the specified *message_id* already exists on the device, it will be replaced with the *message* text in this packet; otherwise, the message will be added.

The type definition for the *canned_message_receipt_data_type* is shown below.

```
typedef struct  /*  D604  */
    {
    uint32  message_id;                 /* message identifier */
    boolean result_code;                /* result (true if successful, false otherwise) */
    uint8   reserved[3];                /* Set to 0 */
    } canned_message_receipt_data_type;
```

The *message_id* is the same number in the corresponding *canned_message_data_type*. The *result_code* indicates whether the add/update operation was successful.

5.1.5.6.2 Delete Canned Message Protocol

The Delete Canned Message Protocol is used to delete a canned message from the Client.

The packet sequence for the Delete Canned Message Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0052 – Delete Canned Message Packet ID	canned_message_delete_data_type
1	Client to Server	0x0053 – Delete Canned Message Receipt Packet ID	canned_message_receipt_data_type

The type definition for the *canned_message_delete_data_type* is shown below.

```
typedef struct  /*  D604  */
    {
    uint32 message_id;          /* message identifier  */
    } canned_message_delete_data_type;
```

The *message_id* is a number identifying the message to be deleted.

The type definition for the *canned_message_receipt_data_type* is shown below.

```
typedef struct  /*  D604  */
    {
    uint32  message_id;                /* message identifier */
    boolean result_code;               /* result (true if successful, false otherwise) */
    uint8   reserved[3];               /* Set to 0 */
    } canned_message_receipt_data_type;
```

The *message_id* is the same number in the corresponding *canned_message_delete_data_type*. The *result_code* indicates whether the delete operation was successful; this will be true if the specified message was successfully deleted or was not on the device.

5.1.5.6.3 Refresh Canned Message List Protocol

The Refresh Canned Message List Protocol is initiated by the Client when it requires an updated list of canned messages. In response to this packet, the Server shall initiate a Set Canned Message protocol for each message that should be on the Client.

This protocol is only supported on Clients that report A604 as part of their protocol support data, and is throttled by default on Clients that report A605 as part of their protocol support data. See the Message Throttling Protocols (Section 5.1.17) to enable the Refresh Canned Message List Protocol on these Clients.

The packet sequence for the Refresh Canned Message Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Client to Server	0x0054 – Refresh Canned Message List	None

		Packet ID	
1..n		(Set Canned Message protocols)	

5.1.5.7 Other Text Message Protocols (Deprecated)

Other Text Message protocols are described in Section 6.6.1. These protocols are deprecated, and could exist now, but may be removed in the near future.

5.1.6 Stop (Destination) Protocols

The Stop protocols are used to inform the Client of a new destination. When the Client receives a Stop from the Server, it displays the Stop to the user and gives the user the ability to start navigating to the Stop location. There are four protocols available to the Server for sending Stops to the Client: the A603 Stop protocol, the A614 Path Specific Stop protocol, the A618 Stop protocol and the (deprecated) A602 Stop protocol.

The A602 Stop protocol does not have the Stop status reporting capability, but all other Stop protocols report the status of a Stop back to the Server. If a Client supports both the A603 and A602 Stop protocols, it is recommended that the Server uses the A603 Stop protocol to send Stops to the Client.

The A614 Path Specific Stop protocol allows the Server to provide a series of locations to shape the route to the Stop. Like the A603 Stop protocol, client will report the status of a Stop back to the Server.

The A618 Stop protocol provides the same functionality as the A603Stop protocol, and also allows the Stop Text to extend to 2,000 bytes, which is sent to the Client using the File Transfer protocol.

Stop	Stop ID	Text bytes	Stop Status	Route Shaping	File Transfer	Non-Compressed Format	Compressed Format	Hyperlink
A603	Y	200	Y	N	N	Y	N	5.1.6.1
A614	Y	200	Y	Y	Y	N	Y	5.1.6.2
A618	Y	2,000	Y	N	Y	Y	Y	5.1.6.3
*A602	N	51	N	N	N	Y	N	6.6.4.1

*A602 Stop protocol currently deprecated

5.1.6.1 A603 Stop Protocol

This protocol is used to send Stops or destinations from the Server to the Client. When the Client receives a Stop, it will display it to the user and give the user the ability to start navigating to the Stop location. The Client will report Stop status (unread, read, active…) for Stops it received using this protocol. The packet sequence for the A603 Stop protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0101 – A603 Stop Protocol Packet ID	A603_stop_data_type

The type definition for the *A603_stop_data_type* is shown below.

```
typedef struct  /* D603 */
    {
    time_type         origination_time;
    sc_position_type  stop_position;
    uint32            unique_id;
    uchar_t8          text[/* variable length, null-terminated string, 200 bytes max */];
    } A603_stop_data_type;
```

The *origination_time* is the time that the Stop was sent from the Server. The *stop_position* is the location of the Stop. The *unique_id* contains a 32-bit unique identifier for the Stop. The *text* member contains the text that will be displayed on the Client's user interface for this Stop.

If a stop with the same *unique_id* already exists on the device, the *origination_time*, *stop_position* and *text* will be updated. If the active stop is updated, the Client will recalculate the route to the new location. Updating a stop does not change the status; the Server must use the Stop Status Protocol described in Section 5.1.7 to change the status.

5.1.6.2 A614 Path Specific Stop (PSS) Protocol

The Path Specific Stop protocol allows the Server to create a detailed route to the Stop by using shaping and intermediate destination points based on latitude and longitude coordinates. The Path Specific Stop data is sent to the Client in the form of a file.

5.1.6.2.1 Path Specific Stop (PSS) file data format

Path Specific Stop (PSS) file format rules are as follows:

1. The file must start with PSS header data.
2. The file header must contain Stop Text.
3. A Destination Point type (not a Shaping Point) must always immediately follow the header data.
4. Following the initial Destination Point, a series of either Shaping Point or more Destination Point types can be used.
5. The last point type must always be a Destination Point type (not a Shaping Point).

A Path Specific Stop file contains two sections, a **header section**, and a **Destination Point/Shaping Point section**. These data sections are shown below:

Header section:

Data Type	Name	Description
uint32	Signature	('P', 'S', 'R', '+') Hard-coded value to indicate the file type
uint16	Format version	0 = File format version_0
uint32	Timestamp	Timestamp from Server
uint32	Unique ID number	Unique ID number for this Stop assigned by the Server
uchar_t8	Stop Text	Stop Name shown on device->Dispatch->My Stops (up to 200 bytes)

NOTE: The Stop Text is a NULL terminated string, and the 200 byte max field size includes the NULL terminator. Following the NULL terminator, the next data type immediately follows (so these strings do not occupy the complete maximum length unless the string actually requires that space).

The header must always be followed by a Destination Point type. Any other point type (e.g., Shaping Point) will generate an error response. This initial point type (a Destination Point) is shown below, and immediately follows the header data:

(Initial) Destination Point format:

sint32	Latitude	Latitude of Destination point type (Destination must be first point)
sint32	Longitude	Longitude of Destination point type (Destination must be first point)
uint8	Point Type	0 = Destination (Destination must be the first point type)
uchar_t8	Destination Name	Destination Name shown on device navigation turn-by-turn text list (up to 40 bytes including a NULL terminator, see note below)

NOTE: The Destination Name is a NULL terminated string, and the 40 bytes max field size includes a NULL terminator. Following the NULL terminator, the next data type immediately follows (so these strings do not occupy the complete maximum length unless the string actually requires that space). If a Desination Name is not needed, a single NULL terminator must be used to indicate this condition.

Next, a series of Destination Points <u>and</u> Shaping Points can be appended to the initial Destination Point data if needed, and a Final Destination point must end the file.

The format of a Shaping Point entry is shown below:

Shaping Point format:
Note: The maximum Shaping Point count between Destination Points = 100

sint32	Latitude	Latitude of Destination point type (Destination must be first point)
sint32	Longitude	Longitude of Destination point type (Destination must be first point)
uint8	Point Type	1 = Shaping (Shaping point cannot be the end Stop point type)

The format of a Destination Point entry is shown below:

Destination Point (or Final Destination Point) format:
Note: The maximum Destination Point count = 25

sint32	Latitude	Latitude of Destination point type (Destination must be first point)
sint32	Longitude	Longitude of Destination point type (Destination must be first point)
uint8	Point Type	0 = Destination (Can be an intermediate or final point)
uchar_t8	Destination Name	Destination Name shown on device navigation turn-by-turn text list (up to 40 bytes including a NULL terminator, see note below)

This Destination Point format is the same format as an initial Destination Point shown earlier.
NOTE: The Destination Name is a NULL terminated string, and the 40 bytes max field size includes a NULL terminator. Following the NULL terminator, the next data type immediately follows (so these strings do not occupy the complete maximum length unless the string actually requires that space). If a Desination Name is not needed, a single NULL terminator must be used to indicate this condition.

5.1.6.2.2 Path Specific Stop (PSS) send to Client Protocol

This A614 protocol allows the Server to send a Path Specific Stop to the Client. Path Specific Stop files are transferred from the Server to the Client with the same file transfer mechanism used to send GPI, AOBRD driver log downloads, Custom Form Templates, and A618 Stop files to the Client (a Garmin device). A Path Specific Stop file can be sent to the Client only when no other files are being transferred (being sent) to the Client. Additional information regarding the file transfer process can be found in Section 5.1.13.1.

Any error detected during a download will halt the *Path Specific Stop* file transfer process, and that file will not be saved by the Client. Once the Path Specific Stop file is downloaded to the Client, the Stop file will be analyzed for format errors. Two distinct levels of result codes will be sent back to the Server (**one code to indicate the success of the file download process** and **one code to indicate the file format/content examination results**) in the *File End Receipt Packet* of Section 5.1.13.1.6.4.

N	Direction	Fleet Management Packet ID	Hyper-Link	Fleet Management Packet Data Type
0	Server to Client	0x0400 – File Transfer Start Packet ID	5.1.13.1.1	file_info_data_type
1	Client to Server	0x0403 – File Start Receipt Packet ID	5.1.13.1.2	file_receipt_data_type
2..n-3	Server to Client	0x0401 – File Data Packet ID	5.1.13.1.3	file_packet_data_type
3..n-2	Client to Server	0x0404 – Packet Receipt Packet ID	5.1.13.1.4	packet_receipt_data_type
n-1	Server to Client	0x0402 – File Transfer End Packet ID	5.1.13.1.5	file_end_data_type
n	Client to Server	0x0405 – File End Receipt Packet ID	5.1.13.1.6	path_specific_stop_file_receipt_data_type

NOTE: The File Transfer function is performed by the generic File Transfer method in Section 5.1.13.1.

After the File Transfer Receipt is sent back to the Server, the *Path Specific Stop Status I*nfo data is sent to the Server and is described in the next section below:

5.1.6.2.2.1 Path Specific Stop (PSS) Status Info Protocol

Once the Path Specific Stop file has been processed, the Stop route information is processed. The Client calculates the Stop route distance in meters (starting from the first destination to the final destination), which is sent back to the Server in the `path_specific_stop_info_to server_data_type` definition result packet, and contains the *unique_id*, *distance* and *result_code* shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1220 – Path Specific Stop Status Info	path_specific_stop_info_to_server_data_type
1	Client to Server	0x1221 – Path Specific Stop Status Info Receipt	path_specific_stop_info_to_client_receipt_type

```
typedef struct  /*  D614  */
    {
    uint32 unique_id;          /* Unique Stop ID */
    uint32 distance;           /* Calculated distance in meters (first destination to Stop)
    uint8 result_code;         /* 0 = No errors, non-zero indicates error (see table below)
    } path_specific_stop_info_to_server_data_type;
```

Error codes for the *path_specific_stop_info_to_server_data_type* packet are listed below:

Result Code (Decimal)	Meaning	Recommended Response
0	Success, PSS distance calculated	No errors
1	Incorrect number of routes	Verify Stop data, send Path Specific Path file again
2	Too many routes	Verify Stop data, send Path Specific Path file again
81	Calculation truncated with success	No errors, but calculation truncated
82	Invalid first destination	Verify Stop data, send Path Specific Path file again
83	Invalid stop destination	Verify Stop data, send Path Specific Path file again
84	Calculation overflow	Create two separate Path Specific Path files
85	Invalid route calculation	Send Path Specific Path file again
86	Route calculation canceled by another calculation	Send Path Specific Path file again
All other	Internal error	Contact Garmin support

After the Server receives the distance packet, it should respond back to the Client with an acknowledge packet that contains the *unique_id*. It should be noted the Client will continue to send the distance packet every 30 seconds until the Server correctly responds with the acknowledgement *path_specific_stop_info_to_client_receipt_type* definition shown below:

```
typedef struct  /*  D614  */
    {
    uint32 unique_id;       /* Unique Stop ID received from the Client distance packet */
    } path_specific_stop_info_to_client_receipt_type;
```

5.1.6.3 A618 Stop Protocol

The A618 Stop protocol provides the same functionality as the A603 Stop protocol, but A618 allows the Stop Text to extend up to 2,000 bytes (the A603 Stop Text has a maximum of 200 bytes). Since the A618 Stop data could potentially exceed 256 bytes, the Server will send this new Destination (Stop) to the Client using the existing File Transfer Protocol in Section 5.1.13.1.

5.1.6.3.1 A618 Stop file data format

The A618 Stop file must contain the following format:

Data Type	Name	Description
uint32	Signature	('S', 'T', 'O', 'P') Hard-coded value to indicate the file type
uint16	Format version	0 = File format version_0
uint32	Timestamp	Timestamp from Server
uint32	Unique ID number	Unique ID number for this Stop assigned by the Server
uchar_t8	Stop Text[up to 2,000 bytes]	Text shown on device->Dispatch->My Stops (up to 2,000 bytes including a NULL terminator)
sint32	Latitude	Latitude of Destination
sint32	Longitude	Longitude of Destination

5.1.6.3.2 A618 Stop send to Client Protocol

The Server has the option of either compressing the entire file (using the gZip format) or sending a regular non-compressed binary file. The Client will examine the file and determine if the file is compressed in gZip format or non-compressed binary format, and process the file accordingly. If the file is compressed and gZip errors are detected then the File End Receipt Packet will contain the error codes (shown in Section 5.1.13.1.6.5).

Any error detected during file download will halt the *A618 Stop* file transfer process, and that file will not be saved by the Client. Once the file successfully downloads to the Client, the file will be analyzed for format errors. Two distinct levels of result codes will be sent back to the Server (**one code to indicate the success of the file download process** and **one code to indicate the file format/content examination results**) in the *File End Receipt Packet* of Section 5.1.13.1.6.

N	Direction	Fleet Management Packet ID	Hyper-Link	Fleet Management Packet Data Type
0	Server to Client	0x0400 – File Transfer Start Packet ID	5.1.13.1.1	file_info_data_type
1	Client to Server	0x0403 – File Start Receipt Packet ID	5.1.13.1.2	file_receipt_data_type
2..n-3	Server to Client	0x0401 – File Data Packet ID	5.1.13.1.3	file_packet_data_type
3..n-2	Client to Server	0x0404 – Packet Receipt Packet ID	5.1.13.1.4	packet_receipt_data_type
n-1	Server to Client	0x0402 – File Transfer End Packet ID	5.1.13.1.5	file_end_data_type
n	Client to Server	0x0405 – File End Receipt Packet ID	5.1.13.1.6	log_file_receipt_data_type

NOTE: The File Transfer function is performed by the generic File Transfer in this document. Go to the desired section by clicking on the Hyperlink above.

5.1.7 Stop Status Protocol

This protocol is used by the Server to request or change the status of a Stop on the Client. The protocol is also used by the Client to send the status of a Stop to the Server whenever the status of a Stop changes on the Client. The packet sequences for the Stop status protocol are shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0210 – Stop Status Request Packet ID	stop_status_data_type
1	Client to Server	0x0211 – Stop Status Data Packet ID	stop_status_data_type
2	Server to Client	0x0212 – Stop Status Receipt	stop_status_receipt_data_type

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Client to Server	0x0211 – Stop Status Data Packet ID	stop_status_data_type
1	Server to Client	0x0212 – Stop Status Receipt	stop_status_receipt_data_type

The type definition for the *stop_status_data_type* is shown below.

```
typedef struct  /* D603 */
    {
    uint32      unique_id;
    uint16      stop_status;
    uint16      stop_index_in_list;
    } stop_status_data_type;
```

The *unique_id* contains the 32-bit unique identifier for the Stop. The Server should ignore any Stop status message with a *unique_id* of 0xFFFFFFFF. The *stop_status* will depend on the current status of the Stop on the Client or the status the Server would like to change the Stop to. The *stop_index_in_list* can either represent the current position of the Stop in the Stop list (0, 1, 2…) if the message is going from the Client to the Server or the position the Stop should be moved to in the Stop list if the message is going from the Server to the Client. The table below defines the *stop_status* and explains how the value of the *stop_status* affects the contents of the *stop_index_in_list*.

Stop Status	Value (Decimal)	Meaning
Requesting Stop Status	0	This is the Server requesting the status of a Stop from the Client. The value of the stop_index_in_list should be set to 0xFFFF and will be ignored by the Client.
Mark Stop As Done	1	This is the Server telling the Client to mark a Stop as done. The value of the stop_index_in_list should be set to 0xFFFF and will be ignored by the Client.

Activate Stop	2	This is the Server telling the Client to start navigating to a Stop. The value of the stop_index_in_list should be set to 0xFFFF and will be ignored by the Client.
Delete Stop	3	This is the Server telling the Client to delete a Stop from the list. The value of stop_index_in_list should be set to 0xFFFF and will be ignored by the Client.
Move Stop	4	This is the Server telling the Client to move a Stop to a new position in the list. The value of stop_index_in_list should be set to the position the Server would like to move the Stop to in the list.
Stop status – Active	100	This is the Client reporting the current status of a Stop as Active. The value of stop_index_in_list will correspond to the current position of the Stop in the list.
Stop status – Done	101	This is the Client reporting the current status of a Stop as Done. The value of stop_index_in_list will correspond to the current position of the Stop in the list.
Stop status – Unread Inactive	102	This is the Client reporting the current status of a Stop as unread and inactive. The value of stop_index_in_list will correspond to the current position of the Stop in the list.
Stop status – Read Inactive	103	This is the Client reporting the current status of a Stop as read and inactive. The value of stop_index_in_list will correspond to the current position of the Stop in the list.
Stop status – Deleted	104	This is the Client reporting the current status of a Stop as Deleted. The Client will return this status for any Stop not present in the Stop list. The value of stop_index_in_list will be set to 0xFFFF and should be ignored by the Server.

The type definition for the *stop_status_receipt_data_type* is shown below.

```
typedef struct  /* D603 */
    {
    uint32      unique_id;
    } stop_status_receipt_data_type;
```

The *unique_id* contains the 32-bit unique identifier for the Stop.

5.1.8 Estimated Time of Arrival (ETA) Protocol

This protocol is used by the Server to request ETA and destination information from the Client. The Client also uses this protocol to send ETA and destination information to the Server whenever the user starts navigating to a new FMI destination.

Note: Prior to the A616 protocol firmware version, ETA's were sent by the Client each time a device recalculation was performed.

ETA design enhancements were made during the A616 protocol firmware to provide the following functionality: ETA monitoring is performed by the FMI Client every 30 seconds, which could result in ETA packets being sent by the Client to the Server. During Client navigation to a destination, ETA packets are sent by the Client if all of the following conditions are true:

1. *The Client is moving at a speed greater than 5 mph*
2. *The Client's 30-second scheduled ETA calculation has changed by more than 15%*
   Note: Any new calculation delta greater than 20 minutes will be reported, and any new calculation delta less than 5 minutes will not be reported.
3. *The Client has a good GPS fix*
4. *The Client is more than 600 meters away from the destination*

The packet sequences for the ETA protocol are shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0200 – ETA Data Request Packet ID	None
1	Client to Server	0x0201 – ETA Data Packet ID	eta_data_type
2	Server to Client	0x0202 – ETA Data Receipt Packet ID	eta_data_receipt_type

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Client to Server	0x0201 – ETA Data Packet ID	eta_data_type
1	Server to Client	0x0202 – ETA Data Receipt Packet ID	eta_data_receipt_type

The type definition for the *eta_data_type* is shown below.

```
typedef struct  /* D603 */
    {
    uint32                  unique_id;
    time_type               eta_time;
    uint32                  distance_to_destination;
    sc_position_type        position_of_destination;
    } eta_data_type;
```

The *unique_id* is a 32-bit unsigned value that uniquely identifies the ETA message sent to the Server. The *eta_time* is the time that the Client expects to arrive at the currently active destination. If the *eta_time* is set to 0xFFFFFFFF, then the Client does not have a destination active. The *distance_to_destination* is the distance in meters from the Client to the currently active destination. If the *distance_to_destination* is set to 0xFFFFFFFF, then the Client does not have a destination active. The *position_of_destination* is the location of the currently active destination on the Client.

The type definition for the *eta_data_receipt_type* is shown below.

```
typedef struct  /* D603 */
    {
    uint32      unique_id;
    } eta_data_receipt_type;
```

The *unique_id* is a 32-bit unsigned value that uniquely identifies the ETA message sent to the Server.

5.1.9 Auto-Arrival at Stop Protocol

This protocol is used by the Server to change the auto-arrival criteria on the Client. The auto-arrival feature is used on the Client to automatically detect that the user has arrived at a Stop and then to prompt the user if they would like to mark the Stop as done and start navigating to the next Stop in the list. Once the Server sends the auto-arrival at Stop protocol to the Client, the setting will be permanent on the Client until the Server changes it. The packet sequence for the Auto-Arrival at Stop protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0220 – Auto-Arrival Data Packet ID	auto_arrival_data_type

The type definition for the *auto_arrival_data_type* is shown below.

```
typedef struct  /* D603 */
    {
    uint32      stop_time;
    uint32      stop_distance;    /* in meters */
    } auto_arrival_data_type;
```

The *stop_time* value is time in seconds for how long the Client should be stopped close to the destination before the auto-arrival feature is activated. The default for *stop_time* on the Client is 30 seconds. To disable the auto-arrival stop time, set *stop_time* to 0xFFFFFFFF. The *stop_distance* is the distance in meters for how close the Client has to be to the destination before the auto-arrival feature is activated. The default for *stop_distance* on the Client is 100

meters. To disable the auto-arrival stop distance, set *stop_distance* to 0xFFFFFFFF. To disable the auto-arrival feature, set both *stop_time* and *stop_distance* to 0xFFFFFFFF.

5.1.10 Sort Stop List Protocol

This protocol is used to sort all Stops in the list such that they can be visited in order in the shortest total distance possible starting from the driver's current location.

This protocol is only supported on Clients that report A604 as part of their protocol support data. The packet sequence for the Auto-Arrival at Stop protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0110 – Sort Stop List	None
1	Client to Server	0x0111 – Sort Stop List Acknowledgement	None

5.1.11 Waypoint Protocols

There are protocols available to create, modify, and delete waypoints that appear under Favorites on the Client. Only waypoints created through the Create Waypoint Protocol may be subsequently modified and deleted.

5.1.11.1 Create Waypoint Protocol

This protocol allows the Server to create or modify a waypoint on the Client. The packet sequence for creating a waypoint is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0130 – Create Waypoint Packet ID	waypoint_data_type
1	Client to Server	0x0131 – Create Waypoint Receipt Packet ID	waypoint_receipt_data_type

The type definition for the *waypoint_data_type* is shown below.

```
typedef struct  /* D607 */
    {
    uint16             unique_id;
    map_symbol         symbol;
    sc_position_type   posn;
    uint16             cats;
    uchar_t8           name[/* 31 bytes, null-terminated */];
    uchar_t8           comment[/* variable length, null-terminated string, 51 bytes max */];
    } waypoint_data_type;
```

The *unique_id* is a unique identifier for the waypoint. If the specified *unique_id* is already in use, then the existing waypoint will be modified instead of creating a new waypoint. The symbol is the map symbol displayed for this waypoint on the Client. The posn is the position of the waypoint. The cat is a bit field that indicates what categories to put the waypoint in. For example, if the waypoint should be in categories with IDs 0 and 5, cat should have the lowest bit and the 6th lowest bit set to 1 for a value of 33 decimal (00000000 00100001 in binary). The name is the name of the waypoint. The comment is any other notes about the waypoint that should be displayed on the Client. The desired category should already exist before any waypoints are added to it. Any attempt to add a waypoint to a nonexistent category will result in an error code of 0. The added waypoint will remain out of the category.

The type definition for *waypoint_receipt_data_type* is shown below.

```
typedef struct  /* D607 */
    {
    uint16      unique_id;
    boolean     status_code;
    uint8       reserved;         /* set to 0 */
    } waypoint_receipt_data_type;
```

The *unique_id* is the unique ID of the waypoint received. The *status_code* is true if the operation was successful and false otherwise.

5.1.11.2 Waypoint Deleted Protocol

The Client sends this packet when a Fleet Management waypoint is deleted, whether the delete was initiated from the Client side or on the Server side. The packet sequence for the Waypoint Deleted Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Client to Server	0x0133 – Waypoint Deleted Packet ID	waypoint_deleted_data_type
1	Server to Client	0x0134 – Waypoint Deleted Receipt Packet ID	waypoint_deleted_receipt_data_type

The type definition for the *waypoint_deleted_data_type* is shown below.

```
typedef struct  /* D607 */
    {
    uint16      unique_id;
    boolean     status_code;
    uint8       reserved;         /* set to 0 */
    } waypoint_deleted_data_type;
```

The *unique_id* is the unique ID of the waypoint deleted. The *status_code* is true if the waypoint with the specified *unique_id* no longer exists. The *status_code* will always be true when the waypoint is deleted from the Client side.

The type definition for the *waypoint_deleted_receipt_data_type* is shown below.

```
typedef struct  /* D607 */
    {
    uint16      unique_id;
    } waypoint_delete_data_type;
```

The *unique_id* is the unique ID of the waypoint deleted.

5.1.11.3 Delete Waypoint Protocol

This protocol allows the Server to delete a waypoint on the Client. The packet sequence for deleting a waypoint is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0132 – Delete Waypoint Packet ID	waypoint_delete_data_type
1	Client to Server	0x0133 – Waypoint Deleted Packet ID	waypoint_deleted_data_type
2	Server to Client	0x0134 – Waypoint Deleted Receipt Packet ID	waypoint_deleted_receipt_data_type

The type definition for the *waypoint_delete_data_type* is shown below.

```
typedef struct  /* D607 */
    {
    uint16      unique_id;
    } waypoint_delete_data_type;
```

The *unique_id* is the unique ID of the waypoint to be deleted.

5.1.11.4 Delete Waypoint by Category Protocol

This protocol allows the Server to delete all waypoints on the Client that belong to a particular category. The packet sequence for deleting a waypoint is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0135 – Delete Waypoint by Category Packet ID	waypoint_delete_by_cat_data_type
1	Client to Server	0x0136 – Delete Waypoint by Category Receipt Packet ID	waypoint_delete_by_cat_receipt_data_type

The type definition for the *waypoint_delete_by_cat_data_type* is shown below.

```
typedef struct  /* D607 */
    {
    uint16      cats;
    } waypoint_delete_by_cat_data_type;
```

The *cats* is a bit field which accepts multiple categories in the same way that the waypoint creation protocol does.

The type definition for the *waypoint_delete_by_cat_receipt_data_type* is shown below.

```
typedef struct  /* D607 */
    {
    uint16      cats;
    uint16      count;
    } waypoint_delete_by_cat_data_type;
```

The *cats* is the same bit field that was passed to the Client. The *count* is the number of waypoints deleted by the Delete Waypoint by Category protocol.

5.1.11.5 Create Waypoint Category Protocol

This protocol allows the Server to create or modify a waypoint category on the Client. The packet sequence for creating a waypoint category is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0137 – Create Waypoint Category Packet ID	waypoint_cat_data_type
1	Client to Server	0x0138 – Create Waypoint Category Receipt Packet ID	waypoint_cat_receipt_data_type

The type definition for *waypoint_cat_data_type* is shown below.

```
typedef struct  /* D607 */
    {
    uint8       cat_id;
    char        name[/* variable length, null-terminated string, 17 bytes max */];
    } waypoint_cat_data_type;
```

The *cat_id* is an identifier for the category. Its value can be between 0 and 15. If a category with the given ID already exists, then the existing category will be modified and no new category will be created. The *name* is the category's name.

The type definition for the *waypoint_cat_receipt_data_type* is shown below.

```
typedef struct  /* D607 */
    {
    uint8       cat_id;
    boolean     status_code;
    } waypoint_cat_receipt_data_type;
```

The *cat_id* is the category's ID. The *status_code* is *true* if the operation was successful and *false* otherwise.

5.1.12 Driver ID and Status Protocols

These protocols are used to identify the current driver and status. A driver ID may be any text string enterable on the keyboard. The Server specifies a list of statuses the driver can select from.

5.1.12.1 Driver ID Monitoring Protocols

The Driver ID Monitoring Protocols are used to communicate the driver ID. This ID can be set by the Server and sent to the device, or changed by the user on the Driver Information page of the Client device.

5.1.12.1.1 A607 Server to Client Driver ID Update Protocol

The A607 Server to Client Driver ID Update Protocol is used to change the driver ID of the current driver on the Client device. The packet sequence for the A607 Server to Client Driver ID Update Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0813 – A607 Server to Client Driver ID Update Packet ID	driver_id_d607_data_type
1	Client to Server	0x0812 – Driver ID Receipt Packet ID	driver_id_receipt _data_type

The type definition for the *driver_id_d607_data_type* is shown below.

```
typedef struct  /* D607 */
    {
    uint32     change_id;
    time_type  change_time;   /* timestamp of status change */
    uint8      driver_idx;
    uint8      reserved[3];   /* set to 0 */
    uchar_t8   driver_id[];   /* variable length, null terminated string, 50 bytes max */
    uchar_t8   password[];    /* variable length, null terminated string, 20 bytes max */
    } driver_id_d607_data_type;
```

The *change_id* is a unique number per driver used to identify this status change request. The *change_time* is the timestamp when the specified driver ID took effect. The *driver_idx* is the zero-based index of the driver to change. If the multiple drivers feature is disabled, this should always be 0. The *driver_id* is the new *driver_id*. The *password* is ignored by the Client. It is only used when the Client attempts to update the driver ID when driver passwords are enabled.

The type definition for the *driver_id_receipt_data_type* is shown below.

```
typedef struct  /* D604 */
    {
    uint32   change_id;
    boolean  result_code;
    uint8    driver_idx;           /* D607 only */
    uint8    reserved[2];          /* Set to 0 */
    } driver_id_receipt_data_type;
```

The *change_id* identifies the driver ID update being acknowledged. The *result_code* indicates whether the update was successful. This will be *true* if the update was successful or *false* otherwise (for example, the *driver_idx* is out of range). The *driver_idx* is the zero-based index of the driver updated.

Note: For Clients that do not report D607 support, this field is reserved and should always be set to 0.

5.1.12.1.2 A607 Client to Server Driver ID Update Protocol

The A607 Client to Server Driver ID Update Protocol is used to notify the Server when the driver changes the driver ID via the user interface on the Client. The packet sequence for the A607 Client to Server Driver ID Update Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Client to Server	0x0813 – A607 Client to Server Driver ID Update Packet ID	driver_id_d607_data_type
1	Server to Client	0x0812 – Driver ID Receipt Packet ID	driver_id_receipt_data_type

The type definitions for the *driver_id_d607_data_type* and *driver_id_receipt_data_type* are described in Section 5.1.12.1.1. If driver passwords are enabled, the driver ID will not be changed on the Client until the driver ID receipt packet is received and the *result_code* is *true*.

5.1.12.1.3 A607 Server to Client Driver ID Request Protocol

The A607 Server to Client Driver ID Request Protocol is used by the Server to obtain the driver ID currently stored in the device. If no driver ID has been set, a zero length string will be returned in the *driver_id_data_type*. This protocol is only supported on Clients that report A607 as part of their protocol support data. The packet sequence for the Server to Client Driver ID Request Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0810 – Request Driver ID Packet ID	driver_index_data_type
1	Client to Server	0x0813 – A607 Client to Server Driver ID Update Packet ID	driver_id_d607_data_type
2	Server to Client	0x0812 – Driver ID Receipt Packet ID	driver_id_receipt_data_type

The type definitions for the *driver_id_d607_data_type* and *driver_id_receipt_data_type* are described in section 5.1.12.1.1. These data types are only supported on Clients that include D607 in their protocol support data.

The type definition for the *driver_index_data_type* is shown below. This data type is only supported on Clients that include D607 in their protocol support data.

```
typedef struct  /* D607 */
    {
    uint8 driver_idx;
    uint8 reserved[3];        /* set to 0 */
    } driver_index_data_type;
```

The *driver_idx* is a zero-based index that specifies which driver's ID to request. If multiple drivers are not enabled, it should always be 0.

5.1.12.2 Other Driver ID Monitoring Protocols (Deprecated)

Other Driver ID Monitoring protocols are described in Section 6.6.2. These protocols are deprecated, and could exist now, but may be removed in the near future.

5.1.12.3 Driver Status List Protocols

The Driver Status List Maintenance Protocols allow the Server to maintain (add, update, or delete) the list of driver statuses that the user may select from. Each driver status consists of a numeric identifier and an associated text

string. In the Client user interface for the device, the numeric identifier is not displayed, and the list is presented in ascending order by identifier. This allows the Server to control the display order.

The identifier 0xFFFFFFFF should not be used for a Server-defined status; it is used within the device and in the Other Driver Status Monitoring protocols are described in Section 6.6.3. to indicate that the status has not been set.

5.1.12.3.1 Set Driver Status List Item Protocol

This protocol allows the Server to set (add or update) the textual description corresponding to a particular driver status. The driver status list may contain up to 16 items.

This protocol is only supported on Clients that report A604 as part of their protocol support data. The packet sequence for the Set Driver Status List Item Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0800 – Set Driver Status List Item Packet ID	driver_status_list_item_data_type
1	Client to Server	0x0802 – Set Driver Status List Item Receipt Packet ID	driver_status_list_item_receipt_data_type

The type definition for the *driver_status_list_item_data_type* is shown below.

```
typedef struct  /* D604 */
    {
    uint32   status_id;          /* status identifier  */
    uchar_t8 status[];           /* status text, null terminated (50 bytes max) */
    } driver_status_list_item_data_type;
```

The *status_id* is a unique number corresponding to the driver status. If there is already a list item on the device with the specified *status_id*, the *status* text is updated; otherwise, the *status* text is added to the list.

The type definition for the *driver_status_list_item_receipt_data_type* is shown below. This data type is only supported on devices that report D604 as part of their protocol support data.

```
typedef struct  /* D604 */
    {
    uint32  status_id;         /* message identifier */
    boolean result_code;       /* result (true if successful, false otherwise) */
    uint8   reserved[3];       /* Set to 0 */
    } driver_status_list_item_receipt_type;
```

The *status_id* will be the same as the *status_id* from the *driver_status_list_item_data_type*. The *result_code* will be *true* if the status item was added to the device successfully or *false* otherwise (for example, the status list already contains the maximum number of items).

5.1.12.3.2 Delete Driver Status List Item Protocol

This protocol allows the Server to delete (remove) a textual description corresponding to a particular driver status. The Server may not remove the driver's current status; if this occurs, the Client will report a failure.

This protocol is only supported on Clients that report A604 as part of their protocol support data. The packet sequence for the Delete Driver Status List Item Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0801 – Delete Driver Status List Item Packet ID	driver_status_list_item_delete_data_type
1	Client to Server	0x0803 – Delete Driver Status List Item Receipt Packet ID	driver_status_list_item_receipt_data_type

The type definition of the *driver_status_list_item_delete_data_type* is shown below.

```
typedef struct  /* D604 */
    {
    uint32  status_id;         /* message identifier */
    } driver_status_list_item_delete_data_type;
```

The *status_id* identifies the list item to be deleted.

The type definition for the *driver_status_list_item_receipt_data_type* is defined in Section 5.1.12.3.1 and repeated below. This data type is only supported on devices that report D604 as part of their protocol support data.

```
typedef struct  /* D604 */
    {
    uint32  status_id;         /* message identifier */
    boolean result_code;       /* result (true if successful, false otherwise) */
    uint8   reserved[3];       /* Set to 0 */
    } driver_status_list_item_receipt_type;
```

The *status_id* will be the same as the *status_id* from the *driver_status_list_item_delete_data_type*. The *result_code* will be *true* if the status item was deleted from the device or was not found or *false* if the status item is still on the device (for example, the *status_id* corresponds to the driver's current status).

5.1.12.3.3 Refresh Driver Status List Protocol

This protocol allows the Client to request the complete list of driver statuses from the Server. In response to this request from the Client, the Server shall initiate a Set Driver Status List Item protocol for each item that should be in the driver status list.

This protocol is only supported on Clients that report A604 as part of their protocol support data, and is throttled by default on Clients that report A605 as part of their protocol support data. See the Message Throttling Protocols (Section 5.1.17) to enable the Refresh Driver Status List Protocol on these Clients.

The packet sequence for the Delete Driver Status List Item Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Client to Server	0x0804 –Driver Status List Refresh Packet ID	N/A
1..n		(Set Driver Status List Item protocols)	

5.1.12.4 Driver Status Monitoring Protocols

The Driver Status Monitoring Protocols are used to communicate the driver status. This status can be set by the Server and sent to the device, or changed by the user on the Driver Information page of the Client device. Before protocols can be used, the Server must set the allowed driver statuses using the Driver Status List Protocols in Section 5.1.12.3.

5.1.12.4.1 A607 Server to Client Driver Status Update Protocol

The Server to Client Driver Status Update Protocol is used to change the status of the current driver on the Client device. This protocol is only supported on Clients that report A604 as part of their protocol support data. The packet sequence for the Server to Client Driver Status Update Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0823 – A607 Server to Client Driver Status Update Packet ID	driver_status_d607_data_type
1	Client to Server	0x0822 – Driver Status Receipt Packet ID	driver_status_receipt_data_type

The type definition for the *driver_status_d607_data_type* is shown below. This data type is only supported on Clients that include D607 in their protocol support data.

```
typedef struct  /* D607 */
    {
    uint32     change_id;           /* unique identifier */
    time_type  change_time;         /* timestamp of status change */
    uint32     driver_status;       /* ID corresponding to the new driver status */
    uint8      driver_idx;
    uint8      reserved[3];         /* set to 0 */
    } driver_status_d607_data_type;
```

The *change_id* is a unique number which identifies this status update message. The *change_time* is the timestamp when the specified driver status took effect. The *driver_idx* is the zero-based index of the driver to change. If the multiple drivers feature is disabled, this should always be 0.

The type definition for the *driver_status_receipt_data_type* is shown below. This data type is only supported on Clients that include D604 in their protocol support data.

```
typedef struct
    {
    uint32  change_id;
    boolean result_code;
    uint8   driver_idx;               /* D607 only */
    uint8   reserved[2];              /* Set to 0 */
    } driver_status_receipt_data_type;
```

The *change_id* identifies the status update being acknowledged. The *result_code* indicates whether the update was successful. This will be *true* if the update was successful or *false* otherwise (for example, the *driver_status* is not on the Client). The *driver_idx* is the zero-based index of the driver updated.

Note: For Clients that do not report D607 support, this field is reserved and should always be set to 0.

5.1.12.4.2 A607 Client to Server Driver Status Update Protocol

The **Client to Server** Driver Status Update Protocol is used to notify the Server when the driver changes the driver status via the user interface on the Client. This protocol is only supported on Clients that report A604 as part of their protocol support data. The packet sequence for the A607 Client to Server Driver Status Update Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Client to Server	0x0823 – A607 Client to Server Driver Status Update Packet ID	driver_status_data_d607_type
1	Server to Client	0x0822 – Driver Status Receipt Packet ID	driver_status_receipt_data_type

The type definitions for the *driver_status_d607_data_type* and *driver_status_receipt_data_type* are described in Section 5.1.12.4.1. These data types are only supported on Clients that include D604 in their protocol support data.

5.1.12.4.3 A607 Server to Client Driver Status Request Protocol

The **Server to Client** Driver Status Request Protocol is used by the Server to obtain the driver status currently stored in the device. If no driver status has been set, an ID of 0xFFFFFFFF will be returned as the driver status. This protocol is only supported on Clients that report A604 as part of their protocol support data. The packet sequence for the A607 Server to Client Driver Status Request Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0820 – Request Driver Status Packet ID	driver_index_data_type
1	Client to Server	0x0823 – A607 Client to Server Driver Status Update Packet ID	driver_status_d607_data_type

2	Server to Client	0x0822 – Driver Status Receipt Packet ID	driver_status_receipt_data_type

The type definition for the *driver_status_d607_data_type* and *driver_status_receipt_data_type* are described in Section 5.1.12.4.1. The type definition for the *driver_index_data_type* is described in Section 5.1.12.1.3. These data types are only supported on Clients that include D607 in their protocol support data.

5.1.12.5 Other Driver Status Monitoring Protocols (Deprecated)

Other Driver Status Monitoring protocols are described in Section 6.6.3. These protocols are deprecated, and could exist now, but may be removed in the near future.

5.1.13 File Transfer Protocols

The following protocols are used to transfer files between the Server and Client, and allow the Server to obtain information about the files on the Client device.

5.1.13.1 Server to Client - File Transfer Protocol

This protocol sends a file from the **Server to Client**. The Server initially divides the file into small data packets (each packet contains Header data and up to 245 bytes of file data), and sends the data packets one by one to the Client using the protocol described below.

The Client saves the data packet(s) to a temporary file, and acknowledges each packet as it is received. After the final data packet of the file is received by the Client, the CRC32 is checked, and the file is saved and processed. Lastly, a final end-of-file acknowledgment packet is sent to the Server, which indicates the success of the complete file transfer and processing of the file.

Note: Several file types are received and processed by the Client using this generic file transfer procedure.

The File Transfer Protocol packet sequence is listed below as three distinct transfer stages. The first stage initiates the file transfer process:

N	Direction	Fleet Management Packet ID	Hyper-link	Fleet Management Packet Data Type
0	Server to Client	0x0400 – File Transfer Start Packet ID	5.1.13.1.1	file_info_data_type
1	Client to Server	0x0403 – File Start Receipt Packet ID	5.1.13.1.2	file_receipt_data_type

…next, repeat the following Server and Client data stage until all data is sent:

2..n-3	Server to Client	0x0401 – File Data Packet ID	5.1.13.1.3	file_packet_data_type
3..n-2	Client to Server	0x0404 – Packet Receipt Packet ID	5.1.13.1.4	packet_receipt_data_type

…all file data was sent, so end the file transfer sequence as shown below:

n-1	Server to Client	0x0402 – File Transfer End Packet ID	5.1.13.1.5	file_end_data_type
n	Client to Server	0x0405 – File End Receipt Packet ID	5.1.13.1.6	file_receipt_data_type

NOTE: Each of the packets in the table above will be discussed below. Go to the desired section by clicking on the Hyperlink above.

5.1.13.1.1 Server to Client - Packet ID: 0x0400 - File Transfer Start

This *File Transfer Start* step initiates a file transfer from the Server to the Client. The data needed to perform this step is shown in the type definition *file_info_data_type* below, and should be sent to the Client.

```
typedef struct
    {
    uint32 file_size;
    uint8  file_version_length;
    uint8  file_type;                  /* See table below */
    uint8  reserved[2];                /* Set to 0 */
    uint8  file_version[/* 16 bytes */];
    } file_info_data_type;
```

The *file_size* is the size of the file that will be transferred, in bytes. The maximum file size is limited by the amount of available space on the device. The *file_version* contains up to 16 bytes of arbitrary data to be associated with the file being transmitted, as a version number or for other purposes. The *file_version_length* indicates the number of bytes of *file_version* that are valid. The *file_type* is the type of file being transferred to the Client, and is shown below:

File Type	Meaning
0	GPI File
1	A610 AOBRD Event Log File
2	A612 Custom Form Template
3	A614 Path Specific Stop
4	A615 IFTA
5	A618 Stop

Go to packet sequence table, click here> 5.1.13.1

5.1.13.1.2 Client to Server - Packet ID: 0x0403 - File Start Receipt

Once the Client correctly receives the *File Transfer Start* packet from the previous step, the Client responds back to the Server with a receipt packet. The receipt data type definition *file_receipt_data_type* shown below is created by the Client, and sent back to the Server. Note, the *file_type* from the *File Transfer Start* data is also included in the response.

```
typedef struct
    {
    uint8  result_code;
    uint8  file_type;          /* See table below */
    uint8  reserved[2];        /* Set to 0 */
    } file_receipt_data_type;
```

The *file_type* will be identical to the value received in the *File Transfer Start* Packet ID. The *result_code* indicates whether the operation was successful. Result codes and their meanings are described in the table below. At a minimum, the Server must differentiate between a result code of zero, indicating success, and a non-zero result code, indicating failure.

File Type	Meaning
0	GPI File
1	A610 AOBRD Event Log File
2	A612 Custom Form Template
3	A614 Path Specific Stop
4	A615 IFTA
5	A618 Stop

Result Code (Decimal)	Meaning	Recommended Response
0	Success	None.
2	Insufficient space on device	The user of the device should remove any unnecessary files to make space available.
3	Unable to open file	Invalid file.
4	No transfer in progress	The Server should resend the GPI File Transfer Start packet.
5	Transfer not allowed	Try again later or wait for a file request before trying again.
8	File Operation error	Verify that the file can be opened when transferred to a device using a local connection such as USB or an SD card.
9	File Operation error	The Server should restart the entire file transfer.
13	Busy	Path Specific Stop file type detected, but currently busy. Try again later.

Go to packet sequence table, click here> 5.1.13.1

5.1.13.1.3 Server to Client - Packet ID: 0x0401 - File Data

After the Server receives the Client's *File Start Receipt* packet, the Server sends *File Data* packets to the Client. These packets contain the actual file data. The type definition for the *file_packet_data_type* is shown below.

***Note:* The *file_data* is limited to 245 bytes. If a file contains more than 245 bytes then multiple file data packets will be required to be sent.**

```
typedef struct
    {
    uint32 offset;             /* offset of this data from the beginning of the file */
    uint8  data_length;        /* length of file_data (0..245)   */
    uint8  file_type;          /* type of file being tranferred */
    uint8  reserved[2];        /* Set to 0 */
    uint8  file_data[245 max]; /* File data, 245 bytes maximum per packet, file_data[] greater */
                               /* than 245 bytes will require additional packets to be sent */
    } file_packet_data_type;
```

The *offset* indicates the position that should be written in the file; the first byte of the file has offset zero. The *data_length* indicates the number of bytes of file data that are being transmitted in this packet. The *file_type* will be identical to the value received in the File Transfer Start Packet ID. The *file_data* is the actual data chunk to be written to the file.

File Type	Meaning
0	GPI File
1	A610 AOBRD Event Log File
2	A612 Custom Form Template
3	A614 Path Specific Stop
4	A615 IFTA - **Not sent to Server**
5	A618 Stop

***Note:* The Server must send file packets in ascending order by *offset.* The Server may vary *data_length* from packet to packet to suit the needs of the application.**

Go to packet sequence table, click here> 5.1.13.1

5.1.13.1.4 Client to Server - Packet ID: 0x0404 – Data Packet Receipt

The Client responds to *File Data* packets sent from the Server with a *Data Packet Receipt* packet. The type definition for the *packet_receipt_data_type* is shown below.

```
typedef struct
    {
    uint32 offset;              /* offset of data received */
    uint32 next_offset;         /* offset of next data the Server should send */
    } packet_receipt_data_type;
```

The *offset* is the offset of the file packet received, which is the same as the *offset* of the corresponding *file_packet_data_type* that contained the file data from the Server. The next_offset indicates the offset that the Server should use when sending the next chunk of file data. The *offset* and the *next_offset* are to be interpreted as follows:

- Normally, the *next_offset* will be equal to the sum of the *offset* and the *data_length* from the corresponding *file_packet_data_type*.
- If the *next_offset* is equal to the size of the file, all file data has been received.
- If the *next_offset* is less than the *offset*, the Client has rejected the file data, as the data beginning at *next_offset* has not yet been received.
- If the *next_offset* is equal to *offset*, a temporary error has occurred; the Server should resend the data packet.
- If the *next_offset* is equal to 0xFFFFFFFF hexadecimal (4294967295 decimal) a severe error has occurred; the transfer should be aborted.

These rules enable a simple mechanism for event-driven file transfer, when the Server receives the *packet_receipt_data_type*, it should send the file data beginning at *next_offset*, unless an error has occurred or the entire file has been sent. If this approach is taken, the Server should also check the *offset* received against the *offset* sent; if they do not match, the receipt packet should be ignored, as it indicates a delayed receipt after the Server retransmitted a packet.

Go to packet sequence table, click here> 5.1.13.1

5.1.13.1.5 Server to Client - Packet ID: 0x0402 - File Transfer End

The *File Transfer End* packet is sent by the Server when all of the file data has been successfully sent to the Client. This packet contains the file CRC value for the complete file that was sent from the Server, so the Client will now verify the CRC value against the received file.

The type definition for the *file_end_data_type* is shown below.

```
typedef struct
    {
    uint32 crc;           /* CRC of entire file as computed by UTL_calc_crc32 */
    } file_end_data_type;
```

The *crc* is the CRC32 checksum of the entire file. The CRC32 algorithm is included in an Appendix (Section 6.4) and in electronic form in the *Fleet Management Interface Developer Kit*.

Go to packet sequence table, click here> 5.1.13.1

5.1.13.1.6 Client to Server - Packet ID: 0x0405 - File End Receipt

The *File End Receipt* packet is sent by the **Client to Server** following the reception of *File Transfer End* packet and any Client post- processing of the File Transfer data. Typically each *file_type* received by the Client requires specific post-processing, which is described below by *file_type*.

Once the post-processing has completed, a receipt packet is generated by the Client and sent back to the Server to indicate the results of the post-processing by the Client.

Refer to receipt format based by File Type of file sent:

File Type	Meaning	*file_type* Hyperlink
0	GPI File	5.1.13.1.6.1
1	A610 AOBRD Event Log File	5.1.13.1.6.2
2	A612 Custom Form Template	5.1.13.1.6.3
3	A614 Path Specific Stop	5.1.13.1.6.4
4	A615 IFTA – **Not sent to Server**	N/A
5	A618 Stop	5.1.13.1.6.5

5.1.13.1.6.1 Post Processing Receipt - GPI file type

```
typedef struct
    {
    uint8   result_code;           /* 0 = No transfer errors, non-zero shows error */
    uint8   file_type;             /* GPI file type = 0 */
    uint8   reserved[2];           /* Set to 0 */
    } gpi_file_receipt_data_type;
```

Result Code	Meaning	Recommended Response
0	Success, GPI file downloaded	None
1	CRC error	The Server should re-send the GPI file.
4	File not transferred error	The Server should re-send the GPI file.
5	GPI file contents error	Send a valid GPI file.
6	Invalid file type detected	Send a GPI file type.
7	Unable to process the GPI file	The Server should re-send the GPI file.

Go to packet sequence table, click here> 5.1.13.1

5.1.13.1.6.2 Post Processing Receipt - A610 AOBRD Event file type

During an AOBRD driver login session, an AOBRD Driver Status Change Log file is requested by the Client. The log file receipt is sent to the Server following log processing by the Client.

```
typedef struct  /*  D610  */
    {
    uint8     result_code;
    uint8     multi_use;       /* If result_code is NOT = 0 or 5 then it's file_type */
                               /* If result_code = 0 or 5 then use AOBRD Parsing Results below */
    uint16    record_counter;  /* Last record read in log file */
    } log_file_receipt_data_type;
```

The *result_code* indicates whether the operation was successful. Result codes and their meanings are described in the table below. At minimum, the Server must differentiate between a result code of zero, indicating success, and a nonzero result code, indicating failure.

Result Code	Meaning	Recommended Response
0	Success, AOBRD log file downloaded	None
1	CRC error	The Server should re-send the file
3	Unable to process	Send an AOBRD file from a Client request
4	File not transferred error	The Server should re-send the file
5	AOBRD file contents error	***multi_use* field represents *AOBRD Parsing Results* in table below**
6	Invalid file type detected	Send an AOBRD file type

After a file is received, parsing is performed, which halts on the first error found (determined when *result_code* = 5) along with an error code set in the multi_use field to reflect the Client's *AOBRD Parsing Results*. The *record_counter* indicates the faulty record that stopped the processing within that log file. (If no error occurs, the *record_counter* indicates the total number of records processed.)

AOBRD Parsing Results (Decimal)	**Error Type Text**	**Error Meaning**
0	No errors	None
1	Unused	Not used in log file receipts
2	Unable to open file	The Server should restart the entire file transfer
3	Internal file system error	The user of the device should remove any unnecessary files to make space available. The Server should restart the entire file transfer
4	Invalid record type	Record entry is not a Drive Status Change record
5	Wrong record version	Record entry has an invalid record version, and cannot be read
6	Unexpected record data	Record did not contain the correct data to allow processing
7	Invalid text length	Record contained a text string field that was too large
8	Unused data text length	Record contained an unused text string field that was too large
9	Non-printable text	Record contained non-printable text in a text string
10	Internal storage error_1	Initial data storage setup error
11	Non-zero timestamp	First record contained a timestamp value of zero
12	Wrong driver log	Wrong driver ID contained in first record
13	No driver profile found	Internal error of not finding a driver's profile in system
14	Internal storage error_2	Initial data storage container error
15	Wrong driver record	Following first record, an invalid driver ID was found in a record
16	Invalid timestamp	Record timestamp was not greater (newer) than previous record
17	Invalid driver status	Record contained an invalid old driver status or new driver status
18	Invalid status link	Record old driver status did not match new status of previous record
19	Internal storage error_3	Unable to store a driver record
20	Invalid last driver status	Last record in log did not set new driver status to "OFF DUTY"
21	Internal storage error_4	Could not store the log entries
22	Driver status error_1	Record contained unknown status data
23	Driver status error_2	Record contained unknown status data
24	Driver status error_3	Record contained unknown status data
25	Driver status error_4	Record contained unknown status data

Go to packet sequence table, click here> 5.1.13.1

5.1.13.1.6.3 Post Processing Receipt – A612 Custom Form file type

After the Client processes the Custom Form Template file, the **Client to Server** type definition for the *custom_form_file_receipt_data_type* is sent to the Server (shown below) to report the success status of the received Custom FormTemplate.

```
typedef struct  /*  D612  */
    {
    uint8   result_code;       /* 0 = No transfer errors, non-zero shows error */
    uint8   multi_use;         /* If result_code is NOT = 0 or 11 then it's file_type */
                               /* If result_code = 0 or 11 then use CF Parsing Results below */
    uint16  last_line_number;  /* Last line processed of Custom Form Template */
    } custom_form_file_receipt_data_type;
```

Result Code (Decimal)	**Meaning**	**Recommended Response**

0	Success, Custom Form downloaded	None
1	CRC error	The Server should resend Custom Form file
4	File not transferred error	The Server should resend Custom Form file
5	PSS file contents error	See Custom Form Return Code (below), and send a valid Custom Form file
6	Invalid file type detected	Send a Custom Form file type
11	File contents error	***multi_use* field represents *CF Parsing Results* in table below**
12	Invalid gZip file format	The Server should resend the Custom Form file

NOTE: Table below only valid when *result_code* is set to 0 or 11 (decimal). This table uses *multi_use* to indicate the CF file parsing results.

CF Parsing Results (Decimal)	Meaning	Recommended Response
0	Success, Template downloaded	None
11-41	Schema Error	Ensure correct XML encoding, and re-send
101 - 139	XML Parsing Error	Ensure correct XML data, and re-send
205	Exceeds 10 Templates on device	Server should first delete a Template on device

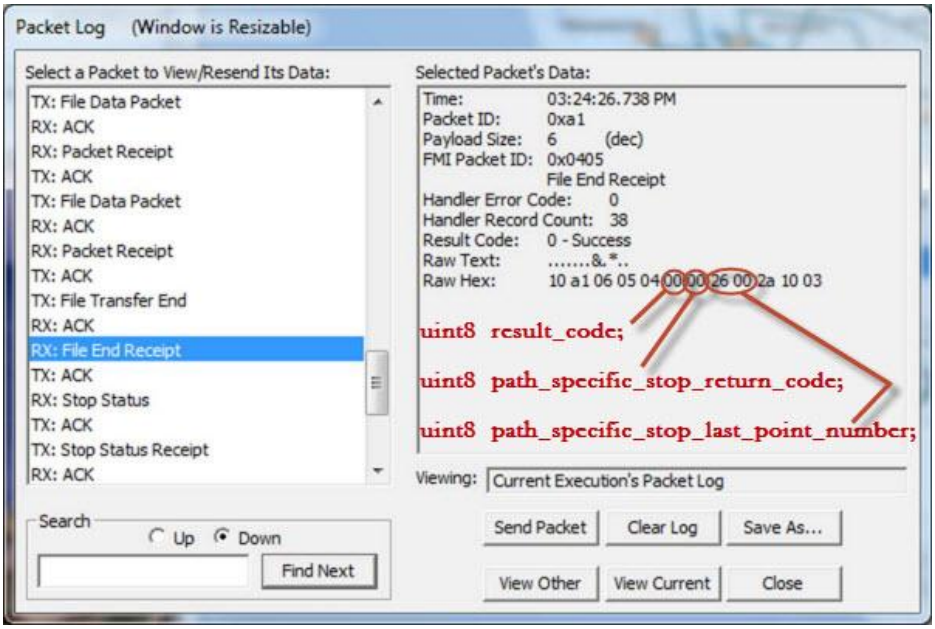
Go to packet sequence table, click here> 5.1.13.1

5.1.13.1.6.4 Post Processing Receipt – A614 Path Specific Stop (PSS) file type

After the Client processes the Path Specific Stop file, the **Client to Server** type definition for the *path_specific_stop_file_receipt_data_type* is sent to the Server (shown below) to report the success status of the received Path Specific Stop.

```
typedef struct  /*  D614  */
    {
    uint8   result_code;        /* 0 = No transfer errors, non-zero is error */
    uint8   multi_use;          /* If result_code is NOT = 0 or 11 then it's file_type */
                                /* If result_code = 0 or 11 then use PSS Return Code below */
    uint16  path_specific_stop_last_point_number; /* Last processed point (1=First point) */
    } path_specific_stop_file_receipt_data_type;
```

Below: Receipt data from the Fleet Management Controller when using "View Packet Log", point number = 26



Result Code (Decimal)	Meaning	Recommended Response
0	Success, PSS file downloaded	None
1	CRC error	The Server should re-send the PSS file
4	File not transferred error	The Server should re-send the PSS file
5	PSS file contents error	See PSS Return Code (below), and send a valid PSS file
6	Invalid file type detected	Send a PSS file type
11	File contents error	***multi_use* field represents *PSS Return Code* in table below**
13	PSS busy, already processing	The Server should re-send the PSS file

NOTE: Table below only valid when *result_code* is set to 0 or 11 (decimal). This table indicates the PSS file parsing results.

PSS Return Code (Decimal)	Meaning	Recommended Response
0	Success, PSS file downloaded	No errors
1	Unexpected signature	Send a correct signature
2	Unexpected version	Send a correct version
3	Zero length Stop Text name	Include a Stop Text name in every PSS file
4	Nonprintable ASCII in Stop Text name	Send printable text in all text fields
5	Exceeded text maximum length	Send a text string within the maximum values
6	Invalid unique ID	Send ID's other than 0xFFFF FFFF
7	Non-printable ASCII in name. (Use *last point number* to determine error location in file. 0 indicates Stop Text name contains the error, any other value indicates the point entry containing the error.)	Send printable text in all text fields
8	Invalid Latitude value	Send a valid Latitude value
9	Invalid Longitude value	Send a valid Longitude value
10	Invalid point type (non-Shaping or non-Destination) found	Send valid point type values of (0 = Destination, 1 = Shaping)

11	First point type not a Destination	Send the first point type as a Destination Type
12	Last point type not a Destination	Send the last point type as a Destination Type
13	Exceeded maximum number of Shaping points	Send a maximum of 100 Shaping points between Destinations
14	Exceeded maximum number of Destination points	Send a maximum of 25 Destination points per Stop
15	Exceeded maximum total of Shaping and Destination points	Send less than the total maximum allowed points per Stop
16	Less than 2 destination points found	Send 2 or more destinations in file
17 or greater	Garmin specific errors	Contact Garmin support

Once the Path Specific Stop file has been processed, the Stop route information is processed. The Client calculates the Stop route distance in meters (starting from the first destination to the final destination), which is sent back to the Server (see Section 5.1.6.2.2.1).

Go to packet sequence table, click here> 5.1.13.1

5.1.13.1.6.5 Post Processing Receipt – A618 Stop file type

After the Client processes the A618 Stop file, the **Client to Server** type definition for the *path_specific_stop_file_receipt_data_type* is sent to the Server (shown below) to report the success status of the received A618 Stop.

```
typedef struct
    {
    uint8   result_code;   /* 0 = No transfer errors, non-zero shows error (see below) */
    uint8   multi_use      /* If result_code is NOT = 0 or 11 then it's a file_type */
                           /* If result_code = 0 or 11 then use Stop Return Code below */
    uint16  reserved;      /* Set to 0 */
    } a618_stop_file_receipt_data_type;
```

result_code (Decimal)	Meaning	Recommended Response
0	Success, Stop file downloaded	None
1	CRC error	Server should resend file
3	Invalid file	Server should resend file
4	File not transferred error	Server should resend file
5	File not transferred error	Server should resend file
6	Invalid file type detected	Send a Stop file type.
11	File contents error	***multi_use* field represents *Stop Return Code* from table below**
12	gZip error	Server should resend file
All others	Garmin specific errors	Contact Garmin support

NOTE: Table below only valid when *result_code* is set to 0 or 11 (decimal). This table indicates the Stop file parsing results.

Stop Return Code	Meaning	Recommended Response
0	Success, file processed	None
1	Unexpected signature	Send a correct signature
2	Unexpected version	Send a correct version
3	Zero length Stop Text name	Include a Stop Text name in every Stop file

5	Exceeded text maximum length	Send a text string within the maximum value
6	Invalid unique ID	Send ID's other than 0xFFFF FFFF
7	Invalid Latitude value	Send a valid Latitude value
8	Invalid Longitude value	Send a valid Longitude value
All others	Garmin specific errors	Contact Garmin support

Go to packet sequence table, click here> 5.1.13.1

5.1.13.2 Client to Server File Transfer Protocol

This protocol is used to send a file from the **Client to Server**. This protocol is essentially the same as **the Server to Client** File Transfer Protocol defined in Section 5.1.13.1, except the roles of the Client and Server are now reversed.

The Client initially divides the file data into small data packets (each packet contains Header data and up to 245 bytes of file data), and sends the data packets one by one to the Server using the protocol described below. The Server saves the data packet(s), and acknowledges each packet as it is received.

After the final data packet of the file is received by the Server, the CRC32 of the received file is checked. Lastly a final end-of-file acknowledgment packet is sent to the Client, which indicates the success of the complete file transfer.

Note: Currently, only the D610-AOBRD Event Log file of FMI Hours of Service (HOS), D612-Custom Forms Submit, and D616-IFTA Files can be transferred from the Client to the Server.

The File Transfer Protocol packet sequence is listed below as three distinct transfer stages. The first stage initiates the file transfer process:

N	Direction	Fleet Management Packet ID	Hyper-link	Fleet Management Packet Data Type
0	Client to Server	0x0400 – File Transfer Start Packet ID	5.1.13.2.1	file_info_data_type
1	Server to Client	0x0403 – File Start Receipt Packet ID	5.1.13.2.2	file_receipt_data_type

…next, repeat the following Server and Client data stage until all data is sent:

2..n-3	Client to Server	0x0401 – File Data Packet ID	5.1.13.2.3	file_packet_data_type
3..n-2	Server to Client	0x0404 – Packet Receipt Packet ID	5.1.13.2.4	packet_receipt_data_type

…all file data was sent, so end the file transfer sequence as shown below:

n-1	Client to Server	0x0402 – File Transfer End Packet ID	5.1.13.2.5	file_end_data_type
n	Server to Client	0x0405 – File End Receipt Packet ID	5.1.13.2.6	file_receipt_data_type

NOTE: Each of the packets in the table above will be discussed below. Go to the desired section by clicking on the Hyperlink.

5.1.13.2.1 Client to Server - Packet ID: 0x0400 - File Transfer Start

This *File Transfer Start* step initiates a file transfer from the **Client to Server**. The data needed to perform this step is shown in the type definition *file_info_data_type* below, and should be sent to the Server.

Note: Currently, only the D610-AOBRD Event Log file of FMI Hours of Service (HOS), D612-Custom Forms Submit, and D616-IFTA Files can be transferred from the Client to the Server.

```
typedef struct
    {
    uint32 file_size;
    uint8  file_version_length;
    uint8  file_type;           /* 1 = AOBRD Event, 2 = Custom Form submit */
    uint8  reserved[2];         /* Set to 0 */
    uint8  file_version[/* 16 bytes */];
    } file_info_data_type;
```

The *file_size* is the size of the file that will be transferred, in bytes. The maximum file size is limited by the amount of available space on the device. The *file_version* contains up to 16 bytes of arbitrary data to be associated with the file being transmitted, as a version number or for other purposes. The *file_version_length* indicates the number of bytes of *file_version* that are valid. The *file_type* should be set to indicate the file type.

Go to packet sequence table, click here> 5.1.13.2

5.1.13.2.2 Server to Client - Packet ID: 0x0403 - File Start Receipt

Once the Server correctly receives the *File Transfer Start* packet from the previous step, the Server responds back to the Client with a receipt packet. The receipt data type definition *file_receipt_data_type* shown below is created by the Server, and sent back to the Client. Note, the *file_type* from the *File Transfer Start* data is also included in the response.

```
typedef struct
    {
    uint8  result_code;
    uint8  file_type;           /* 1 = AOBRD Event, 2 = Custom Form submit, 4 = IFTA Files */
    uint8  reserved[2];         /* Set to 0 */
    } file_receipt_data_type;
```

The *file_type* will be identical to the value received in the File Transfer Start Packet ID. The *result_code* indicates whether the operation was successful. Result codes and their meanings are described in the table below. At minimum, the Client must differentiate between a result code of zero, indicating success, and a nonzero result code, indicating failure.

Result Code (Decimal)	Meaning	Recommended Response
0	Success	None
All other	Error	Abort the file transfer.

Go to packet sequence table, click here> 5.1.13.2

5.1.13.2.3 Client to Server - Packet ID: 0x0401 - File Data

After the Client receives the Server's *File Start Receipt* packet indicating no errors, the Client sends a File Data packet to the Server. This packet type contains the actual file data. The type definition for the *file_packet_data_type* is shown below.

Note: Currently, only the D610-AOBRD Event Log file of FMI Hours of Service (HOS), D612-Custom Forms Submit, and D616-IFTA Files can be transferred from the Client to the Server.

***Note:* The *file_data* is limited to 245 bytes. Files containing data larger than 245 bytes will require multiple file data packets to be sent.**

```
typedef struct
    {
    uint32 offset;             /* offset of this data from the beginning of the file */
    uint8  data_length;        /* length of file_data (0..245)   */
    uint8  file_type;          /* 1 = AOBRD Event, 2 = Custom Form submit, 4 = IFTA Files */
    uint8  reserved[2];        /* Set to 0 */
    uint8  file_data[245 max]; /* File data, 245 bytes maximum per packet, file_data[] greater */
                               /* than 245 bytes will require additional packets to be sent */
    } file_packet_data_type;
```

The *offset* indicates the position that should be written in the file, the first byte of the file has offset zero. The *data_length* indicates the number of bytes of file data that are being transmitted in this packet. The *file_type* will be identical to the value received in the File Transfer Start Packet ID. The *file_data* is the actual data to be written to the file.

***Note:* The Client must send file packets in ascending order by *offset*. The Client may vary *data_length* from packet to packet to suit the needs of the application.**

Go to packet sequence table, click here> 5.1.13.2

5.1.13.2.4 Server to Client - Packet ID: 0x0404 - Packet Receipt

The Server responds to *File Data* packets sent from the Client with a *Data Packet Receipt* packet. The type definition for the *packet_receipt_data_type* is shown below.

```
typedef struct
    {
    uint32 offset;               /* offset of data received */
    uint32 next_offset;          /* offset of next data the Server should send */
    } packet_receipt_data_type;
```

The *offset* is the offset of the file packet received, which is the same as the *offset* of the corresponding *file_packet_data_type* that contained the file data from the Client. The next_offset indicates the offset that the Client should use when sending the next chunk of file data. The *offset* and the *next_offset* are to be interpreted as follows:

- Normally, the *next_offset* will be equal to the sum of the *offset* and the *data_length* from the corresponding *file_packet_data_type*.
- If the *next_offset* is equal to the size of the file, all file data has been received.
- If the *next_offset* is less than the *offset*, the Server has rejected the file data, as the data beginning at *next_offset* has not yet been received.
- If the *next_offset* is equal to *offset*, a temporary error has occurred; the Client should resend the data packet.
- If the *next_offset* is equal to 0xFFFFFFFF hexadecimal (4294967295 decimal) a severe error has occurred; the transfer should be aborted.

These rules enable a simple mechanism for event-driven file transfer, when the Client receives the *packet_receipt_data_type*, it should send the file data beginning at *next_offset*, unless an error has occurred or the entire file has been sent. If this approach is taken, the Client should also check the offset received against the offset sent; if they do not match, the receipt packet should be ignored, as it indicates a delayed receipt after the Client retransmitted a packet.

Go to packet sequence table, click here> 5.1.13.2

5.1.13.2.5 Client to Server - Packet ID: 0x0402 - File Transfer End

The *File Transfer End* packet is sent by the Client when all of the file data has been successfully sent to the Server. This packet contains the file CRC value for the complete file that was sent by the Client, and the Server will now verify the CRC value against the received file.

```
typedef struct
    {
    uint32 crc;                /* CRC of entire file as computed by UTL_calc_crc32 */
    } file_end_data_type;
```

The *crc* is the CRC32 checksum of the entire file. The CRC32 algorithm is included in an Appendix (Section 6.4) and in electronic form in the *Fleet Management Interface Developer Kit.*

Go to packet sequence table, click here> 5.1.13.2

5.1.13.2.6 Server to Client - Packet ID: 0x0405 - File End Receipt

The *File End Receipt* packet is sent by the Server to the Client following the reception of *File Transfer End* packet from the Client. Any *result_code* other than '0' will invoke the Client to re-send the file to the Server. The format of the Server to Client *File End Receipt* is shown below:

```
typedef struct
    {
    uint8   result_code;           /* 0 = No transfer errors, non-zero shows error */
    uint8   file_type;             /* 1 = AOBRD Event, 2 = Custom Form submit, 4 = IFTA Files */
    uint8   reserved[2];           /* Set to 0 */
    } file_receipt_data_type;
```

Go to packet sequence table, click here> 5.1.13.2

5.1.13.3 File Information Protocol

This protocol allows the Server to retrieve information about files on the Client. Currently, only GPI files will provide information using this protocol.

5.1.13.3.1 GPI File Information

This protocol allows the Server to determine the size and version of the current Fleet Management GPI file on the device. The information returned will be for the last Fleet Management GPI file that was successfully transferred to the Client.

This protocol is only supported on Clients that report A604 as part of their protocol support data. The packet sequence for the GPI File Transfer Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0406 – GPI File Information Request Packet ID	None
1	Client to Server	0x0407 – GPI File Information Packet ID	file_info_data_type

This data type is only supported on Clients that report D604 as part of their protocol support data.

```
typedef struct  /*  D604  */
    {
    uint32 file_size;
    uint8  file_version_length;
    uint8  file_type;                   /* Set to 0 for GPI files */
    uint8  reserved[2];                 /* Set to 0 */
    uint8  file_version[/* 16 bytes */];
    } file_info_data_type;
```

The *file_size* is the size of the file currently in use on the device. If no file exists on the device, the *file_size* is zero. The *file_version* contains up to 16 bytes of version information sent by the Server during the GPI File Transfer Protocol. The *file_version_length* indicates the number of bytes of *file_version* that are valid. If no file exists on the device, or the file was not transferred via the Fleet Management Interface, the *file_version_length* will be zero.

5.1.14 Data Deletion Protocol

This protocol is used by the Server to delete data on the Client. This protocol is only supported on Clients that report A603 as part of their protocol support data. The packet sequence for the Data Deletion protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0230 – Data Deletion Packet ID	data_deletion_data_type

The type definition for the *data_deletion_data_type* is shown below.

```
typedef struct  /* D603 */
    {
    uint32      data_type;
    } data_deletion_data_type;
```

The value for the *data_type* corresponds to the type of data to be manipulated on the Client. The table below defines the values for *data_type*, along with the protocol support data required for the value.

Value (Decimal)	Meaning	Support
0	Delete all stops on the Client	D603
1	Delete all messages on the Client	D603
2	Delete the active navigation route on the Client.	D604
3	Delete all canned messages on the Client.	D604
4	Delete all canned replies on the Client. (All Server to Client Canned Response Text messages that have not been replied will become A604 Open text messages.)	D604
5	Delete the Fleet Management GPI file on the Client.	D604
6	Delete all driver ID and status information on the Client.	D604
7	**Delete the fleet management interface on the Client (this will delete all data relating to Fleet Management on the Client).**	D604
8	Delete all waypoints created through the Create Waypoint Protocol on the Client.	D607
9	Reserved	
10	Delete all Custom Forms on the Client	D612
11	Delete all Custom Avoidances on the Client	D613
12	Delete all Sensor Displays and Sensor Configurations	D617

5.1.15 User Interface Text Protocol

This protocol is used to customize the text of certain Fleet Management user interface elements. Currently, only the "Dispatch" text on the device main menu can be changed.

This protocol is only supported on Clients that report A604 as part of their protocol support data. The packet sequence for the User Interface Customization Protocol is shown below.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0240 – User Interface Text Packet ID	user_interface_text_data_type
1	Client to Server	0x0241 – User Interface Text Receipt Packet ID	user_interface_text_receipt_data_type

The type definition for the *user_interface_text_data_type* is shown below.

```
typedef struct  /*  D604  */
    {
    uint32 text_element_id;
    uchar_t8 new_text[];             /* variable length, null terminated, 50 bytes max */
    } user_interface_text_data_type;
```

The supported *text_element_ids* and their meanings are described in the table below. The *new_text* is the replacement text for that user interface element.

Element ID (decimal)	Meaning
0	"Dispatch" text on Client main menu

```
typedef struct  /*  D604  */
    {
    uint32  text_element_id;
    boolean result_code;
    uint8   reserved[3];     /* Set to 0 */
    } user_interface_text_receipt_data_type;
```

The *text_element_id* will be the same as that of the corresponding *user_interface_text_data_type*. The *result_code* indicates whether the text was updated successfully. It will be *false* if the *text_element_id* is not supported, or if the *new_text* is a null string.

5.1.16 Ping (Communication Link Status) Protocol

This protocol is used to send a "ping" to determine whether the communication link is still active.

This protocol is only supported on Clients that report A604 as part of their protocol support data. **Client to Server** pings are throttled by default on Clients that report A605 as part of their protocol support data. See the Message Throttling Protocols (Section 5.1.17) to enable the Ping protocol on these Clients.

The packet sequences for the Ping Protocol are shown below.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Client to Server	0x0260 – Ping Packet ID	None
1	Server to Client	0x0261 – Ping Response Packet ID	None

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0260 – Ping Packet ID	None
1	Client to Server	0x0261 – Ping Response Packet ID	None

5.1.17 Message Throttling Protocols

The Message Throttling protocols allow the Server to enable or disable certain Fleet Management protocols that are normally initiated by the Client, and determine which protocols are enabled and disabled. When a protocol is disabled, the Client will not initiate the protocol. However, user interface elements related to that protocol remain enabled. For example, if the **Client to Server** Open Text Message protocol is disabled, the user may still create a new text message, but the message will not actually be sent until the protocol is enabled again.

Note: Position, Velocity, and Time (PVT) packets (packet ID 51 decimal), are enabled and disabled using the PVT protocol. See Section 5.2.4 for more information.

5.1.17.1 Message Throttling Control Protocol

This protocol is used to enable or disable certain Fleet Management protocols that would normally be initiated by the Client.

The Message Throttling Control Protocol is only supported on Clients that report A604 as part of their protocol support data. The packet sequence for the Message Throttling Protocol is shown below.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0250 – Message Throttling Command Packet ID	message_throttling_data_type
1	Client to Server	0x0251 – Message Throttling Response Packet ID	message_throttling_data_type

The type definition for the *message_throttling_data_type* is described below.

```
typedef struct  /*  D604  */
    {
    uint16  packet_id;
    uint16  new_state;
    } message_throttling_data_type;
```

The *packet_id* identifies the first Fleet Management Packet ID in the packet sequence. Protocols that can be throttled, along with the corresponding packet ID, are listed in the table below. Clients that report A605 as part of their protocol support data will have certain protocols throttled by default, as listed below. Clients that report A604 but not A605 will have all protocols enabled by default.

Fleet Management Protocol	Packet ID (Hexadecimal)	Default State (A605)	Support
Message Status	0x0041	Enabled	D605
Refresh Canned Response Text	0x0034	Disabled	D605
Refresh Canned Message List	0x0054	Disabled	D605
Client to Server Open Text Message	0x0024	Enabled	D605
Stop Status	0x0211	Enabled	D605
Estimated Time of Arrival (ETA)	0x0201	Enabled	D605
Driver ID Update	0x0811	Enabled	D605
Driver Status List Refresh	0x0804	Disabled	D605
Driver Status Update	0x0821	Enabled	D605
Ping (Communication Link Status)	0x0260	Disabled	D605
Waypoint Deleted	0x0133	Disabled	D607

On the command packet, the *new_state* is one of the values from the table below. On the response packet, the *new_state* is a status which indicates whether the protocol is disabled or enabled after the command is processed.

New State (Decimal)	Meaning
0	Disable the specified protocol.

1	Enable the specified protocol.
65535	Error (invalid protocol ID or state)

5.1.17.2 Message Throttling Query Protocol

The Message Throttling Query Protocol is used to obtain the throttling state of all protocols that may be throttled.

The Message Throttling Query Protocol is only supported on Clients that report A605 as part of their protocol support data. The packet sequence for the Message Throttling Protocol is shown below.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0252 – Message Throttling Query Packet ID	none
1	Client to Server	0x0253 – Message Throttling Query Response Packet ID	message_throttling_list_data_type

The type definition for the *message_throttling_list_data_type* is described below.

```
typedef struct  /*  D605  */
    {
    uint16  response_count;
    message_throttling_data_type response_list[ /* one element for each protocol in the table
above, up to 60 */ ];
    } message_throttling_list_data_type;
```

The *response_count* is the number of elements in the *response_list* array. The *response_list* array contains one *message_throttling_data_type* element for each protocol that can be throttled, with *new_state* set to the current throttle status of the protocol. The Server should not expect *response_list* to be in any particular order.

5.1.18 FMI Safe Mode Protocol

The FMI Safe Mode Protocol is used to enable FMI Safe Mode (henceforth FMISM) and to set the threshold speed at which it will be enforced. Once the FMISM is turned on, it overrides the normal consumer safe mode and hides the "Safe Mode" setting. When the FMISM is turned off, the usual consumer safe mode setting becomes effective.

The following restrictions go into effect when the threshold speed is exceeded:

- The driver will be restricted from going to 'Dispatch' and 'Tools' menus
- If the driver is browsing a page descending from the 'Dispatch' or 'Tools' menus, the driver will be taken to the main map page
- The driver will not be able to read new stops or non-immediate text messages

The FMISM protocol is only supported on Clients that report A606 as part of their protocol support data. The packet sequence for the FMISM Protocol is shown below.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0900 –FMI Safe Mode Packet ID	fmi_safe_mode_data_type
1	Client to Server	0x0901 –FMI Safe Mode Response Packet ID	fmi_safe_mode_receipt_data_type

The type definition for the *fmi_safe_mode_data_type* is shown below.

```
typedef struct  /* D606 */
    {
    float32     speed;          /* in meters per second */
    } fmi_safe_mode_data_type;
```

To turn on the FMISM, set the *speed* to a positive decimal number in meters per second. The range of the *speed* is 0 to 5MPH(2.2352 m/s). If the set *speed* is greater than 5MPH, then the threshold *speed* will be set to 5MPH. To turn off the FMISM protocol, set a negative *speed*. The table below shows the effects of setting *speed* in different ranges.

Speed MPH(m/s)	Effect
Less than 0	Turn off FMI Safe Mode
Between 0 and 5(2.2352)	Turn on FMI Safe Mode
Greater than 5(2.2352)	Turn on FMI Safe Mode. Speed is set to 5(2.2352)

The type definition for the *fmi_safe_mode_receipt_data_type* is shown below.

```
typedef struct  /* D606 */
    {
    boolean     result_code;
    uint8       reserved[3];           /* Set to 0 */
    } fmi_safe_mode_receipt_data_type;
```

The *result_code* indicates whether the operation took effect on the Client device; it will be *true* if the FMISM operation is successful or *false* if an error occurred.

5.1.19 Speed Limit Alert Protocols

The Speed Limit Alert protocol (henceforth SLA) is used to alert the Server of speed limit violations. Once enabled, the device will begin monitoring vehicle speed, speed limits and send alerts during speeding events. If the device database does not contain the speed limit, it will behave as if the speed limit is arbitrarily large. Some PNDs allow the user to update a posted speed limit. In that case, only the original speed limits will be used by SLA.

The SLA protocol is only supported on Clients that report A608 as part of their protocol support data.

5.1.19.1 Speed Limit Alert Setup Protocol

SLA is off by default, awaiting a setup packet from the host. SLA settings are saved across power cycles. The packet sequence to setup A608 SLA is shown below.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1000 – Speed Limit Alert Setup	setup_data_type
1	Client to Server	0x1001 – Speed Limit Alert Setup Receipt	setup_receipt_data_type

The type definition for the *setup_data_type* is shown below.

```
typedef struct  /* D608 */
    {
    uint8    mode;
    uint8    time_over;
    uint8    time_under;
    uint8    alert_user;
    float32  threshold;
    } setup_data_type;
```

Mode is used to enable or disable SLA. Car and truck speed limits can be different, therefore an option to specify either one is provided. The table below shows all allowed values for *mode*.

Mode	Meaning
0	Car
1	Off
2	Truck

Time_over is the time in seconds since *threshold* is exceeded after which speeding event starts. *Time_under* is the time in seconds since speed in decreased below the *threshold* after which speeding event ends. *Alert_user* denotes whether the driver is to be notified with an audible tone when the speeding event starts. *Threshold* is the speed in meters per second above (positive) or below (negative) speed limit after which the driver is considered speeding.

Note: Negative threshold use is recommended for testing purposes only.

The type definition for the *setup_receipt_data_type* is shown below.

```
typedef struct  /* D608 */
    {
    uint8  result_code;
    uint8  reserved[3];         /* Set to 0 */
    } setup_receipt_data_type;
```

Result_code contains the result. The table below shows all possible values for *result_code*.

result_code	Meaning
0	Success
1	Error
2	Unsupported mode

5.1.19.2 Speed Limit Alert Protocol

A speeding event starts when the speed *threshold* is exceeded for *time_over* seconds, and ends when speed drops below *threshold* for *time_under* seconds. The packet sequence for SLA alerts is shown below.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Client to Server	0x1002 – Speed Limit Alert	alert_data_type
1	Server to Client	0x1003 – Speed Limit Alert Receipt	alert_receipt_data_type

The type definition for the *alert_data_type* is shown below.

```
typedef struct  /* D608 */
    {
    uint8             category;
    uint8             reserved[3];     /* set to 0 */
    sc_position_type  position;
    time_type         timestamp;
    float32           speed;
    float32           speed_limit;
    float32           max_speed;
    } alert_data_type;
```

If SLA is turned off, or any of the settings are changed during a speeding event, an alert of 'Invalid' *category* will be sent. For alerts of 'Error' and 'Invalid' categories, only the category value is significant, and all alerts since last 'Begin', should be deemed invalid. The table below shows all of the possible values for category.

Category	Meaning
0	Begin – Speeding event began
1	Change – Speed limit changed
2	End – Speeding event ended

3	Error – Internal error
4	Invalid – Invalidate speeding event

Position is a semicircle position at the time of the alert. *Timestamp* is the time at the time of the alert. *Speed* is the speed in meters per second at the time of the alert. *Speed_limit* is the speed limit at the time of the alert. An arbitrarily large, i.e. 2,000MPH value indicates there is currently no speed limit in the device database. *Max_speed* is the maximum speed in meters per second achieved since the last alert. In the case of an alert of 'Begin' *category*, *max_speed* is the maximum speed achieved since the threshold was broken.

After receiving an alert packet, the host needs to respond with a receipt packet. The *timestamp* must be the same as the alert being confirmed. In case no receipt packet is received, up to 50 alerts will be queued. When the 51st alert happens, it will be discarded and SLA will be reset to ready state. If reset happens during a speeding event, then all of the alerts for the current speeding event will be removed from the queue. If this results in clearing of the whole queue then an alert of type 'invalid' is added to the queue.

The type definition for the *alert_receipt_data_type* is shown below.

```
typedef struct  /* D608 */
    {
    time_type  timestamp;
    } alert_receipt_data_type;
```

5.1.20 A609 Remote Reboot

This protocol allows the Server to request a Client side Remote Reboot, which should only be performed if the Client device is no longer responding to a user. The execution of this request could potentially lose Client data and should be used as a last option to regain use of a Client. The Server will request a Remote Reboot using the packet below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1010 – Remote Reboot Request	None

5.1.21 Custom Form Protocols

The Custom Forms protocols allow the Server to send Server-defined form templates to the device and to reorder and delete those templates. The form templates are used to create forms that can be completed and submitted back to the Server. The Garmin Fleet Management Development kit contains the following Custom Form Template examples, which can be sent to the Client:

1. *Example_Customer_Trip_Template.xml*
2. *Example_Truck_Inspection_Template.xml*

5.1.21.1 A612 Custom Form file format

Custom Form files consist of XML data, and follow a specific format as described in the Garmin Custom Form XML Schema Definition (XSD). A copy of the Garmin XSD can be found in the Garmin Fleet Management Development kit: http://developer.garmin.com/lbs/fleet-management/

Custom Form XML files can potentially be very large, so the GZIP file compression method was implemented for all Custom Form file transfers (from the Server, and to the Server) to optimize file transmission/reception. Each XML file must be compressed into the GZIP file compression format in order to be sent or processed.

NOTE: All Custom Form XML files should be transferred in the GZIP file format.

5.1.21.2 A612 Custom Form Template send to Client Protocol

This protocol allows the Server to send a *Custom Form Template* to the Client. The same file transfer mechanism is utilized for Custom Forms as used by GPI and AOBRD driver log downloads to the Client. A Custom Form Template can be sent to the Client any time that no other files are being sent to the Client.

Any error detected during a download will halt the Custom Form Template file transfer process, and that form will not be saved by the Client. Once the template is downloaded to the Client, the form will be analyzed for format errors. Two distinct levels of result codes will be sent back to the Server (**one code to indicate the success of the file download process** and **one code to indicate the file format/content examination results**) in the *File End Receipt Packet* of Section 5.1.13.1.6.

This process is described in the File Transfer protocol section, click here> 5.1.13.1

5.1.21.3 A612 Custom Form submit to Server Protocol

This protocol allows the Client to submit a driver completed *Custom Form* to the Server using a generic File Transfer mechanism also utilized by the D610 AOBRD protocol. This protocol is invoked when the driver presses the "Submit" user interface button. If the Server is not available, the Client will continue to attempt file transmission until the Custom Form is successfully sent.

Go to Client to Server File Transfer described in the File Transfer protocol section, click here> 5.1.13.2

5.1.21.4 A612 Custom Form Template delete on Client Protocol

The Custom Form Template delete request protocol allows the Server to delete a Custom Form Template on the Client device. The Server indicates which Custom Form Template to delete by setting the *form_id* in the *custom_form_delete_to_client_type* structure sent to the Client.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1200 – Delete Form Request Packet ID	custom_form_delete_to_client_type
1	Client to Server	0x1201 – Delete Form Receipt Packet ID	custom_form_delete_to_server_type

```
typedef struct  /*  D612  */
    {
    uint32   form_id;                    /* Form Template ID to delete on client */
    } custom_form_delete_to_client_type;
```

Client to Server packet receipt contains a template delete receipt packet that includes the *form_id* and a *return_code* to indicate the success of the request using the *custom_form_delete_to_server_type* structure shown below.

Result Code (Decimal)	Meaning	Recommended Response
0	Success, Custom Form Template deleted	None
All other	Internal Client error during Server request	The Server should try again later.

```
typedef struct  /*  D612  */
    {
    uint32  form_id;            /* Form Template ID to delete on client */
    uint8   return_code;        /* 0 = Form Template deleted, non-zero indicates error */
    } custom_form_delete_to_server_type;
```

5.1.21.5 A612 Custom Form Template move position on Client Protocol

The Custom Form Template move position request protocol allows the Server to alter the current position of a Custom Form Template as it is listed on the Client device. If the requested position is currently occupied by another Template then the Form Template will be inserted into the list at the position specified by the Server, and could affect the position of other Template Forms due to an insertion into an existing Template list.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1202 – Move Position Request Packet ID	move_form_position_request_to_client_type
1	Client to Server	0x1203 – Move Position Receipt Packet ID	form_position_receipt_to_server_type

Server to **Client** packet contains the *move_form_position_request_to_client_type* to move a template position .

```
typedef struct  /*  D612  */
    {
    uint32    form_id;               /* Form Template ID to reposition on client */
    uint32    move_to_position;      /* New list position on client */
    } move_form_position_request_to_client_type;
```

Client to Server packet response contains *form_id*, *current_position* and a *return_code* to indicate the success of the request shown below. The *form_position_receipt_to_server_type* receipt structure is also used for the Custom Form Template position request protocol described in this document.

```
typedef struct  /*  D612  */
    {
    uint32  form_id;              /* Form Template ID to move on client */
    uint8   current_position;     /* Current list position on client */
    uint8   return_code;          /* 0 = Form Template repositioned, non-zero indicates error */
    } form_position_receipt_to_server_type;
```

Return Code (Decimal)	Meaning	Recommended Response
0	Success	None
16	Form Template ID not found	The Server should send an ID that exists on Client
All other	Internal Client error during Server request	The Server should try again later.

5.1.21.6 A612 Custom Form Template position request on Client Protocol

The Custom Form Template position request protocol allows the Server to find the current position of a Custom Form Template as it is listed on the Client device.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1204 – Form Position Request Packet ID	custom_form_position_request_to_client_type
1	Client to Server	0x1205 – Form Position Receipt Packet ID	form_position_receipt_to_server_type

Server to **Client** packet contains the *request custom_form_position_request_to_client_type*, which includes the *form_id* of a particular Custom Form (as shown below).

```
typedef struct  /*  D612  */
    {
    uint32  form_id;          /* Form Template ID position on client */
    } custom_form_position_request_to_client_type;
```

Client to Server receipt position receipt packet includes the *form_id*, the *current_position* and a *return_code* to indicate the success of the request (as shown below). The *form_position_receipt_to_server_type* receipt structure is also used for the Custom Form move position request protocol described in this document.

```
typedef struct  /*  D612  */
    {
    uint32  form_id;           /* Form Template ID for position on client */
    uint8   current_position;  /* Position of Form Template on client */
    uint8   return_code;       /* 0 = Template position reported, non-zero indicates error */
    } form_position_receipt_to_server_type;
```

Result Code (Decimal)	Meaning	Recommended Response
0	Success	None
16	Form Template ID not found	The Server should send an ID that exists on Client
All other	Internal Client error during Server request	The Server should try again later.

5.1.22 Custom Avoidance Protocols

The A613 Custom Avoidance Protocols are used to allow the Server to define areas that Garmin navigation should avoid when creating routes for the driver. These areas can be create, modified and deleted as well as being enabled or disabled by the Server.

5.1.22.1 A613 Custom Avoidance Area Feature Enable Protocol

This protocol allows the server to enable or disable the Custom Avoidance Area Feature.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1236 – Custom Avoidance Area Feature Packet ID	custom_avoid_feature_enable_type
1	Client to Server	0x1237 – Custom Avoidance Area Feature Receipt Packet ID	custom_avoid_feature_enable_type

Server to **Client** packet enables/disables the Custom Avoidance protocol by sending the type definition for the *custom_avoid_feature_enable_type* as shown below:

```
typedef struct  /*  D613  */
    {
    time_type    origination_time;      /* Time sent from Server */
    boolean      enable;                /* 0 = Disable, 1 = Enable Custom Avoidance feature */
    } custom_avoid_feature_enable_type;
```

Client to Server receipt packet indicates the success of enabling/disabling the Custom Avoidance Feature by returning the *custom_avoid_feature_enable_type* back to the Server as shown above.

5.1.22.2 A613 Custom Avoidance New/Modify Protocol

This protocol allows the server to create or modify a custom avoidance on the client when the "Custom Avoidance Feature" is enabled (see the "Custom Avoidance Area Feature Enable protocol" in Section 5.1.22.1).

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1230 –Custom Avoidance Create Packet ID	custom_avoid_type
1	Client to Server	0x1231 – Custom Avoidance Create Receipt Packet ID	custom_avoid_rcpt_type

Server to **Client** packet creates or modifies an existing Custom Avoidance area by sending the *custom_avoid_type* structure to the Client. The type definition for the *custom_avoid_type* is shown below:

```
typedef struct  /*  D613  */
    {
    scposn_type  point1;          /* Coordinates for Northeast corner */
    scposn_type  point2;          /* Coordinates for Southwest corner */
    uint16       unique_id;       /* Server-assigned unique ID for the avoidance */
    boolean      enable;          /* 0 = disable, 1 = enable the custom avoidance */
    uint8        reserved;        /* Set to 0 */
    uchar_t8     name[49];        /* 49 bytes, null-terminated */
    } custom_avoid_type;
```

Client to Server receipt packet indicates the success of creating or modifying a Custom Avoidance. The type definition for the *custom_avoid_rcpt_type* is shown below:

```
typedef struct  /*  D613  */

    {
    uint16 unique_id;             /* Server-assigned unique ID */
    uint8  result_code;           /* 0 = success, non-zero indicates error (see below) */
    } custom_avoid_rcpt_type;
```

The expected values for *result_code* are listed below:

Value (Decimal)	Result Code
0	Avoidance Area was added
1	Could not found unique Id.
2	Avoidance List database full
3	Could not save to database
4	Avoidance Name currently in-use
5	Custom Avoidance feature is not enable

5.1.22.3 A613 Custom Avoidance Delete Protocol

This protocol allows the server to delete a custom avoidance on the client.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1232 –Custom Avoidance Delete Packet ID	custom_avoid_delete_type
1	Client to Server	0x1233 – Custom Avoidance Deleted Packet ID	custom_avoid_deleted_type

Server to Client packet deletes an existing Custom Avoidance area by sending the *custom_avoid_delete_type* structure to the Client (as shown below):

```
typedef struct  /*  D613  */
    {
    uint16    unique_id;            /* Server-assigned unique ID to be deleted */
    } custom_avoid_delete_type;
```

Client to Server receipt packet indicates the success of deleting a Custom Avoidance. The type definition for the *custom_avoid_deleted_type* is shown below:

```
typedef struct  /*  D613  */
    {
    uint16 unique_id;              /* Server-assigned unique ID */
    uint8  result_code;            /* 0 = Avoidance Area deleted, 1 = Unique ID not found */
    } custom_avoid_deleted_type;
```

The expected values for *result_code* are listed below:

Value (Decimal)	Result Code
0	Avoidance Area was deleted
1	Unique ID not found

5.1.22.4 A613 Custom Avoidance Enable/Disable Protocol

This protocol allows the server to enable or disable an existing Custom Avoidance on the client.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1234 –Custom Avoidance Enable/Disable Packet ID	custom_avoid_enable_type
1	Client to Server	0x1235 – Custom Avoidance Enabled/Disabled Packet ID	custom_avoid_enable_rcpt_type

Server to Client packet enables/disables an existing Custom Avoidance area by sending the *custom_avoid_enable_type* (as shown below):

```
typedef struct  /*  D613  */
    {
    uint16    unique_id;   /* Server-assigned unique ID for the custom avoidance to be enabled */
    boolean   enable;      /* 0 = disable, 1 = enable a custom avoidance *
    } custom_avoid_enable_type;
```

Client to Server receipt packet indicates the success of enabling/disabling an existing Custom Avoidance. The type definition for the *custom_avoid_enable_rcpt_type* is shown below:

```
typedef struct  /*  D613  */
    {
    uint16          unique_id;       /* Server-assigned unique ID */
    uint8           result_code;     /* 0 = Success, non-zero indicates error */
    } custom_avoid_enable_rcpt_type;
```

5.1.23 A616 Set Baud Rate Protocol

This protocol allows the Server to select a RS232 serial baud rate on the Client. Currently the Client default baud rate is set to 9,600. After the Client (FMI) is enabled using the existing Enable Fleet Management Protocol, the Server has the option to modify the Client's RS232 baud rate at any time.

Note: It is recommended to limit the sending of any other protocols when using the Set Baud Rate Protocol until a successful re-sync of communication has been established (e.g., no PVT, no Ping packet traffic).

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0011 – Baud Rate Request	baud_rate_request_data_type
1	Client to Server	0x0012 – Baud Rate Receipt	baud_rate_receipt_data_type

Server to Client *Set Baud Rate* Protocol packet contains a Change Baud Rate *request_type* and the desired *baud_rate* members (shown below). The Server would then wait for a Client response before the Server switches its own baud rate.

```
typedef struct  /*  D616  */
    {
    uint8    request_type;          /* 0 = Change baud rate request, 1 = SYNC request
    uint8    baud_rate;             /* 0x06 = 9600, 0x0c = 57600 */
    } baud_rate_request_data_type;
```

request_type	Meaning
0	Set baud rate
1	Sync with Client (sent after rate change)

baud_rate (Decimal)	Meaning
6	9,600 baud rate (default)
12	57,600 baud rate

The **Client to Server** receipt response contains the echoed request data from the Server along with a *result_code* to indicate if a request error was detected by the Client using the *baud_rate_receipt_data_type* packet (shown below).

```
typedef struct  /*  D616  */
    {
    uint8    result_code            /* result code */
    uint8    request_type;          /* 0 = Change baud rate request, 1 = SYNC request
    uint8    baud_rate;             /* 0x06 = 9600, 0x0c = 57600 */
    } baud_rate_receipt_data_type;
```

result_code	Meaning
0	Success
1	Invalid request type
2	Invalid baud rate
All others	Internal error, try again later

Note: When the Server receives a success response from the Client for a baud rate change request, the Server should then switch its own baud rate, and begin sending the "Sync"* request_type *at the new baud rate.

The **Server to Client** "Sync" request uses the same *baud_rate_request_data_type* packet (shown above), but now the *request_type* member should contain the "Sync with Client" value (see table above). **This "Sync" request type packet can be sent to the Client at any time to verify communication**, but it is required to **ALWAYS** be sent after a Server baud rate change to cancel the Client's "30-second baud switch watchdog timer".

Note: If the Client switches rate and does not receive the Server "Sync" packet within 30 seconds, the Client will ALWAYS switch back to the default 9,600 baud rate. This ensures the Server/Client can re-establish communication if some unexpected switch error occurs by either end.

Client to Server "Sync" receipt *baud_rate_receipt_data_type* packet (shown above) informs the Server that Client "Sync" has been established. Following the reception of this Client response, the Server should be able to resume normal FMI packet exchanges.

Note: If at any time, the Client detects the loss of FMI communication through the serial communication port, which causes the FMI Connection error icon to appear on the Client's user interface screen, the Client will automatically switch back to the 9,600 baud rate.

5.1.24 A617 Alert Popup Protocol

The Alert Pop-up protocol allows the Server to send a pop-up alert to the User Interface on the Client's device. Once the pop-up appears on the User Interface, the driver can press anyplace on the User Interface to remove it.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1400 – Alert Pop-up	alert_popup_type
1	Client to Server	0x1401 – Alert Pop-up Receipt	alert_popup_reciept_type

Server to **Client** type definition for *alert_popup_type* is shown below:

```
typedef struct  /* D617 */
    {
    uint16    unique_id;      /* unique identifier for the alert popup */
    uint16    icon;           /* Icon selected from Garmin pre-loaded icons */
    uint8     timeout;        /* 1 to 15 (seconds), values above 15 default to 15, 0 = 2 hours */
    uint8     severity;       /* 0 = normal, 1 = medium, 2 = high  */
    boolean   play_sound;     /* Play sound on device, 0 = No, 1 = Yes */
    char      alert_text[up to 110]; /* Alert text, 110 bytes including null terminator */
    } alert_popup_type;
```

The Alert Pop-up structure contains a *unique_id* to identify this Alert Pop-up. The *icon* field selects an existing Garmin icon (listed in the table below) on the Client's User Interface. The available icons are:

Value (Decimal)	Icon
0	No icon needed
1	Driver Behavior
2	Tire Pressure
3	Temperature
4	Door Ajar
5	Vehicle Maintenance
6	OBD-II Generic Sensor
7	Generic Sensor 1
8	Generic Sensor 2
9	Generic Sensor 3
10	No signal
11	Day Hours counter

Value (Decimal)	Icon
12	Weekly Hours counter
13	Rest Hours counter
14	Break Hours counter
15	Dispatcher Tasks
16	Weight
17	Information
18	Fuel
19	Available (EU symbol)
20	Driving (EU symbol)
21	Rest (EU symbol)
22	Working (EU symbol)

The *timeout* field determines the number of seconds the pop-up will remain on the Client's User Interface for the driver. **The driver can remove the pop-up by pressing anyplace on the User Interface**. The setting range is 1 to 15 (seconds). Any value received above 15 will use a default setting of 15 (seconds).

Note: A Server *timeout* setting of "0" will cause the pop-up to remain on the User Interface for 2 hours (or until a driver presses anyplace on the User Interface).

The *severity* field controls the pop-up color when displayed. The *play_sound* field determines if a pre-recorded sound should be played when this packet is received. The *alert_text* string contains the text displayed on the pop-up.

Client to **Server** type definition for *alert_popup_receipt_type* is shown below:

```
typedef struct  /* D617 */
    {
    uint32  unique_id;      /* unique identifier from the server request */
    uint8   result_code;    /* 0 = success, non-zero indicates error */
    } alert_popup_receipt_type;
```

The Alert Pop-up contains the *unique_id* sent from the Server. The *result_code* table is listed below:

result_code	Meaning
0	Success
1	Alert name too long

2	Alert icon range error
3	No Alert name
4	Severity out of range
5	Timeout range error

5.1.25 A617 Sensor Display Protocols

The A617 Sensor Display protocol allows the Server or black-box to send sensor information to the Client device, which can be displayed to the driver. The Server has the ability to configure, update and delete a sensor. A sensor's history containing the last 24 data updates will be maintained on the Client device and available for the driver on demand via the Client User Interface.

Note: The Server may use this protocol to send and display any other data on the Client's User Interface (not only sensor data).

As mentioned above, the Sensor functionality is segregated into three distinct types of operation, and listed below:

- Configure Sensor – Create a sensor display object on the Client's User Interface
- Update Sensor – New sensor data to be displayed or recorded on the Client User Interface
- Delete Sensor – Remove a configured sensor from the Client's User Interface
- Sensor Display List Position – Find a sensor's position in the sensor list

5.1.25.1 A617 Configure Sensor Display Protocol

The Configure Sensor Display Protocol allows a Server to create a new sensor, or modify the configuration of an existing sensor on the Client's User Interface.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1402 – Configure Sensor Display	sensor_config_display_type
1	Client to Server	0x1403 – Configure Sensor Display Receipt	sensor_config_display_receipt_type

Server to Client type definition for *sensor_config_display_type* is shown below:

```
typedef struct  /* D617 */
    {
    uint32  change_id;     /* Unique identifier for this Configure Sensor Display packet */
    uint32  unique_id;     /* Unique identifier for this sensor display */
    uint16  icon;          /* Icon selected from Garmin preloaded icons (see list below) */
    uint8   display_index; /* Sensor sorting index used to order sensor, 1-based, 0 is invalid */
    uint8   reserved[3];   /* set to 0 */
    char    name[1 to 40]; /* Sensor name, 40 bytes max including null terminator */
    } sensor_config_display_type;
```

The Configure Display packet contains a *change_id* that identifies a unique sensor packet. A *unique_id* indicates a particular sensor. When modifying the configuration of an existing sensor, the same *unique_id* value must always be used.

The *icon* field selects an existing Garmin icon on the Client's User Interface, and listed below:

Value (Decimal)	Icon		Value (Decimal)	Icon
0	No icon needed		12	Weekly Hours counter
1	Driver Behavior		13	Rest Hours counter
2	Tire Pressure		14	Break Hours counter
3	Temperature		15	Dispatcher Tasks
4	Door Ajar		16	Weight

5	Vehicle Maintenance		17	Information
6	OBD-II Generic Sensor		18	Fuel
7	Generic Sensor 1		19	Available (EU symbol)
8	Generic Sensor 2		20	Driving (EU symbol)
9	Generic Sensor 3		21	Rest (EU symbol)
10	No signal		22	Working (EU symbol)
11	Day Hours counter			

The sensor will be displayed in a sensor list for the driver, and the list position of this sensor will be dictated by the *display_index*, which allows the Server to control (or adjust the list order of) the sensor when displayed for the driver. If a *display_index* value is greater than the current last sensor shown then the sensor will be displayed directly at the end of the sensor list. Sensors will always be shown packed in one contiguous list (no empty or blank lines) regardless if *display_index* value gaps occur between sensors.

The *name* field is a character string displayed for this sensor.

Once a sensor is successfully configured, the *Update Sensor Display* or *Delete Sensor Display* protocols can be used for that sensor.

Note: A maximum of 16 Sensor Displays can be configured on a Client device.

Client to **Server** type definition for *sensor_config_display_receipt_type* is shown below:

```
typedef struct  /* D617 */
    {
    uint32  change_id;      /* Unique identifier from the Server request */
    uint8   result_code;    /* 0 = Success, Non-zero indicates error */
    uint8   operation_mode; /* 1 = Inserted in table, 2 = Modified existing sensor */
    } sensor_config_display_receipt_type;
```

The receipt packet contains the same *change_id* from the Server. The *result_code* table is listed below.

Result code	Meaning
0	Success
1	Sensor name too long
2	Sensor icon range error
3	No Sensor name
4	Severity out of range
5	Status string too long
6	Description too long
7	Too many sensors
8	Sensor ID not found
All others	Contact Garmin Support

5.1.25.2 A617 Update Sensor Display Status Protocol

The A617 Update Sensor Display Status protocol receives Server or black-box data, which is used to update the Sensor User Interface and sensor history for a driver. The received data could have originated or been triggered from virtually any source (e.g., tire sensor, temperature sensor, trailer door sensor, RFID reader, remote data source).

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1406 – Update Sensor Display Status	update_sensor_display_status_type
1	Client to Server	0x1403 – Update Sensor Display Status Receipt	update_sensor_display_status_receipt_type

Server to **Client** type definition for *update_sensor_display_status_type* is shown below:

```
typedef struct  /* D617 */
    {
    uint32  change_id;      /* Unique identifier for this Update Status Display packet */
    uint32  unique_id;      /* Unique identifier of a sensor, used during sensor configure */
    uint8   severity;       /* 0 = Normal, 1 = Medium, 2 = High  */
    boolean play_sound;     /* Trigger sound now, 0 = No, 1 = Yes */
    boolean record_sensor;  /* Record to this sensor's history log, 0 = No, 1 = Yes */
    uint8   reserved[3];    /* Set to 0 */
    char    status[80];     /* 80 bytes including null terminator */
    char    description[up to 110];  /* 110 bytes including null terminator */
    } update_sensor_display_status_type;
```

The Update Sensor Display Status structure contains a *change_id* that identifies a unique sensor packet. A *unique_id* indicates a particular sensor (and must be the same *unique_id* value used during sensor configuration). The *severity* level controls the sensor's color when displayed in the sensor list. The *play_sound* field determines if a pre-recorded sound should be played when this packet is received.

Client to Server type definition for *update_sensor_display_status_receipt_type* is shown below:

```
typedef struct  /* D617 */
    {
    uint32  change_id;      /* Unique identifier from Server request packet */
    uint8   result_code;    /* 0 = Success, Non-zero indicates error, see table below */
    uint8   operation_mode; /* 2 = Update sensor status requested */
    } update_sensor_display_status_receipt_type;
```

The receipt packet contains the same *change_id* from the Server. The *result_code* table is listed below.

The *operation_mode* indicates a "modify" operation was requested.

Result code	Meaning
0	Success
1	Sensor name too long
2	Sensor icon range error
3	No Sensor name
4	Severity out of range
5	Status string too long
6	Description too long
7	Too many sensors
8	Sensor ID not found
All others	Contact Garmin Support

5.1.25.3 A617 Delete Sensor Display Protocol

The Delete Sensor Display protocol allows the Server to delete a previously configured sensor display on the Client device.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1404 – Delete Sensor Display	delete_sensor_display_type
1	Client to Server	0x1405 – Delete Sensor Display Receipt	delete_sensor_display_receipt_type

Server to Client type definition for *delete_sensor_display_type* is shown below:

```
typedef struct  /* D617 */
    {
    uint32  change_id;      /* Unique identifier for this Delete Sensor packet */
    uint32  unique_id;      /* Unique identifier of a sensor, used during sensor configure */
    } delete_sensor_display_type;
```

The Delete Sensor structure contains a *change_id* that identifies a unique sensor packet. A *unique_id* indicates a particular sensor (and must be the same *unique_id* value used during sensor configuration).

Client to **Server** type definition for *delete_sensor_display_receipt_type* is shown below:

```
typedef struct  /* D617 */
    {
    uint32  change_id;      /* Unique identifier used from Server request */
    uint8   result_code;    /* 0 = Success, Non-zero indicates error */
    uint8   operation_mode; /* 0 = Delete sensor requested */
    } delete_sensor_display_receipt_type;
```

The receipt packet contains the same *change_id* from the Server. The *result_code* table is listed below.

The *operation_mode* indicates a "delete" sensor operation was requested.

Result code	Meaning
0	Success
8	Sensor ID not found
All others	Contact Garmin Support

5.1.25.4 A617 Sensor Display List Position Protocol

The Sensor Display List Position Request protocol allows the Server to request the position of a sensor in the current sensor list.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1407 –Sensor Display List Position	sensor_display_position_type
1	Client to Server	0x1408 – Sensor Display List Position Receipt	sensor_display_position_receipt_type

Server to Client type definition for *sensor_display_position_type* is shown below:

```
typedef struct  /* D617 */
    {
    uint32  change_id;      /* Unique identifier for this List Position packet */
    uint32  unique_id;      /* Unique identifier of a sensor, used during sensor configure */
    } sensor_display_position_type;
```

The Sensor List Position Request structure contains a *change_id* that identifies a unique sensor packet. The *unique_id* indicates a particular sensor (and must be the same *unique_id* value used during sensor configuration).

Client to Server type definition for *sensor_display_position_receipt_type* is shown below:

```
typedef struct  /* D617 */
    {
    uint32  change_id;      /* Unique identifier from the server request */
    uint8   result_code;    /* 0 = Success, Non-zero indicates error */
    uint8   display_index;  /* Indicates position in sensor list this sensor will appear */
    } sensor_display_position_receipt_type;
```

The receipt packet contains the same *change_id* from the Server. The *result_code* table is listed below. The *display_index* indicates the list position a driver would view using the Client's User Interface sensor list.

Result code	Meaning
0	Success
8	Sensor ID not found
All others	Contact Garmin Support

5.2 Other Relevant Garmin Protocols

All the protocols described in this section are Garmin protocols that are supported on all Garmin devices that support fleet management. The protocols are not related to the fleet management, but can prove to be very useful in having a complete fleet management system design.

5.2.1 Command Protocol

This section describes a simple protocol used in commanding the Client or Server to do something (e.g. send position data). For a list of command IDs relevant to this document refer to Appendix 6.3. The link layer packet for the command protocol would be represented as shown below.

Byte Number	Byte Description	Notes
0	Data Link Escape	16 (decimal)
1	Packet ID	10 (decimal)
2	Size of Packet Data	2
3-4	Command ID	See Appendix 6.3
5	Checksum	2's complement of the sum of all bytes from byte 1 to byte 4
6	Data Link Escape	16 (decimal)
7	End of Text	3 (decimal)

5.2.2 Unit ID/ESN Protocol

This protocol is used to extract the Client's unit ID (or electronic serial number). The packet sequence for this protocol is shown below.

N	Direction	Packet/Command ID	Packet Data Type
0	Server to Client	14 – Request Unit ID Command ID	No data (command)
1	Client to Server	38 – Unit ID Packet ID	unit_id_data_type

The type definition for the *unit_id_data_type* is shown below.

```
typedef struct
    {
    uint32                  unit_id;
    uint32                  other_stuff[ /* variable length */ ];
    } unit_id_data_type;
```

The *unit_id* is the first 32-bits of the data type. Some Clients could append additional information used by Garmin manufacturing. That data should be ignored.

5.2.3 Date and Time Protocol

The Date and Time protocol is used to transfer the current date and time on the Client to the Server. The packet sequence for this protocol is shown below.

N	Direction	Packet/Command ID	Packet Data Type
0	Server to Client	5 – Request Date/Time Data Command ID	No data (command)
1	Client to Server	14 – Date/Time Data Packet ID	date_time_data_type

The type definition for the *date_time_data_type* is shown below.

```
typedef struct
    {
    struct
        {
        uint8   month;    /* month (1-12)                 */
        uint8   day;      /* day (1-31)                   */
        uint16  year;     /* Real year (1990 means 1990!) */
        } date;
    struct
        {
        sint16  hour;     /* hour (0-65535), range required for correct ETE conversion */
        uint8   minute;   /* minute (0-59) */
        uint8   second;   /* second (0-59) */
        } time;
    } date_time_data_type;
```

Note: The *date_time_data_type* differs from the *time_type* used throughout the rest of this document.

5.2.4 Position, Velocity, and Time (PVT) Protocol

The PVT Protocol is used to provide the Server with real-time position, velocity, and time (PVT), which is transmitted by the Client approximately once per second. The Server can turn PVT on or off by using a Command Protocol (see Appendix 6.3). ACK and NAK packets are optional for this protocol; however, unlike other protocols, the Client will not retransmit a PVT packet in response to receiving a NAK from the host. The packet sequence for the PVT Protocol is shown below:

N	Direction	Packet/Command ID	Packet Data Type
0	Server to Client	49 – Turn On PVT Data Command ID or 50 – Turn Off PVT Data Command ID	No data (command)
1	Client to Server	51 – PVT Data Packet ID	pvt_data_type

The type definition of the *pvt_data_type* is shown below.

```
typedef struct
    {
    float32                 altitude;
    float32                 epe;
    float32                 eph;
    float32                 epv;
    uint16                  type_of_gps_fix;
    float64                 time_of_week;
    double_position_type    position;
    float32                 east_velocity;
    float32                 north_velocity;
    float32                 up_velocity;
    float32                 mean_sea_level_height;
    sint16                  leap_seconds;
    uint32                  week_number_days;
    } pvt_data_type;
```

The *altitude* member provides the altitude above the WGS 84 ellipsoid in meters. The *mean_sea_level_height* member provides the height of the WGS 84 ellipsoid above mean sea level at the current *position*, in meters. To find the altitude above mean sea level, add the *mean_sea_level_height* member to the *altitude* member.

The *epe* member provides the estimated position error, 2 sigma, in meters. The *eph* member provides the epe but only for horizontal meters. The *epv* member provides the epe but only for vertical meters.

The *time_of_week* member provides the number of seconds (excluding leap seconds) since the beginning of the current week, which begins on Sunday at 12:00 AM (i.e., midnight Saturday night-Sunday morning). The *time_of_week* member is based on Universal Coordinated Time (UTC), except UTC is periodically corrected for leap seconds while *time_of_week* is not corrected for leap seconds.

To find UTC, subtract the *leap_seconds* member from *time_of_week*. Since this may cause a negative result for the first few seconds of the week (i.e., when *time_of_week* is less than *leap_seconds*), care must be taken to properly translate this negative result to a positive time value in the previous day. In addition, since *time_of_week* is a floating point number and may contain fractional seconds, care must be taken to round off properly when using *time_of_week* in integer conversions and calculations.

The *position* member provides the current position of the Client.

The *east_velocity*, *north_velocity*, and *up_velocity* are used to calculate the current speed of the Client as shown with the equations below.

2 dimension speed = $\sqrt{}$ (*east_velocity*^{2} + *north_velocity*^{2})

The *week_number_days* member provides the number of days that have occurred from UTC December 31st, 1989 to the beginning of the current week (thus, *week_number_days* always represents a Sunday). To find the total number of days that have occurred from UTC December 31st, 1989 to the current day, add *week_number_days* to the number of days that have occurred in the current week (as calculated from the *time_of_week* member).

The enumerated values for the *type_of_gps_fix* member are shown below. It is important for the Server to inspect this value to ensure that other data members are valid.

```
type_of_gps_fix enum
    {
    unusable     = 0,   /* failed integrity check */
    invalid      = 1,   /* invalid or unavailable */
    2D           = 2,   /* two dimensional */
    3D           = 3,   /* three dimensional */
    2D_diff      = 4,   /* two dimensional differential */
    3D_diff      = 5    /* three dimensional differential */
    };
```

6 Appendices

6.1 Packet IDs

A packet ID is an 8-bit unsigned integer type used to identify the type of packet transmitted from the Client to the Server or visa-versa.

The packet IDs that are relevant to this document are listed below. This is not a complete list of packet IDs. The Server should ignore any unrecognized packet ID that it receives from the Client. The values of ASCII DLE (16 decimal) and ASCII ETX (3 decimal) are reserved and will never be used as packet IDs. This allows the software implementation to detect packet boundaries more efficiently.

Packet ID type	Value (decimal)	Description
ACK	6	See Section 3.1.3
Command	10	See Section 5.2.1
Date/Time Data	14	See Section 5.2.3
NAK	21	See Section 3.1.3
Unit ID/ESN	38	See Section 5.2.2
PVT Data	51	See Section 5.2.4
StreetPilot Stop message	135	See Section 6.6.4.1
StreetPilot text message	136	See Section 6.6.1.5
Fleet Management packet	161	See Section 5.1

6.2 Fleet Management Packet IDs

A fleet management packet ID is a 16-bit unsigned integer type used to identify the type of fleet management related data transmitted from one device to another.

The fleet management packet IDs are listed below. The Server should ignore any unrecognized fleet management packet ID that it receives from the Client.

Note: The Description Section number contains hyperlinks to the noted Section.

Fleet Management Packet Type	Value in (hex)	Description	Server to Client	Client to Server
Enable Fleet Management Protocol Request	0x0000	Section 5.1.2	✓	
Product ID and Support Request	0x0001	Section 5.1.3	✓	
Product ID Data	0x0002	Section 5.1.3		✓
Protocol Support Data	0x0003	Section 5.1.3		✓
Unicode Support Request	0x0004	Section 5.1.4		✓
Unicode Support Response	0x0005	Section 5.1.4	✓	
IFTA File Request	0x0006	Section 6.5.9	✓	
IFTA File Receipt	0x0007	Section 6.5.9		✓
IFTA Delete File Request	0x0008	Section 6.5.9.2	✓	
IFTA Delete File Receipt	0x0009	Section 6.5.9.2		✓
Baud Rate Request	0x0011	Section 5.1.23	✓	
Baud Rate Receipt	0x0012	Section 5.1.23		✓
Text Message Acknowledgement	0x0020	Section 5.1.5.1		✓
Text Message (A602 Open Server to Client)	0x0021	Section 6.6.1.1	✓	
Text Message (Simple Acknowledgement)	0x0022	Section 6.6.1.3	✓	
Text Message (Yes/No Confirmation)	0x0023	Section 6.6.1.4	✓	
Text Message (Open Client to Server)	0x0024	Section 5.1.5.3		✓
Text Message Receipt (Open Client to Server)	0x0025	Section 5.1.5.3	✓	
A607 Client to Server Text Message	0x0026	Section 5.1.5.5		✓
Set Canned Response List	0x0028	Section 5.1.5.1.3	✓	
Canned Response List Receipt	0x0029	Section 5.1.5.1.3		✓
Text Message (A604 Open Server to Client)	0x002a	Section 5.1.5.1.1	✓	
Text Message Receipt (A604 Open Server to Client)	0x002b	Section 5.1.5.1.1		✓
Text Message Ack Receipt	0x002c	Section 5.1.5.1.3	✓	
Text Message Delete	0x002d	Section 5.1.5.3	✓	
Text Message Delete Response	0x002e	Section 5.1.5.3		✓
Set Canned Response	0x0030	Section 5.1.5.4.1	✓	
Delete Canned Response	0x0031	Section 5.1.5.4.2	✓	
Set Canned Response Receipt	0x0032	Section 5.1.5.4.1		✓
Delete Canned Response Receipt	0x0033	Section 5.1.5.4.2		✓
Request Canned Response List Refresh	0x0034	Section 5.1.5.4.3		✓
Text Message Status Request	0x0040	Section 5.1.5.2	✓	
Text Message Status	0x0041	Section 5.1.5.2		✓
Set Canned Message	0x0050	Section 5.1.5.6.1	✓	
Set Canned Message Receipt	0x0051	Section 5.1.5.6.1		✓
Delete Canned Message	0x0052	Section 5.1.5.6.2	✓	
Delete Canned Message Receipt	0x0053	Section 5.1.5.6.2		✓
Refresh Canned Message List	0x0054	Section 5.1.5.6.3		✓
Long Text Message (A611 Server to Client)	0x0055	Section 5.1.5.1.2	✓	
Long Text Message Receipt (A611 Server to Client)	0x0056	Section 5.1.5.1.2		✓

Fleet Management Packet Type	Value in (hex)	Description	Server to Client	Client to Server
A602 Stop	0x0100	Section 6.6.4.1	✓	
A603 Stop	0x0101	Section 5.1.6.1	✓	
Sort Stop List	0x0110	Section 5.1.10	✓	
Sort Stop List Acknowledgement	0x0111	Section 5.1.10		✓
Create Waypoint	0x0130	Section 5.1.11.1	✓	
Create Waypoint Receipt	0x0131	Section 5.1.11.1		✓
Delete Waypoint	0x0132	Section 5.1.11.3	✓	
Waypoint Deleted	0x0133	Section 5.1.11.2	✓	✓
Waypoint Deleted Receipt	0x0134	Section 5.1.11.2	✓	
Delete Waypoint by Category	0x0135	Section 5.1.11.4	✓	
Delete Waypoint by Category Receipt	0x0136	Section 5.1.11.4		✓
Create Waypoint Category	0x0137	Section 5.1.11.5	✓	
Create Waypoint Category Receipt	0x0138	Section 5.1.11.5		✓
ETA Data Request	0x0200	Section 5.1.8	✓	
ETA Data	0x0201	Section 5.1.8		✓
ETA Data Receipt	0x0202	Section 5.1.8	✓	
Stop Status Request	0x0210	Section 5.1.7	✓	
Stop Status	0x0211	Section 5.1.7		✓
Stop Status Receipt	0x0212	Section 5.1.7	✓	
Auto-Arrival	0x0220	Section 5.1.9	✓	
Data Deletion	0x0230	Section 5.1.14	✓	
User Interface Text	0x0240	Section 5.1.15	✓	
User Interface Text Receipt	0x0241	Section 5.1.15		✓
Message Throttling Command	0x0250	Section 5.1.17.1	✓	
Message Throttling Response	0x0251	Section 5.1.17.1		✓
Message Throttling Query	0x0252	Section 5.1.17.2	✓	
Message Throttling Query Response	0x0253	Section 5.1.17.2		✓
Ping (Communication Link Status)	0x0260	Section 5.1.16	✓	✓
Ping (Communication Link Status) Response	0x0261	Section 5.1.16	✓	✓

Fleet Management Packet Type	Value in (hex)	Description	Server to Client	Client to Server
GPI File Transfer Start	0x0400	Section 5.1.13.1.1	✓	
GPI File Data Packet	0x0401	Section 5.1.13.1.3	✓	
GPI File Transfer End	0x0402	Section 5.1.13.1.5	✓	
GPI File Start Receipt	0x0403	Section 5.1.13.1.2		✓
GPI Packet Receipt	0x0404	Section 5.1.13.1.4		✓
GPI File End Receipt	0x0405	Section 5.1.13.1.6		✓
GPI File Information Request	0x0406	Section 5.1.13.3.1	✓	
GPI File Information	0x0407	Section 5.1.13.3.1		✓
Path Specific Stop File Server Transfer Start	0x0400	Section 5.1.13.1.1	✓	
Path Specific Stop File Server Data Packet	0x0401	Section 5.1.13.1.3	✓	
Path Specific Stop File Server Transfer End	0x0402	Section 5.1.13.1.5	✓	
Path Specific Stop File Client Start Receipt	0x0403	Section 5.1.13.1.2		✓
Path Specific Stop Packet Client Receipt	0x0404	Section 5.1.13.1.4		✓
Path Specific Stop File Client End Receipt	0x0405	Section 5.1.13.1.6		✓
Custom Form File Server Transfer Start	0x0400	Section 5.1.13.1.1	✓	
Custom Form File Server Data Packet	0x0401	Section 5.1.13.1.3	✓	

Fleet Management Packet Type	Value in (hex)	Description	Server to Client	Client to Server
Custom Form File Server Transfer End	0x0402	Section 5.1.13.1.5	✓	
Custom Form File Client Start Receipt	0x0403	Section 5.1.13.1.2		✓
Custom Form Packet Client Receipt	0x0404	Section 5.1.13.1.4		✓
Custom Form File Client End Receipt	0x0405	Section 5.1.13.1.6		✓
Custom Form File Client Transfer Start	0x0400	Section 5.1.13.2.1		✓
Custom Form File Client Data Packet	0x0401	Section 5.1.13.2.3		✓
Custom Form File Client Transfer End	0x0402	Section 5.1.13.2.5		✓
Custom Form File Server Start Receipt	0x0403	Section 5.1.13.2.2	✓	
Custom Form Packet Server Receipt	0x0404	Section 5.1.13.2.4	✓	
Custom Form File Server End Receipt	0x0405	Section 5.1.13.2.6	✓	
A618 Stop File Server Transfer Start	0x0400	Section 5.1.13.1.1	✓	
A618 Stop File Server Data Packet	0x0401	Section 5.1.13.1.3	✓	
A618 Stop File Server	0x0402	Section 5.1.13.1.5	✓	
A618 Stop File Client	0x0403	Section 5.1.13.1.2		✓
A618 Stop File Client	0x0404	Section 5.1.13.1.4		✓
A618 Stop File Client	0x0405	Section 5.1.13.1.6		✓
Set Driver Status List Item	0x0800	Section 5.1.12.3.1	✓	
Delete Driver Status List Item	0x0801	Section 5.1.12.3.2	✓	
Set Driver Status List Item Receipt	0x0802	Section 5.1.12.3.1		✓
Delete Driver Status List Item Receipt	0x0803	Section 5.1.12.3.2		✓
Driver Status List Refresh	0x0804	Section 5.1.12.3.3		✓
Request Driver ID	0x0810	Section 5.1.12.1	✓	
Driver ID Update	0x0811	Section 5.1.12.1	✓	✓
Driver ID Receipt	0x0812	Section 5.1.12.1	✓	✓
A607 Driver ID Update	0x0813	Section 5.1.12.1.1	✓	✓
Request Driver Status	0x0820	Section 5.1.12.4	✓	
Driver Status Update	0x0821	Section 5.1.12.4	✓	✓
Driver Status Receipt	0x0822	Section 5.1.12.4	✓	✓
A607 Driver Status Update	0x0823	Section 5.1.12.4.1	✓	✓
FMI Safe Mode	0x0900	Section 5.1.18	✓	
FMI Safe Mode Receipt	0x0901	Section 5.1.18		✓
Speed Limit Alert Setup	0x1000	Section 5.1.19	✓	
Speed Limit Alert Setup Receipt	0x1001	Section 5.1.19		✓
Speed Limit Alert	0x1002	Section 5.1.19		✓
Speed Limit Alert Receipt	0x1003	Section 5.1.19	✓	
Driver Login Request	0x1101	Section 6.5.1		✓
Driver Login Response	0x1102	Section 6.5.1	✓	
(generic) Driver Profile Request	0x1103	Section 6.5.1.2.1		✓
HOS_1.0 Driver Profile Response	0x1104	Section 6.5.1.2.1	✓	
HOS_1.0 Driver Profile Update	0x1105	Section 6.5.3.1	✓	
HOS_1.0 Event Log Request	0x1106	Section 6.5.1.3.1		✓
HOS_1.0 Event Log Response	0x1107	Section 6.5.1.3.2	✓	
HOS_1.0 Event Log Receipt	0x1108	Section 6.5.1.3.3		✓
HOS_1.0 Shipment Request	0x1109	Section 6.5.1.4		✓
HOS_1.0 Shipment Response	0x110a	Section 6.5.1.4	✓	
HOS_1.0 Shipment Receipt	0x110b	Section 6.5.1.4		✓
HOS_1.0 Annotation Request	0x110d	Section 6.5.1.5		✓
HOS_1.0 Annotation Response	0x110e	Section 6.5.1.5	✓	
HOS_1.0 Annotation Receipt	0x110f	Section 6.5.1.5		✓
(generic) Driver Profile Receipt	0x110c	Section 6.5.3.1		✓

Fleet Management Packet Type	Value in (hex)	Description	Server to Client	Client to Server
HOS_2.0 Driver Profile Update	0x1110	Section 6.5.3.2	✓	
HOS_2.0 Driver Profile Response	0x1111	Section 6.5.1.2.2	✓	
Custom Form Delete Request	0x1200	Section 5.1.21.4	✓	
Custom Form Delete Receipt	0x1201	Section 5.1.21.4		✓
Custom Form Move Position Request	0x1202	Section 5.1.21.5	✓	
Custom Form Move Position Receipt	0x1203	Section 5.1.21.5		✓
Custom Form Position Request	0x1204	Section 5.1.21.6	✓	
Custom Form Position Receipt	0x1205	Section 5.1.21.6		✓
Path Specific Stop Status Info	0x1220	Section 5.1.6.2.2.1		✓
Path Specific Stop Status Info Receipt	0x1221	Section 5.1.6.2.2.1	✓	
Custom Avoidance New/Modify Request	0x1230	Section 5.1.22.2	✓	
Custom Avoidance New/Modify Receipt	0x1231	Section 5.1.22.2		✓
Custom Avoidance Delete Request	0x1232	Section 5.1.22.3	✓	
Custom Avoidance Delete Receipt	0x1233	Section 5.1.22.3		✓
Custom Avoidance Enable/Disable Request	0x1234	Section 5.1.22.4	✓	
Custom Avoidance Enable/Disable Receipt	0x1235	Section 5.1.22.4		✓
Custom Avoidance Area Enable Request	0x1236	Section 5.1.22.1	✓	
Custom Avoidance Area Enable Receipt	0x1237	Section 5.1.22.1		✓
Driver Auto Status Update	0x1300	Section 6.5.6	✓	
Driver Auto Status Update Receipt	0x1301	Section 6.5.6		✓
Server Initiated Logoff Client Driver	0x1310	Section 6.5.2.2	✓	
Server Initiated Logoff Client Driver Receipt	0x1311	Section 6.5.2.2		✓
Driver 8-Hour Rule Enable	0x1312	Section 6.5.8	✓	
Driver 8-Hour Rule Enable Receipt	0x1313	Section 6.5.8		✓
Alert Pop-up	0x1400	Section 5.1.24	✓	
Alert Pop-up Receipt	0x1401	Section 5.1.24		✓
Configure Sensor Display	0x1402	Section 5.1.25.1	✓	
Configure Sensor Display Receipt	0x1403	Section 5.1.25.1		✓
Update Sensor Display Status	0x1406	Section 5.1.25.2	✓	
Update Sensor Display Status Receipt	0x1403	Section 5.1.25.2		✓
Delete Sensor Display	0x1404	Section 5.1.25.3	✓	
Delete Sensor Display Receipt	0x1405	Section 5.1.25.3		✓
Sensor Display List Position	0x1407	Section 5.1.25.4	✓	
Sensor Display List Position Receipt	0x1408	Section 5.1.25.4		✓
A619 Auto-Status Driver Update	0x1500	Section 6.5.10.1	✓	
A619 Auto-Status Driver Update Receipt	0x1501	Section 6.5.10.1		✓
A619 Driver 8-Hour Rule Enable	0x1500	Section 6.5.10.2	✓	
A619 Driver 8-Hour Rule Enable Receipt	0x1501	Section 6.5.10.2		✓
A619 Periodic Driver Status	0x1500	Section 6.5.10.3	✓	
A619 Periodic Driver Status Receipt	0x1501	Section 6.5.10.3		✓

6.3 Command IDs

The command IDs listed below are decimal. This is not a complete list of command IDs. Only the command IDs that are relevant within this document are listed. The Server should ignore unrecognized command IDs.

Command Description	ID
Request Date/Time Data	5
Request Unit ID/ESN	14

Turn on PVT Data	49
Turn off PVT Data	50

6.4 CRC-32 Algorithm

6.4.1 CRC method

FMI uses the CRC algorithm named "CRC-32 Reversed Representation (0xEDB88320)". The code below generates the CRC table shown in Section 6.4.2

```
unsigned long c;
int n, k;
for (n = 0; n < 256; n++
{
    c = (unsigned long)n;
    for (k = 0; k < 8; k++)
    {
        if (c & 1)
        {
            c = 0xedb88320L ^ (c >> 1);
        }
        else
        {
            c = c >> 1;
        }
    }
    crc_table[n] = c;
}
```

6.4.2 CRC algorithm example

The following is the "CRC-32 Reversed Representation (0xEDB88320)" implementation used by the Fleet Management Interface. To compute a CRC, call *UTL_calc_crc32*, passing a pointer to the file data, the size of the file, and an initial value of zero. For example:

```
uint32 fmi_crc = UTL_calc_crc32(file_data, file_size, 0);
```

Alternately, the CRC value can be computed one block of data at a time by calling the *UTL_accumulate_crc32* function for each block of file data sequentially, then calling *UTL_complete_crc32* at the end to complete the computation. This approach can be used when it is not feasible to keep the entire file in memory.

```
uint32 fmi_crc = 0;
foreach (data_block in file_data)
    {
     fmi_crc = UTL_accumulate_crc32(data_block, sizeof(data_block), fmi_crc);
    }
fmi_crc = UTL_complete_crc32(fmi_crc);
```

```
/*****************************************************************************
*
*   MODULE NAME:
*       UTL_crc.c - CRC Routines
*
*   DESCRIPTION:
*
*   PUBLIC PROCEDURES:
*       Name                        Title
*       ------------------------    ---------------------------------------
*       UTL_calc_crc32              Calculate 32-bit CRC
*
*   PRIVATE PROCEDURES:
*       Name                        Title
*       ------------------------    ---------------------------------------
*       UTL_accumulate_crc32        Accumulate 32-bit CRC Calculation
*       UTL_complete_crc32          Complete 32-bit CRC Calculation
*
*   LOCAL PROCEDURES:
*       Name                        Title
*       ------------------------    ---------------------------------------
*
*   NOTES:
*
*   Copyright 1990-2008 by Garmin Ltd. or its subsidiaries.
*****************************************************************************/

/*---------------------------------------------------------------------
                            MEMORY CONSTANTS
---------------------------------------------------------------------*/
static uint32       const my_crc32_tbl[ 256 ] =
    {
    0x00000000, 0x77073096, 0xEE0E612C, 0x990951BA, 0x076DC419, 0x706AF48F, 0xE963A535,
0x9E6495A3,
    0x0EDB8832, 0x79DCB8A4, 0xE0D5E91E, 0x97D2D988, 0x09B64C2B, 0x7EB17CBD, 0xE7B82D07,
0x90BF1D91,
    0x1DB71064, 0x6AB020F2, 0xF3B97148, 0x84BE41DE, 0x1ADAD47D, 0x6DDDE4EB, 0xF4D4B551,
0x83D385C7,
    0x136C9856, 0x646BA8C0, 0xFD62F97A, 0x8A65C9EC, 0x14015C4F, 0x63066CD9, 0xFA0F3D63,
0x8D080DF5,
    0x3B6E20C8, 0x4C69105E, 0xD56041E4, 0xA2677172, 0x3C03E4D1, 0x4B04D447, 0xD20D85FD,
0xA50AB56B,
    0x35B5A8FA, 0x42B2986C, 0xDBBBC9D6, 0xACBCF940, 0x32D86CE3, 0x45DF5C75, 0xDCD60DCF,
0xABD13D59,
    0x26D930AC, 0x51DE003A, 0xC8D75180, 0xBFD06116, 0x21B4F4B5, 0x56B3C423, 0xCFBA9599,
0xB8BDA50F,
    0x2802B89E, 0x5F058808, 0xC60CD9B2, 0xB10BE924, 0x2F6F7C87, 0x58684C11, 0xC1611DAB,
0xB6662D3D,
    0x76DC4190, 0x01DB7106, 0x98D220BC, 0xEFD5102A, 0x71B18589, 0x06B6B51F, 0x9FBFE4A5,
0xE8B8D433,
    0x7807C9A2, 0x0F00F934, 0x9609A88E, 0xE10E9818, 0x7F6A0DBB, 0x086D3D2D, 0x91646C97,
0xE6635C01,
    0x6B6B51F4, 0x1C6C6162, 0x856530D8, 0xF262004E, 0x6C0695ED, 0x1B01A57B, 0x8208F4C1,
0xF50FC457,
    0x65B0D9C6, 0x12B7E950, 0x8BBEB8EA, 0xFCB9887C, 0x62DD1DDF, 0x15DA2D49, 0x8CD37CF3,
0xFBD44C65,
    0x4DB26158, 0x3AB551CE, 0xA3BC0074, 0xD4BB30E2, 0x4ADFA541, 0x3DD895D7, 0xA4D1C46D,
0xD3D6F4FB,
    0x4369E96A, 0x346ED9FC, 0xAD678846, 0xDA60B8D0, 0x44042D73, 0x33031DE5, 0xAA0A4C5F,
0xDD0D7CC9,
    0x5005713C, 0x270241AA, 0xBE0B1010, 0xC90C2086, 0x5768B525, 0x206F85B3, 0xB966D409,
0xCE61E49F,
    0x5EDEF90E, 0x29D9C998, 0xB0D09822, 0xC7D7A8B4, 0x59B33D17, 0x2EB40D81, 0xB7BD5C3B,
0xC0BA6CAD,
    0xEDB88320, 0x9ABFB3B6, 0x03B6E20C, 0x74B1D29A, 0xEAD54739, 0x9DD277AF, 0x04DB2615,
0x73DC1683,
    0xE3630B12, 0x94643B84, 0x0D6D6A3E, 0x7A6A5AA8, 0xE40ECF0B, 0x9309FF9D, 0x0A00AE27,
0x7D079EB1,
    0xF00F9344, 0x8708A3D2, 0x1E01F268, 0x6906C2FE, 0xF762575D, 0x806567CB, 0x196C3671,
0x6E6B06E7,
```

```
    0xFED41B76, 0x89D32BE0, 0x10DA7A5A, 0x67DD4ACC, 0xF9B9DF6F, 0x8EBEEFF9, 0x17B7BE43,
0x60B08ED5,
    0xD6D6A3E8, 0xA1D1937E, 0x38D8C2C4, 0x4FDFF252, 0xD1BB67F1, 0xA6BC5767, 0x3FB506DD,
0x48B2364B,
    0xD80D2BDA, 0xAF0A1B4C, 0x36034AF6, 0x41047A60, 0xDF60EFC3, 0xA867DF55, 0x316E8EEF,
0x4669BE79,
    0xCB61B38C, 0xBC66831A, 0x256FD2A0, 0x5268E236, 0xCC0C7795, 0xBB0B4703, 0x220216B9,
0x5505262F,
    0xC5BA3BBE, 0xB2BD0B28, 0x2BB45A92, 0x5CB36A04, 0xC2D7FFA7, 0xB5D0CF31, 0x2CD99E8B,
0x5BDEAE1D,
    0x9B64C2B0, 0xEC63F226, 0x756AA39C, 0x026D930A, 0x9C0906A9, 0xEB0E363F, 0x72076785,
0x05005713,
    0x95BF4A82, 0xE2B87A14, 0x7BB12BAE, 0x0CB61B38, 0x92D28E9B, 0xE5D5BE0D, 0x7CDCEFB7,
0x0BDBDF21,
    0x86D3D2D4, 0xF1D4E242, 0x68DDB3F8, 0x1FDA836E, 0x81BE16CD, 0xF6B9265B, 0x6FB077E1,
0x18B74777,
    0x88085AE6, 0xFF0F6A70, 0x66063BCA, 0x11010B5C, 0x8F659EFF, 0xF862AE69, 0x616BFFD3,
0x166CCF45,
    0xA00AE278, 0xD70DD2EE, 0x4E048354, 0x3903B3C2, 0xA7672661, 0xD06016F7, 0x4969474D,
0x3E6E77DB,
    0xAED16A4A, 0xD9D65ADC, 0x40DF0B66, 0x37D83BF0, 0xA9BCAE53, 0xDEBB9EC5, 0x47B2CF7F,
0x30B5FFE9,
    0xBDBDF21C, 0xCABAC28A, 0x53B39330, 0x24B4A3A6, 0xBAD03605, 0xCDD70693, 0x54DE5729,
0x23D967BF,
    0xB3667A2E, 0xC4614AB8, 0x5D681B02, 0x2A6F2B94, 0xB40BBE37, 0xC30C8EA1, 0x5A05DF1B,
0x2D02EF8D,
    };
```

```
/*---------------------------------------------------------------------
                              PROCEDURES
---------------------------------------------------------------------*/

/**********************************************************************
*
*   PROCEDURE NAME:
*       UTL_calc_crc32 - Calculate 32-bit CRC
*
*   DESCRIPTION:
*
*
**********************************************************************/

uint32 UTL_calc_crc32
    (
    uint8   const * const data,
    uint32          const size,
    uint32          const initial_value = 0
    )
{
/*-------------------------------------------------------------
Local Variables
-------------------------------------------------------------*/
uint32      actual_crc;

actual_crc = UTL_accumulate_crc32( data, size, initial_value );
actual_crc = UTL_complete_crc32( actual_crc );

return( actual_crc );
}   /* UTL_calc_crc32() */


/**********************************************************************
*
*   PROCEDURE NAME:
*       UTL_accumulate_crc32 - Accumulate 32-bit CRC Calculation
*
*   DESCRIPTION:
*
**********************************************************************/

uint32 UTL_accumulate_crc32
    (
    uint8   const * const data,
    uint32          const size,
    uint32          const accumulative_value
    )
{
/*-------------------------------------------------------------
Local Variables
-------------------------------------------------------------*/
uint32      actual_crc;
uint32      i;

actual_crc = accumulative_value;
for( i = 0; i < size; i++ )
    {
    actual_crc = my_crc32_tbl[ (data[ i ] ^ actual_crc) & 255] ^ (0xFFFFFF & (actual_crc >> 8));
    }

return( actual_crc );
}   /* UTL_accumulate_crc32 */
```

```
/*****************************************************************
*
*   PROCEDURE NAME:
*       UTL_complete_crc32 - Complete 32-bit CRC Calculation
*
*   DESCRIPTION:
*
*
*****************************************************************/

uint32 UTL_complete_crc32
    (
    uint32          const actual_crc
    )
{
return( ~actual_crc );
}   /* UTL_complete_crc32() */
```

6.5 Hours of Service (HOS) Functionality

The Hours of Service FMI functionality is provided on Dezl navigation devices. Current functionality is intended to work with a Server to qualify as an Automatic On-Board Recording Device (AOBRD) that would meet the FMCSA's §395.15 standard.

To utilize the AOBRD functionality, a driver must be "logged-in" to the Garmin device with the A610 or A615 protocol enabled. The full login procedure requires 5 sequential steps, which are listed below (in the order of execution), but note that a driver will be shown as "Logged-in" upon completion of the Driver Profile step.

1. **Driver Authentication** (see Section 6.5.1)
   - Client sends Driver ID and Password to Server for authentication (packet_id = 0x1101)
   - Server responds indicating pass/fail result (packet_id = 0x1102)

2. **Download of Driver's Profile** (see Section 6.5.1.2)
   - Client requests a driver's profile (packet_id = 0x1103)
   - Server responds with driver profile data (packet_id = 0x1104 (Property only), or 0x1111)

   Note: Following a successful driver profile download, the driver will be shown as "Logged-in" on the Client. Any errors occurring in the remaining login steps (steps 3 to 5) will not affect this driver's "Logged-in" state (e.g., errors during download of Status Change logs, errors during download of Shipments, or errors during download of Annotations).

3. **Download of Driver's Status Change logs** (see Section 6.5.1.3)
   - Client requests availability for a driver's Status Change log file (packet_id = 0x1106)
   - Server responds with yes/no that a log file is available (packet_id = 0x1107)
   - Client requests Server for the driver's Status Change log file (packet_id = 0x1108)
   - Server responds by sending a log file using the File Transfer protocol (see Section for packet ids)

4. **Download of Driver's Shipments** (see Section 6.5.1.4)
   - Client requests a single driver's shipment record (packet_id = 0x1109)
   - Server responds with a shipment record (packet_id = 0x110a)
   - Client responds with a ACK (packet_id = 0x110b)
   - Server responds with next shipment record or a "done" indicator (packet_id = 0x110a)
   - Client responds with a ACK (packet_id = 0x110b)

5. **Download of Driver's Annotations** (see Section 6.5.1.5)
   - Client requests a single driver's annotation record (packet_id = 0x110d)
   - Server responds with a shipment record (packet_id = 0x110e)
   - Client responds with a ACK (packet_id = 0x110f)
   - Server responds with next shipment record or a "done" indicator (packet_id = 0x110e)
   - Client responds with a ACK (packet_id = 0x110f)

6.5.1 AOBRD Driver Login

6.5.1.1 AOBRD Driver Authentication Protocol

The login protocol is used to allow a driver to log into the Garmin device, and take advantage of the available electronic logging capabilities. The driver must be identified within the system to allow the appropriate data to be loaded onto the device, so events recorded by the device can be attributed to the correct person. The packet sequence is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Client to Server	0x1101 –Driver Login Request	driver_login_request_data_type
1	Server to Client	0x1102 – Driver Login Response	driver_login_response_data_type

Client to Server type definition for the *driver_login_request_data_type* is shown below. This data type is only supported on Clients that report D610 as part of their protocol support data.

```
typedef struct  /*  D610  */
    {
    time_type ui_timestamp;
    uchar_t8  driver_password[];/* variable length, null terminated string, 20 bytes max */
    uchar_t8  driver_id[]; /* variable length, null terminated string, 50 bytes max */
    } driver_login_request_data_type;
```

The *ui_timestamp* is the time that the login request was sent from the Client. The *driver_password* is the password for the driver logging into the system, and *driver_id* is the login ID of the driver.

Server to Client type definition for the *driver_login_response_data_type* is shown below. This data type is only supported on Clients that report D610 as part of their protocol support data.

```
typedef struct  /*  D610  */
    {
    time_type   ui_timestamp;
    uint8       result_code;
    } driver_login_response_data_type;
```

The *ui_timestamp* will be identical to the value received in the Driver Login Request packet. The *result_code* indicates whether or not the login was successful. A *result_code* of zero indicates success, a nonzero *result_code* means that an error occurred, according to the table below.

Note: The protocol should not continue if the result_code is nonzero.

Result Code (Decimal)	Meaning
0	Success
1	Error
2	Unsupported mode

6.5.1.2 AOBRD Driver Profile Protocols

The driver profile protocols are used to provide the Client with the most up-to-date driver profile information. After the Client has successfully completed a login exchange, it will request the profile data for that driver. Later during device use, the Server can also push driver profile updates to the Client.

6.5.1.2.1 Driver Profile Request to Server (Property Carrying only)

This protocol allows the Client to request the profile information for the specified driver. The packet sequence is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Client to Server	0x1103 – (generic) Driver Profile Request	driver_profile_request_data_type
1	Server to Client	0x1104 – HOS_1 Driver Profile Response	driver_profile_response_data_type

Client to Server type definition for the *driver_profile_request_data_type* is shown below. This data type is only supported on Clients that report D610 as part of their protocol support data.

```
typedef struct  /*  D610  */
    {
    uchar_t    driver_id[];          /* variable length, null terminated string, 50 bytes max */
    } driver_profile_request_data_type;
```

The driver_id is the login ID of the driver.

The **Server to Client** type definition for the *driver_profile_response_data_type* is shown below. This data type is only supported on Clients that report D610 as part of their protocol support data.

```
typedef struct  /*  D610  */
    {
    uint8       reserved[4]; /* Set to 0 */
    uchar_t    first_name[]; /* variable length, null terminated string, 35 bytes max */
    uchar_t    last_name[]; /* variable length, null terminated string, 35 bytes max */
    uchar_t    driver_id[]; /* variable length, null terminated string, 40 bytes max */
    uchar_t    carrier_name[]; /* variable length, null terminated string, 120 bytes max */
    uchar_t    carrier_id[]; /* variable length, null terminated string, 8 bytes max */
    uint8       rule_set;
    uint8       time_zone;
    uint8       reserved; /* Set to 0 */
    uint8       result_code;
    } driver_profile_response_data_type;
```

The *first_name* and *last_name* will be the driver's first and last names, respectively. The *driver_id* will be identical to the value received in the Driver Profile Request packet. The *carrier_name* is the name or trade name of the motor carrier company and possibly the address of that company. The *carrier_id* is the USDOT number of the motor carrier.

The *rule_set* specifies the hours-of-services rules used by the driver. The table below defines the different values for *rule_set*.

Rule Set (Decimal)	Meaning
0	Sixty hour/seven day rule set
1	Seventy hour/eight day rule set

The time*_zone* is the time zone to use when recording driver statuses. The table below defines the different values for *time_zone*.

Time Zone (Decimal)	Meaning
0	Eastern Time Zone
1	Central Time Zone
2	Mountain Time Zone
3	Pacific Time Zone
4	Alaska Time Zone
5	Hawaii Time Zone

The *result_code* indicates whether or not driver profile being provided by the Server is valid. A *result_code* of zero indicates success, a nonzero *result_code* means that an error occurred, according to the table below.

Note: The protocol should not continue if the result_code is nonzero.

Result Code (Decimal)	Meaning
0	Success
1	Unknown Driver ID
2	Error

6.5.1.2.2 Driver Profile Request to Server (Property/Passenger Carrying)

This protocol allows the Client to request the profile information for the specified driver. This event occurs when a driver logs-in on the Client. The request is sent to the Server using the same request packet as used for HOS_1.0 driver login. The Server then has the option to either send the new "HOS_2.0 Driver Profile" format or the existing "HOS_1.0 driver profile format" (which will not provide Passenger-carrying rule sets or Adverse Conditions functionality).

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Client to Server	0x1103 – (generic) Driver Profile Request	driver_profile_request_data_type
1	Server to Client	0x1111 – HOS_2.0 Driver Profile Response	hos2_driver_profile_response_data_type

The **Client to Server** type definition for *driver_profile_request_data_type* is shown below. This data type is only supported on Clients that report D610 or D615 as part of their protocol support data.

```
typedef struct  /*  D610 or D615  */
    {
    uchar_t8    driver_id[];            /* null terminated string, 50 bytes */
    } driver_profile_request_data_type;
```

The *driver_id* is the login ID of the driver.

The **Server** has the option to either respond with a "HOS_2.0 Driver Profile" profile structure that contains Passenger/Property Carrying and Adverse Conditions entries, or to respond using the HOS_1.0 driver profile structure. The packet shown below depicts a Server's response using the new "HOS_2.0 Driver Profile" hos2*_driver_profile _response_data_type* with Passenger/Property and Adverse Conditions.

Server to Client type definition for the *HOS_2.0 Driver Profile Response* is shown below. This data type is only supported on Clients that report D615 as part of their protocol support data.

```
typedef struct  /*  D615  */
    {
    char           first_name[];           /* null terminated string, 35 bytes */
    char           last_name[];            /* null terminated string, 35 bytes */
    char           driver_id [];           /* null terminated string, 40 bytes */
    char           carrier_name[];         /* null terminated string, 120 bytes */
    char           carrier_id[];           /* null terminated string, 8 bytes */
    uint8          long_term_rule_set;     /* 0 = sixty-hour/7 day rule-set, */
                                           /* 1 = seventy-hour/8 day rule-set */
    uint8          load_type_rule_set;     /* 0 = property-carrying vehicle, */
```

```
                                        /* 1 = passenger-carrying vehicle */
date_time_t32  adverse_condition_time;  /* Timestamp of Client reported Adverse Conditions */
                                        /* If no Timestamp exists, set to 0 */
uint8          time_zone;               /* See Time Zone table below */
uint8          reserved;                /* Set to 0 */
uint8          result_code;             /* See Result Code table below */
} hos2_driver_profile_response_data_type;
```

The *first_name* and *last_name* will be the driver's first and last names, respectively. These values MUST match the values provided in the original driver profile request. The *driver_id* will be identical to the value received in the *Driver Profile Request* packet.

The *carrier_name* is the name or trade name of the motor carrier company and possibly the address of that company. The *carrier_id* is the USDOT number of the motor carrier.

The *long_term_rule_set* is the set of hours-of-service driving rules for the driver. The *load_type_rule_set* specifies the type of load the driver is carrying.

The *adverse_condition_time* reflects a *Timestamp* from an earlier Client reported *Adverse Conditions Annotation* packet described in Section 6.5.4.1.3.1. If the Server had not received an *Adverse Conditions Annotation* packet from the Client for this driver, the Server should ensure this member value is set to '0'.

The *time_zone* is the time zone to use when recording driver statuses (see below):

Time Zone (Decimal)	Meaning
0	Eastern Time Zone
1	Central Time Zone
2	Mountain Time Zone
3	Pacific Time Zone
4	Alaska Time Zone
5	Hawaii Time Zone

The *result_code* indicates whether or not the driver profile being provided by the Server is valid. A *result_code* of zero indicates success, a nonzero *result_code* means that an error occurred, according to the table below:

Note: The login protocol will not continue if the result_code *is nonzero.*

Result Code (Decimal)	Meaning
0	Success
1	Unknown Driver ID
2	Error

6.5.1.3 Duty Status Change Event Logs File Request by Client

On the Server side, Client driver status change event log records are collected as a list of contiguous records, and sent from the Server to the Client using the File Transfer Protocols described in Section 5.1.13.1 during driver login. This restores a driver's status change history on the Client.

This protocol allows the Client to request a driver's Duty Status Change event logs from the Server. This process is typically done during driver login. A log file will consist of one contiguous bundle of individual Duty Status Change events, which would have previously been reported from the Client during driver status change events.

NOTE: Each record contained in this log file should be an exact copy (format) of the original Duty Status Change Event packet as originally sent from the Client.

Each record within the log will be interrogated by the Client for proper format and contents. Each Duty Status Change record must indicate type '1' in the *Event Type* field as defined in the event header in Section 6.5.4.1.1. Any other event types will cause an error to be returned to the Server.

In addition, the record entries in the log file should be ordered from oldest to newest. Any Client detected error will halt record processing of a log file, and results in no Duty Status changes being saved for that driver. The packet sequence is shown below:

N	Direction	Fleet Management Packet ID	Hyper-Link	Fleet Management Packet Data Type
0	Client to Server	0x1106 – Driver Event Log Request	6.5.1.3.1	driver_event_log_request_data_type
1	Server to Client	0x1107 – Driver Event Log Response	6.5.1.3.2	driver_event_log_response_data_type
2	Client to Server	0x1108 – Driver Event Log Receipt	6.5.1.3.3	driver_event_log_receipt_data_type
3	Server to Client	0x0400 – File Transfer Start Packet ID	5.1.13.1.1	file_info_data_type
4	Client to Server	0x0403 – File Start Receipt Packet ID	5.1.13.1.2	file_receipt_data_type
5..n-3	Server to Client	0x0401 – File Data Packet ID	5.1.13.1.3	file_packet_data_type
6..n-2	Client to Server	0x0404 – Packet Receipt Packet ID	5.1.13.1.4	packet_receipt_data_type
n-1	Server to Client	0x0402 – File Transfer End Packet ID	5.1.13.1.5	file_end_data_type
n	Client to Server	0x0405 – File End Receipt Packet ID	5.1.13.1.6	log_file_receipt_data_type

NOTE: The File Transfer portion is performed by the generic File Transfer in this document, and can be found by clicking the hyperlinks above.

6.5.1.3.1 Client to Server - Packet ID: 0x1106 - Driver Event Log Request

The Client to Server type definition for the *driver_event_log_request_data_type* is shown below. This data type is only supported on Clients that report D610 as part of their protocol support data.

```
typedef struct  /*  D610  */
    {
    uchar_t8    driver_id[];    /* variable length, null terminated string, 40 bytes max */
    } driver_event_log_request_data_type;
```

The *driver_id* is the login ID of the driver.

Back to packet sequence table click here> 6.5.1.3

6.5.1.3.2 Server to Client - Packet ID: 0x1107 - Driver Event Log Response

Server to Client type definition for the *driver_event_log _response_data_type* is shown below. This data type is only supported on Clients that report D610 as part of their protocol support data.

```
typedef struct  /*  D610  */
    {
    uchar_t8    driver_id[]; /* variable length, null terminated string, 40 bytes max */
    uint8       result_code;
    } driver_event_log_response_data_type;
```

The *driver_id* should be identical to the value sent in the Driver Event Log Request packet. The *result_code* indicates whether or not the Server will be able to send a log file to the Client. A *result_code* of zero indicates that

an event log file will follow. A nonzero *result_code* means that no event log file will be sent.

Result Code (Decimal)	Meaning
0	Event log file will follow
1	No event logs exist for this driver
2	Error

Back to packet sequence table, click here> 6.5.1.3

6.5.1.3.3 Client to Server - Packet ID: 0x1108 - Driver Event Log Receipt

The Client to Server type definition for the *driver_event_log _receipt_data_type* is shown below. This data type is only supported on Clients that report D610 as part of their protocol support data.

```
typedef struct  /*  D610  */
    {
    uchar_t8    driver_id[];     /* variable length, null terminated string, 40 bytes max */
    uint8       result_code;
    } driver_event_log_receipt_data_type;
```

The *driver_id* will be identical to the value received in the Driver Event Log Response packet. The *result_code* indicates whether or not the Server can start sending the log file to the Client. A *result_code* of zero indicates that the Client is ready to receive the event log file. A nonzero *result_code* means that the Server should abort sending the event log file.

Result Code (Decimal)	Meaning
0	Send the event log
1	Driver ID sent from Server does not match the requested Driver ID
2	Invalid Server response
3	Unexpected Server response

If the Server receives a result code of "0", it will then begin to transfer the event log file to the Client using the File Transfer Protocol described in Section 5.1.13.1 and repeated below.

Back to packet sequence table, click here> 6.5.1.3

6.5.1.4 AOBRD Shipment Protocol

The shipment protocol allows the Server to provide the Client with a set of shipments that are assigned to the specified driver. The Client will request this set of shipments using the packet sequence below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Client to Server	0x1109 – Shipment Request	shipment_request_data_type
1..n-1	Server to Client	0x110a – Shipment Response	shipment_response_data_type
2..n	Client to Server	0x110b – Shipment Receipt	shipment_receipt_data_type

The Client to Server type definition for the *shipment_request_data_type* is shown below. This data type is only supported on Clients that report D610 as part of their protocol support data.

```
typedef struct  /*  D610  */
    {
    uchar_t8    driver_id[];       /* variable length, null terminated string, 40 bytes max */
    } shipment_request_data_type;
```

The *driver_id* is the login ID of the driver.

The Server to Client type definition for the *shipment_response_data_type* is shown below. This data type is only supported on Clients that report D610 as part of their protocol support data.

```
typedef struct  /*  D610  */
    {
    time_type   shipment_timestamp;
    time_type   start_time;
    time_type   end_time;
    uchar_t8    shipper_name[]; /* variable length, null terminated string, 40 bytes max */
    uchar_t8    document_number[]; /* variable length, null terminated string, 40 bytes max */
    uchar_t8    commodity[]; /* variable length, null terminated string, 40 bytes max */
    uchar_t8    driver_id[]; /* variable length, null terminated string, 40 bytes max */
    uint8       result_code;
    } shipment_response_data_type;
```

The *shipment_timestamp* is the time that the shipment entry was created. The *start_time* and *end_time* are the times that the shipment started being shipped and shipping was completed, respectively. The *shipper_name* is the name of the shipping company. The *document_number* is the document number for the shipment. The *commodity* is the type of commodity in the shipment.

The *driver_id* will be identical to the value received in the Shipment Request packet, and match the *driver_id* in the driver's profile. The *result_code* indicates whether or not the shipment data is valid.

Result Code (Decimal)	**Meaning**
0	Contains Shipment Data
1	No Shipment Data
2	Unknown Driver ID
3	Unsupported Request

The Client to Server type definition for the shipment_receipt_data_type is shown below. This data type is only supported on Clients that report D610 as part of their protocol support data.

```
typedef struct  /*  D610  */
    {
    uchar_t8    driver_id[];       /* variable length, null terminated string, 40 bytes max */
    uint8       result_code;
    } shipment_receipt_data_type;
```

The *driver_id* will be identical to the value received in the Shipment Response packet. The *result_code* indicates how the shipment data was handled by the Client.

Result Code (Decimal)	**Meaning**
0	Success
1	Driver ID mismatch
2	Invalid value for Driver ID
3	Unexpected Packet
4	Shipment data rejected by Client

There can be more than one Shipment Response packet sent after a single Shipment Request is sent if there are multiple shipments to be provided to the Client. After the Server receives a Shipment Receipt for the shipment sent, it will send the next shipment until all have been sent. Once all shipments have been sent, the final Shipment Response will indicate that there is no more Shipment Data by setting the *result_code* to 1. Shipments will only be accepted by the Client following a Client Shipment Request or Client Shipment Receipt.

Note: Once the Server informs the Client no other Shipments are available, the Client will reject all additional Shipment packets for that driver.

6.5.1.5 AOBRD Annotation Protocol

This protocol allows the Client to request a driver's event Annotations (see Section 6.5.4.1.3.1) from the Server during driver login, so the driver's previously created Annotations are restored back into the Client. After driver login, the Annotations can be observed on the Client. The Client will request this set of Annotations using the packet sequence below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Client to Server	0x110d – Annotation Request	annotation_request_data_type
1..n-1	Server to Client	0x110e – Annotation Response	annotation_response_data_type
2..n	Client to Server	0x110f – Annotation Receipt	annotation_receipt_data_type

Client to Server type definition for the *annotation_request_data_type* is shown below. This data type is only supported on Clients that report D610 as part of their protocol support data.

```
typedef struct  /*  D610  */
    {
    uchar_t8    driver_id[];        /* variable length, null terminated string, 40 bytes max */
    } annotation_request_data_type;
```

The *driver_id* is the login ID of the driver.

The Server to Client type definition for the *annotation_response_data_type* is shown below. This data type is only supported on Clients that report D610 as part of their protocol support data.

```
typedef struct  /*  D610  */
    {
    time_type   comment_timestamp;
    time_type   start_time;
    time_type   end_time;
    uchar_t8    comment[]; /* variable length, null terminated string, 60 bytes max */
    uchar_t8    driver_id[]; /* variable length, null terminated string, 40 bytes max */
    uint8       result_code;
    } annotation_response_data_type;
```

The *annotation_timestamp* is the time that the Annotation entry was created. The *start_time* and *end_time* are the times entered by the driver, respectively. The *comment* string is the contents of the Annotation data entered by the driver. The *result_code* indicates whether or not Annotation data is included.

Result Code (Decimal)	Meaning
0	Contains Annotation Data
1	No Annotation Data

Client to Server type definition for the *annotation_receipt_data_type* is shown below. This data type is only supported on Clients that report D610 as part of their protocol support data.

```
typedef struct  /*  D610  */
    {
    uchar_t8    driver_id[];        /* variable length, null terminated string, 40 bytes max */
    uint8       result_code;
    } annotation_receipt_data_type;
```

The *driver_id* will be identical to the value received in the Annotation Response packet. The *result_code* indicates how the Annotation data was handled by the Client.

Result Code (Decimal)	Meaning
0	Success
1	Driver ID mismatch
2	Invalid value for Driver ID
3	Unexpected Packet
4	Annotation data rejected by Client

There can be more than one Annotation Response packet sent after a single Annotation Request is sent if there are Annotations to be provided to the Client. After the Server receives an Annotation Receipt for an Annotation sent, it will send the next Annotation until all have been sent.

Once all Annotations have been sent by the Server, the final Annotation Response will indicate that there is no more Annotation Data by the Server setting the *result_code* to 1 "*No Annotation Data*". Annotations will only be accepted by the Client following a Client Annotation Request or Client Annotation Receipt.

Note: Once the Server informs the Client no other Annotations are available, the Client will reject all additional Annotation packets for that driver.

6.5.2 AOBRD Driver Logout

The AOBRD Driver Logout protocols either indicate or initiate the logout of a specific driver on the Client. Following a driver logout event, the Client deletes a driver's AOBRD data. Driver logout can be achieved by any of the three protocols listed in this section.

6.5.2.1 Driver Initiated Logout Protocol

This protocol notifies the Server a specific driver initiated a logout event on the Client by using the Client's User Interface.

Note: If the current driver's duty status is not "OFF_DUTY" then a Duty Status Change is made by the Client to set the driver to an" off-duty" state, which would generate an AOBRD status change event sent to the Server (see Section 6.5.4.1.3.2).

Following a possible Client invoked status change (as noted above), an AOBRD Annotation event is sent to the Server (see the AOBRD Annotation Event Section 6.5.4.1.3.1), which contains the Annotation text of "**System DETECTED Driver Logout**".

6.5.2.2 Server Initiated Logout Protocol

This protocol allows the Server to logout a specific driver on the Client, and the Client will then display a User Interface message if the logout is successful. The Server Initiated Logout Protocol is shown below:

Note: If the current driver's duty status is not "OFF_DUTY" then a Duty Status Change is made by the Client to set the driver to an" off-duty" state, which would generate an AOBRD status change event sent to the Server (see Section 6.5.4.1.3.2).

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1310 – Logout Driver	logout_driver_data_type
1	Client to Server	0x1311 – Logout Driver Receipt	logout_driver_receipt_data_type

The **Server to Client** type definition for the *logout_driver_data_type* is shown below. This data type is only supported on Clients that report D615 as part of their protocol support data.

```
typedef struct  /*  D615  */
    {
    char  driver_id[];         /* null terminated string, 40 bytes */
    } logout_driver_data_type;
```

The *driver_id* represents the driver the Server is requesting to logout the Client.

Client to Server receipt packet is sent back to the Server to indicate the result of the Server initiated logout request, which is shown below:

```
typedef struct  /*  D615  */
    {
    char   driver_id[];               /* null terminated string, 40 bytes */
    uint8  status;                    /* 0 = driver logged-out, 1 = driver NOT logged-out
    uint8  result_code;               /* See table below */
    } logout_driver_receipt_data_type;
```

The *driver_id* represents the driver the Server is requesting to logout the Client. The *status* indicates the current "logged-off" state of the driver. The *result_code* indicates the end result of the Server request.

Result Code (Decimal)	Meaning
0	Success
1	AOBRD not enabled error
2	No driver ID error
3	Driver not found error
All others	Internal error

Following a possible Client invoked status change (as noted above), an AOBRD Annotation event is sent to the Server (see the AOBRD Annotation Event Section 6.5.4.1.3.1), which contains the Annotation text of "**AOBRD Remote/Server Logoff Driver**".

6.5.2.3 Client System Initiated Logout Protocol

This protocol indicates a specific driver was forced to the logout state due to the Client receiving an *Enable Fleet Management* protocol packet with the AOBRD feature bit set to the disable state (see Section 5.1.2).

Note: If the current driver's duty status is not "OFF_DUTY" then a Duty Status Change is made by the Client to set the driver to an" off-duty" state, which would generate an AOBRD status change event sent to the Server (see Section 6.5.4.1.3.2).

Following a possible Client invoked status change (as noted above), an AOBRD Annotation event is then sent to the Server (see the AOBRD Annotation Event Section 6.5.4.1.3.1), which contains the Annotation text of "**AOBRD disable FORCED Driver Logout**".

6.5.3 AOBRD Driver Profile Update Protocol

6.5.3.1 HOS_1.0 Driver Profile Update to Client (Property Carrying only)

This protocol allows the Server to provide an update to the profile information for a specified driver currently "logged-in" on the Client. The packet sequence is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1105 – HOS_1.0 Driver Profile Update	driver_profile_update_data_type
1	Client to Server	0x110c – (generic) Driver Profile Receipt	driver_profile_update_receipt_data_type

The Server to Client type definition for the *driver_profile_update_data_type* is shown below. It contains the same structure and members as the *driver_profile_response_data_type* used during driver login (see Section 6.5.1.2.1 above). This data type is only supported on Clients that report D610 as part of their protocol support data.

Note: The driver name or driver ID will not be modified by a Driver Profile Update.

```
typedef struct  /*  D610  */
    {
    uint8       reserved[4]; /* Set to 0 */
    uchar_t8    first_name[]; /* variable length, null terminated string, 35 bytes max */
    uchar_t8    last_name[]; /* variable length, null terminated string, 35 bytes max */
    uchar_t8    driver_id[]; /* variable length, null terminated string, 40 bytes max */
    uchar_t8    carrier_name[]; /* variable length, null terminated string, 120 bytes max */
    uchar_t8    carrier_id[]; /* variable length, null terminated string, 8 bytes max */
    uint8       rule_set;
    uint8       time_zone;
    uint8       reserved; /* Set to 0 */
    uint8       result_code;
    } driver_profile_update_data_type;
```

The *first_name* and *last_name* will be the driver's first and last names, respectively. These values MUST match the values provided in the original driver profile request. The *driver_id* will be identical to the value received in the Driver Profile Response packet. The *carrier_name* is the name or trade name of the motor carrier company and possibly the address of that company. The *carrier_id* is the USDOT number of the motor carrier.

The *rule_set* is the set of hours-of-service driving rules used by the driver. The *time_zone* is the time zone to use when recording driver statuses. The *result_code* indicates whether or not driver profile being provided is valid. The appropriate values for *rule_set*, *time_zone*, and *result_code* are defined in Section 6.5.1.2.1.

The Client to Server type definition for the *driver_profile_receipt_data_type* is shown below. This data type is only supported on Clients that report D610 as part of their protocol support data.

```
typedef struct  /*  D610  */
    {
    uint8       result_code;
    uchar_t8    driver_id[];       /* variable length, null terminated string, 40 bytes max */
    } driver_profile_update_receipt_data_type;
```

The *driver_id* will be identical to the value received in the Profile Update request packet. The *result_code* indicates how the profile data was handled by the Client.

Result Code (Decimal)	Meaning
0	Success
1	Failed
2	Driver declined update
3	Busy, already processing a profile update
4	No driver id data in Server request
5	Storage error
6	Internal result error
7	Interface error

6.5.3.2 HOS_2.0 Driver Profile Update to Client (Property/Passenger Carrying)

This protocol allows the Server to initiate a driver profile update for a specified driver that specified driver currently "logged-in" on the Client. The Server has the option to either send this "HOS_2.0 Driver Profile" or the existing HOS_1.0 driver profile format (which will not provide Passenger-carrying rule sets or Adverse Conditions functionality). The packet sequence is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1110 – HOS_2.0 Driver Profile Update	hos2_driver_profile_update_data_type
1	Client to Server	0x110c – (generic) Driver Profile Receipt	driver_profile_update_receipt_data_type

Server to Client type definition for the *hos2_driver_profile_update_data_type* is shown below. This data type is only supported on Clients that report D615 as part of their protocol support data.

```
typedef struct  /*  D615  */
    {
    char           first_name[];            /* null terminated string, 35 bytes */
    char           last_name[];             /* null terminated string, 35 bytes */
    char           driver_id [];            /* null terminated string, 40 bytes */
    char           carrier_name[];          /* null terminated string, 120 bytes */
    char           carrier_id[];            /* null terminated string, 8 bytes */
    uint8          long_term_rule_set;      /* 0 = sixty-hour/7 day rule-set, */
                                            /* 1 = seventy-hour/8 day rule-set */
    uint8          load_type_rule_set;      /* 0 = property-carrying vehicle, */
                                            /* 1 = passenger-carrying vehicle */
    date_time_t32  adverse_condition_time;  /* Timestamp of Client reported Adverse Conditions */
                                            /* If no Timestamp exists, set to 0 */
    uint8          time_zone;               /* See Time Zone table below */
    uint8          reserved;                /* Set to 0 */
    uint8          result_code;             /* See Result Code table below */
    } hos2_driver_profile_update_data_type;
```

The *first_name* and *last_name* will be the driver's first and last names, respectively. These values MUST match the values provided in the original driver profile request. The *driver_id* will be identical to the value received in the *Driver Profile Response* packet.

The *carrier_name* is the name or trade name of the motor carrier company and possibly the address of that company. The *carrier_id* is the USDOT number of the motor carrier.

The *long_term_rule_set* specifies the hours-of-services rules used by the driver. The *long_term_rule_set* is the set of hours-of-service driving rules used by the driver. The *load_type_rule_set* specifies the type of load the driver is carrying.

The *adverse_condition_time* reflects a *Timestamp* from a Client reported *Adverse Conditions Annotation* packet described in Section 6.5.7. If the Server had not received an *Adverse Conditions Annotation* packet from the Client for this driver, the Server should ensure this member value is set to '0'.

The *time_zone* is the time zone to use when recording driver statuses (see below).

Time Zone (Decimal)	Meaning
0	Eastern Time Zone
1	Central Time Zone
2	Mountain Time Zone
3	Pacific Time Zone
4	Alaska Time Zone
5	Hawaii Time Zone

The *result_code* indicates whether or not driver profile being provided is a valid one (see below):

Result Code (Decimal)	Meaning
0	Success
1	Unknown Driver ID
2	Error

Client to Server type definition for the *driver_profile_update_receipt_data_type* is shown below. This data type is only supported on Clients that report D610 or D615 as part of their protocol support data.

```
typedef struct  /*  D610 or D615  */
    {
    uint8       result_code;
    uchar_t8    driver_id[];                    /* null terminated string, 40 bytes */
    } driver_profile_update_receipt_data_type;
```

The *driver_id* will be identical to the value received in the Profile Update request packet. The *result_code* indicates how the profile data was handled by the Client (see below):

Result Code (Decimal)	Meaning
0	Success
1	Failed
2	Driver declined update
3	Busy, already processing a profile update
4	No driver id data in Server request
5	Storage error
6	Internal result error
7	Interface error

6.5.4 AOBRD Event Log Protocol

The event log protocol is used to synchronize changes in the driver's status, or alert the Server of some driver activity or system event. Several event types are sent to the Server, but only previous driver status change events are sent back to the Client in the form of a bundled log.

Client event log data is placed into a single record and transferred to the Server using the File Transfer Protocols described in Section 5.1.13.2.

Note: If the Client is unable to successfully transmit the driver's AOBRD event log data to the Server, the Client will buffer the event log data in protected memory. The Client can buffer up to 500 AOBRD event log packets.

6.5.4.1 Event Log Record Format

The event log record will contain a list of contiguous data sections. Each record will consist of the following three sections: header, driver/truck data, and event specific data.

6.5.4.1.1 Event Header

The header is the set of fields describes what will be contained by the event. It indicates which driver/truck data fields and what event specific data will be contained by this event. The header fields are defined below.

Byte Number	Byte Description	Notes
0-1	Version	Version of the Event Log Protocol Format
2-5	Field Flags	Bitmask specifying which fields in the driver/truck data and event data are present.
6-9	Reserved	Reserved for Future Use
10	Event Type	Identifies the Type of Event. The following are valid event values: 0 = Annotation 1 = Duty Status Change 2 = Log Verified 3 = Log Verification Error 4 = New Shipment Added 5 = Existing Shipment Modified 6 = Device Failure Detected 7 = Shipment Deleted 8 = Previous Status Change Modified by System (Passenger Seat Exception) 9 = New Status Change in Past by System (Passenger Seat Exception)

6.5.4.1.2 Driver/Truck Data

The *driver/truck* data is the set of data that can be included with any event. The *Field Flags* entry of the header indicates which of the possible fields are included in this event. The data types for *driver/truck* data are *time_type*, uint32, and *char_string*. The *char_string* data type is defined below.

```
typedef struct
    {
    uint8       size;
    uchar_t8    str[];      /* string with length defined by size field */
    } char_string;
```

The fields defined in the *driver/truck* data are defined below.

Flag Field Bit	Data Type	Name	Description
0	time_type	Timestamp	The timestamp when this event occurred
1	char_string	Driver First Name	First name of the driver for this event
2	char_string	Driver Last Name	Last name of the driver for this event
3	char_string	Driver ID	Driver Identifier for this event
4	char_string	Co-Driver First Name	First name of the co-driver for this event
5	char_string	Co-Driver Last Name	Last name of the co-driver for this event
6	char_string	Co-Driver ID	ID of the co-driver for this event
7	char_string	Tractor Number	Tractor number for this event
8	char_string	Trailer Number	Trailer number for this event
9	char_string	Tractor VIN	VIN of the tractor for this event
10	uint32	Odometer Reading	Odometer reading when this event occurred
11	char_string	Carrier ID	USDOT number of the motor carrier
12	char_string	Carrier Name	Name or trade name of the motor carrier company
13	char_string	Nearest City Name	Nearest city when this event occurred
14	char_string	Nearest State Name	Nearest state when this event occurred

6.5.4.1.3 Event Specific Data

The event specific data is the set of data provided based on the event type. Each event type has its own event specific data. Like the *driver/truck* data, the *Fields Flag* entry of the header defines which of the event specific fields is present in the event. The data types for event data are *time_type*, uint16, uint32, and the *char_string* data type defined in the *driver/truck* data section.

6.5.4.1.3.1 Annotation (Event Type = 0)

Flag Field Bit	Data Type	Name	Description
27	uint32	Start Timestamp	Start of event
28	uint32	End Timestamp	End of event
16	char_string	Annotation Text	The text of the annotation

Annotation events can occur on the Client either when a driver enters annotation text while using the "View Log" User Interface on a Client, or when invoked by the Server or Client systems. The hard-coded Server/Client system annotation text string data is shown in the table below:

System generated Annotation Text	Event that triggered Annotation to be sent
"Driver logged adverse conditions."	Driver clicked on "Log Adverse conditions"
"System modified driver status from ON-DUTY to OFF-DUTY"	Client system determined sufficient time elapsed to apply OFF-DUTY status due to Passenger Seat
"AOBRD disable FORCED Driver Logout"	Server sent an FMI Enable Fleet Management packet with AOBRD disabled
"AOBRD Remote/Server Logoff Driver"	Client received a driver logout request for driver
"System DETECTED Driver Logout"	Driver initiated logout on Client user interface

6.5.4.1.3.2 Duty Status Change (Event Type = 1)

Flag Field Bit	Data Type	Name	Description
17	char_string	Old Driver Status	The duty status of the driver before the change
18	char_string	New Driver Status	The duty status of the driver after the change
19	uint32	Event ID Number	The status change event counter
20	uint32	Latitude of Change	Latitude when this change occurred
21	uint32	Longitude of Change	Longitude when this change occurred
22	uint32	Verified Time	Timestamp of Log Verified
29	uint8	Client Data 0	Data required for Client log entry
30	uint8	Client Data 1	Data required for Client log entry

The possible values for "*Old Driver Status*" and "*New Driver Status*" are shown below.

Status Code	Meaning
OFF	Off Duty
SB	Sleeper Berth
D	On Duty Driving
ON	On-Duty Not Driving

6.5.4.1.3.3 Log Verified (Event Type = 2)

Flag Field Bit	Data Type	Name	Description
23	time_type	Original Timestamp	The timestamp when the logs were verified

Note: *Server action is required. For the Log Verified event, the Timestamp entry of the Driver/Truck data defined in Section 6.5.4.1.2 represents the time that a duty status change was verified. The Server can use the Original Timestamp supplied in the Log Verified data to locate the previously reported duty status change.*

The Verified Time of the status change can then be set to the Timestamp so that the event is correctly reported as "Verified" when the logs are loaded onto the Client during login.

6.5.4.1.3.4 Log Verification Error (Event Type = 3)

Flag Field Bit	Data Type	Name	Description
23	time_type	Original Timestamp	The timestamp when the logs failed verification

6.5.4.1.3.5 New Shipment Added (Event Type = 4)

Flag Field Bit	Data Type	Name	Description
23	time_type	Original Timestamp	Time of event report
27	uint32	Start Timestamp	Start time entered by driver
28	uint32	End Timestamp	End time entered by driver
25	char_string	Shipper Name	Name of the shipping company
24	char_string	Shipment Document Number	Document number for the new shipment
26	char_string	Commodity	Type of commodity in the shipment

6.5.4.1.3.6 Existing Shipment Modified (Event Type = 5)

Flag Field Bit	Data Type	Name	Description
23	time_type	Original Timestamp	Time of event report
27	uint32	Start Timestamp	Start time entered by driver
28	uint32	End Timestamp	End time entered by driver
25	char_string	Shipper Name	Name of the shipping company
24	char_string	Shipment Document Number	Document number for the new shipment
26	char_string	Commodity	Type of commodity in the shipment

6.5.4.1.3.7 Device Failure Detected (Event Type = 6)

Flag Field Bit	Data Type	Name	Description
None	uint16	Event Error Code	Error Code of Failure
None	time_type	Event Error Time	Timestamp of the failure

Event Error Code	Type	Description
0-3	Internal error	Related to a software or hardware error condition that crashed the Garmin. This error is usually reported after the Garmin crashes then reboots and recovers.
4	Driver database error	A driver's database is determined to be corrupted. Drivers databases are scanned for errors (e.g., missing records, corrupted records) intervals and this error report would occur immediately when errors are detected. An attempt to restore that database from the last known good instance of that database.
5	Self-test error	Recovery error from "Driver database error" listed above, and the driver's database could not be successfully restored, so there are now database errors such as gaps (e.g., missing driver status change

		records) in the status change database.
8	Power-up detected	The HOS software module initializes.
9	Power-down detected	The HOS software module is forced to shut-down.
10	Moving, no driver	Truck has been moving more than 30 seconds with no driver in "Driving" Duty Status. (Note: Not reported if *Auto-Status Driver Update* feature is enabled.)
11	No GPS fix	Current GPS fix does not match a valid map location. This could be reported when a Driver Status Change is being processed, or during the 15-minute driver status (e.g., Status Change to Server with old to new status of Driving to Driving) report is done.

6.5.4.1.3.8 Shipment Deleted (Event Type = 7)

Flag Field Bit	Data Type	Name	Description
23	time_type	Original Timestamp	Time of event
27	uint32	Start Timestamp	Start time entered by driver
28	uint32	End Timestamp	End time entered by driver
25	char_string	Shipper Name	Name of the shipping company
24	char_string	Shipment Document Number	Document number for the new shipment
26	char_string	Commodity	Type of commodity in the shipment

6.5.4.1.3.9 Previous Duty Status Change Modified by System (Event Type = 8)

Flag Field Bit	Data Type	Name	Description
17	char_string	Old Driver Status	The duty status of the driver before the change
18	char_string	New Driver Status	The duty status of the driver after the change
23	uint32	Original Timestamp	The identifier of the status change event
29	uint8	Client Data 0	Data required for Client log entry
30	uint8	Client Data 1	Data required for Client log entry

6.5.4.1.3.10 New Duty Status Change in Past by System (Event Type = 9)

Flag Field Bit	Data Type	Name	Description
17	char_string	Old Driver Status	The duty status of the driver before the change
18	char_string	New Driver Status	The duty status of the driver after the change
23	uint32	Original Timestamp	The identifier of the status change event
29	uint8	Client Data 0	Data required for Client log entry
30	uint8	Client Data 1	Data required for Client log entry

6.5.4.2 New Driver Event Log Record to Server

This protocol allows the Client to send a new driver event log record (shown in Section 6.5.4.1) to the Server. Each record is sent to the Server when the event is detected by the Client.

If the Server cannot be reached then the Client queues each record in the order it occurred. Once Server communication is re-established, each new driver event log record will be sent to the Server in the order as it occurred.

The following are valid event types and event ID's that can be sent to the Server:

0 = Annotation
1 = Duty Status Change
2 = Log Verified
3 = Log Verification Error
4 = New Shipment Added
5 = Existing Shipment Modified
6 = Device Failure Detected
7 = Shipment Deleted
8 = Previous Status Change Modified by System (Passenger Seat Exception)
9 = New Status Change in Past by System (Passenger Seat Exception)

The packet sequence of a *Driver Event Log Record* to Server is shown below:

N	Direction	Fleet Management Packet ID	Hyper-link	Fleet Management Packet Data Type
0	Client to Server	0x0400 – File Transfer Start Packet ID	5.1.13.2.1	file_info_data_type
1	Server to Client	0x0403 – File Start Receipt Packet ID	5.1.13.2.2	file_receipt_data_type
2..n-3	Client to Server	0x0401 – File Data Packet ID	5.1.13.2.3	file_packet_data_type
3..n-2	Server to Client	0x0404 – Packet Receipt Packet ID	5.1.13.2.4	packet_receipt_data_type
n-1	Client to Server	0x0402 – File Transfer End Packet ID	5.1.13.2.5	file_end_data_type
n	Server to Client	0x0405 – File End Receipt Packet ID	5.1.13.2.6	file_receipt_data_type

NOTE: The File Transfer is performed by the generic File Transfer in this document, and can be viewed by clicking the hyperlinks above.

6.5.5 AOBRD Set Odometer Request

This protocol allows the Server to set the odometer value used by the Client. As the Client moves, it will add to the odometer value set by the Server. Anytime the odometer is set by the Server, it will overwrite the previously set odometer value.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1100 –Set Odometer Request	set_odometer _request_data_type

The type definition for the *set_odometer _request_data_type* is shown below. This data type is only supported on Clients that report D610 as part of their protocol support data.

```
typedef struct  /*  D610  */
    {
    uint32      odometer_value;
    } set_odometer_request_data_type;
```

The *odometer_value* is the value, in miles, to set as the odometer value.

6.5.6 Auto-Status Driver Update Protocol

The Auto-Status Driver Update feature has the ability to automatically change a driver's status from "DRIVING" to "ON-DUTY" when a Server programmed time (in seconds) has elapsed following the stop of the vehicle.

For a single driver, the driver's status will change from "ON-DUTY" to "DRIVING" after 30-seconds of the vehicle moving. If two drivers are logged-in and both are "ON-DUTY" then a User Interface message will be displayed asking for one driver to select a "DRIVING" status. **If any driver has "DRIVING" status, no action is needed or performed.**

This protocol allows the Server to enable/disable and program the "Auto-Status Driver Updates" functionality on the Client for all drivers.

Scenario	**Single Driver on PND**	**Two-Drivers on PND**
Movement	Driver in "**On Duty"** status will automatically transition to **"Driving".** If no driver in **"On Duty"** status, PND will prompt user to stop the truck and update status.	Driver in **"On Duty"** status will automatically transition to **"Driving".** However, if both drivers **"On Duty"** or no driver is **"On Duty",** PND will prompt user to stop the truck and update status.
No movement	Driver in "**Driving"** status will automatically transition to **"On Duty".** Driver is informed of the change in status via a popup and will have the opportunity to cancel the auto-status update	Driver in "**Driving"** status will automatically transition to "**On Duty**" status. If a 2nd driver is in **"Passenger Seat"**, this driver will also be moved to **"On Duty"** status in order to comply with FMCSA regulations. Driver(s) are informed of the change in status via a popup and will have the opportunity to cancel the auto-status update.

Note: This setting is a Client setting, which would affect all logged-in drivers on the Client. The default feature state is "Disabled", which will not perform Auto-Status feature functionality.

N	**Direction**	**Fleet Management Packet ID**	**Fleet Management Packet Data Type**
0	Server to Client	0x1300 – Driver Auto Status	auto_status_data_type
1	Client to Server	0x1301 – Driver Auto Status Receipt	auto_status_receipt_data_type

Server to Client type definition for *auto_status_data_type* is shown below:

```
typedef struct  /*  D615  */
    {
    uint16   stop_moving_threshold;    /* Stop moving threshold in seconds request */
    boolean  enable;                   /* 0 = Disable, 1 = Enable feature */
    } auto_status_data_type;
```

Note: The minimum allowable threshold value is 60 seconds (e.g., 1 minute), and the maximum value is 900 seconds (e.g., 15 minutes). The Client will adjust the Server's threshold value to stay within the minimum and

maximum limits. The Client's actual settings will be sent back to the Server in the receipt packet.

The **Client** to **Server** type definition for *auto_status_receipt_data_type* is shown below:

```
typedef struct  /*  D615  */
    {
    uint16   stop_moving_threshold;    /* Stop moving threshold in seconds result */
    boolean  enable;                   /* 0 = Disable, 1 = Enabled feature */
    uint8    result_code;              /* Result code from Client */
    } auto_status_receipt_data_type;
```

Result Code (Decimal)	Meaning
0	Success
1	Threshold too low, so now set to "60"
2	Threshold too high, so now set to "900"
255	Internal error

6.5.7 Adverse Driving Conditions Exemption Protocol

The driver will be given the ability to declare *Adverse Driving Conditions*, which will allow an additional 2-hour driving exemption (if the extension does not violate the short-term ON_DUTY limit).

Once the driver successfully declares the *Adverse Conditions (by clicking on the User Interface button)*, a Client generated Annotation will automatically be sent to the Server, and the Client's violation detection logic will then permit a 2-hour extension to driving time for this driving period.

When the Server receives a driver's *Adverse Driving Conditions Annotation*, the Server should then adjust its rules engine to account for the time extension, and save the *Start_Timestamp* from the received Annotation.

Note: This Start_Timestamp value should then be sent to the Client during any subsequent D615 Property/Passenger Driver Logins (see Section 6.5.1.2.2) or Driver Profile Updates (see Section 6.5.3) sent from the Server if the Adverse Driving Conditions exemption is intended to be used later on the Client. ***See the complete Annotation packet format in the Garmin Fleet Management Interface Control Specification.***

Annotation Type: Adverse Conditions (Event Log Type = 0)

Flag Field Bit	Data Type	Name	Description
27	uint32	Start Timestamp	Time of Adverse Conditions logged
28	uint32	End Timestamp	Time of Adverse Conditions logged
16	char_string	Annotation Text	Text = "Driver logged adverse conditions."

6.5.8 Driver 8-Hour Rule Enable Protocol

This protocol allows the Server to enable or disable the FMCSA 8-Hour Rule functionality of the Client's violation detector. The 8-Hour Rule enforces a 30-minute rest period after 8-hours of driving.

Note: This rule only applies to "Property Carrying" vehicles.

Note: This setting is a Client setting, which would affect all logged-in drivers on the Client. The default feature state is "Enabled", which would enforce the 8-Hour Rule.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1312 – 8-Hour Rule Exemption Enable	eight_hour_exemption_enable_data_type
1	Client to Server	0x1313 – 8-Hour Rule Exemption Enable Receipt	eight_hour_exemption_enable_data_type

The **Server to Client** type definition for the *eight_hour_exemption_enable_data_type* is shown below. This data type is only supported on Clients that report D615 as part of their protocol support data.

```
typedef struct  /*  D615  */
    {
    boolean   new_state;      /* 0 = Disable, 1 = Enable 8-Hour Rule Exemption (default state) */
    } eight_hour_exemption_enable_data_type;
```

A **Client to Server** receipt packet to indicate the success of enabling/disabling of the *8-HourRule Exemption* functionality will be returned back to the Server using the same *eight_hour_exemption_enable_data_type* shown above.

6.5.9 IFTA File Protocols

6.5.9.1 IFTA File Request Protocol

This protocol allows the Server to request driver exported IFTA CSV files stored on the Client. File selection will be based on driver export timestamps, which are created when a driver clicks the IFTA "Export" User Interface button. Once the Server issues this protocol request, the Client locates one or more CSV files. The files will be concatenated into one CSV file. This single file then gets compressed using the GZIP compression algorithm.

Next the Client initiates a File Transfer to the Server using a newly created IFTA File Type (File Type = 4), which transfers the GZIP file to the Server. After the IFTA CSV file transfer process completes, the Client sends a final IFTA file result packet.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0006 – IFTA File Request	ifta_file_request_data_type

The **Server to Client** *IFTA File Request* initial file request is shown below:

```
typedef struct  /*  D615  */
    {
    date_time_t32   start_time;    /* Timestamp for start of range */
    date_time_t32   end_time;      /* Timestamp for end of range */
    } ifta_file_request_data_type;
```

Client to Server File Transfer process occurs next, the File Transfer Protocol packet sequence is listed below as three distinct transfer stages. The first stage initiates the file transfer process:

N	Direction	Fleet Management Packet ID	Hyper-link	Fleet Management Packet Data Type
0	Client to Server	0x0400 – File Transfer Start Packet ID	5.1.13.2.1	file_info_data_type
1	Server to Client	0x0403 – File Start Receipt Packet ID	5.1.13.2.2	file_receipt_data_type

…next, repeat the following Server and Client data stage until all data is sent:

2..n-3	Client to Server	0x0401 – File Data Packet ID	5.1.13.2.3	file_packet_data_type
3..n-2	Server to Client	0x0404 – Packet Receipt Packet ID	5.1.13.2.4	packet_receipt_data_type

…all file data was sent, so end the file transfer sequence as shown below, which ends the File Transfer process:

n-1	Client to Server	0x0402 – File Transfer End Packet ID	5.1.13.2.5	file_end_data_type
n	Server to Client	0x0405 – File End Receipt Packet ID	5.1.13.2.6	file_receipt_data_type

The final **Client to Server** IFTA file result response (sent after the File Transfer completes) is shown below:

1	Client to Server	0x0007 – IFTA File Receipt	ifta_file_response_data_type

```
typedef struct  /*  D615  */
    {
    uint8   result_code;            /* 0 = No errors, non-zero value for errors */
    } ifta_file_response_data_type;
```

Result Code (Decimal)	Meaning
0	Success
1	No IFTA CSV files found
2	Processing busy, try again later
All others	Internal error, try again later

6.5.9.2 IFTA File Delete Protocol

This protocol allows the Server to delete driver exported IFTA CSV files on the Client. The selected files will be based on a range of dates from the Server.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0008 – IFTA Delete File Request	ifta_file_request_data_type
1	Client to Server	0x0009 – IFTA Delete File Receipt	ifta_file_response_data_type

Server to Client *IFTA File Delete Request* packet is shown below:

```
typedef struct  /*  D615  */
    {
    date_time_t32   start_time;    /* Timestamp for start of range */
    date_time_t32   end_time;      /* Timestamp for end of range */
    } ifta_file_request_data_type;
```

Client to Server *IFTA File Delete Receipt* packet is shown below:

```
typedef struct  /*  D615  */
    {
    uint8   result_code;            /* 0 = No errors, non-zero value for errors */
    } ifta_file_response_data_type;
```

Result Code (Decimal)	Meaning
0	Success
1	No IFTA CSV files found
2	Processing busy, try again later
All others	Internal error, try again later

6.5.10 A619 HOS Settings Protocol

This protocol allows the Server to set specific HOS Settings by using one generic FMI Packet ID and one setting structure. The Server selects a HOS Setting (from a published list shown below) and programs the Client's HOS setting with a data value. Each listed HOS Setting can also be enabled or disabled with this generic setting structure.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1500 – HOS Setting	hos_settings_data_type
1	Client to Server	0x1501 – HOS Setting Receipt	hos_settings_receipt_data_type

Server to Client *HOS Settings Protocol* packet is shown below:

```
typedef struct  /*  D619  */
    {
    uint16   setting_selector;   /* HOS Setting Selector (see list below) */
    uint16   settings_value;     /* Data value if applicable for data setting */
    boolean  enable_state;       /* 0 = disable, 1 = enable feature */
    } hos_settings_data_type;
```

Client to **Server** *HOS Settings Protocol Receipt* packet is shown below:

```
typedef struct  /*  D619  */
    {
    uint16   setting_selector;   /* HOS Setting Selector from Server */
    uint16   settings_value;     /* New HOS Setting value found on Client */
    boolean  enable_state;       /* New feature state, 0 = disabled, 1 = enabled */
    uint8    result_code;        /* Result code from Client */
    } hos_settings_receipt_data_type;
```

Each setting is shown in the table below, and then described in a dedicated sub-section below.

Setting Name	Setting Selector	Hyperlink	Description
Auto-Status Driver Update	0	Section 6.5.10.1	Enable/Disable with elapsed time to determine Auto Driver Status
Driver 8-Hour Rule	1	Section 6.5.10.2	Enable/Disable for FMCSA 8-Hour Rule
Periodic Driver Status	2	Section 6.5.10.3	Enable/Disable with elapsed time for Driver Status

6.5.10.1 Auto-Status Driver Update

The Auto-Status Driver Update feature has the ability to automatically change a driver's status from "DRIVING" to "ON-DUTY" when a Server programmed time (in seconds) has elapsed following the stop of the vehicle.

This same functionality is available with a previously released FMI protocol named "Auto-Status Driver Update Protocol" as described in Section 6.5.6.

For a single driver, the driver's status will change from "ON-DUTY" to "DRIVING" after 30-seconds of the vehicle moving. If two drivers are logged-in and both are "ON-DUTY" then a User Interface message will be displayed asking for one driver to select a "DRIVING" status. **If any driver has "DRIVING" status, no action is needed or performed.**

This protocol allows the Server to enable/disable and program the "Auto-Status Driver Updates" functionality on the Client for all drivers.

Scenario	Single Driver on Client's User Interface	Two-Drivers on Client's User Interface
Movement	Driver in "**On Duty"** status will automatically transition to **"Driving".** If no driver in **"On Duty"** status, Client's User Interface will prompt user to stop the truck and update status.	Driver in **"On Duty"** status will automatically transition to **"Driving".** However, if both drivers **"On Duty"** or no driver is **"On Duty",** Client's User Interface will prompt user to stop the truck and update status.
No movement	Driver in "**Driving"** status will automatically transition to **"On Duty".**	Driver in "**Driving"** status will automatically transition to "**On Duty**" status. If a 2nd driver is in

	Driver is informed of the change in status via a popup and will have the opportunity to cancel the auto-status update	**"Passenger Seat"**, this driver will also be moved to **"On Duty"** status in order to comply with FMCSA regulations. Driver(s) are informed of the change in status via a popup and will have the opportunity to cancel the auto-status update.

Note: This setting is a Client setting, which would affect all logged-in drivers on the Client. The default feature state is "Disabled", which will not perform Auto-Status feature functionality.

Note: The minimum allowable threshold value is 60 seconds (e.g., 1 minute), and the maximum value is 900 seconds (e.g., 15 minutes). The Client will adjust the Server's threshold value to stay within the minimum and maximum limits. The Client's actual settings will be sent back to the Server in the receipt packet.

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1500 – HOS Setting	hos_settings_data_type
1	Client to Server	0x1501 – HOS Setting Receipt	hos_settings_receipt_data_type

Server to **Client** *HOS Settings Protocol* packet for "Auto-Status Driver Updates" is shown below:

```
typedef struct  /*  D619  */
    {
    uint16   setting_selector;   /* 0 = Auto-Status Driver Update */
    uint16   settings_value;     /* 60 to 900 (seconds) */
    boolean  enable_state;       /* 0 = disable, 1 = enable feature */
    } hos_settings_data_type;
```

Client to **Server** *HOS Settings Protocol Receipt* for "Auto-Status Driver Updates" is shown below:

```
typedef struct  /*  D619  */
    {
    uint16   setting_selector;   /* 0 = Auto-Status Driver Update */
    uint16   settings_value;     /* Value of setting saved in seconds */
    boolean  enable_state;       /* Current feature state, 0 = disabled, 1 = enabled */
    uint8    result_code;        /* Result code from Client */
    } hos_settings_receipt_data_type;
```

Result Code (Decimal)	Meaning
0	Success
1	Threshold too low, so now set to "60"
2	Threshold too high, so now set to "900"
255	Internal error

Go to HOS Settings selection table, click here> 6.5.10

6.5.10.2 Driver 8-Hour Rule Enable

This protocol allows the Server to enable or disable the FMCSA 8-Hour Rule functionality of the Client's violation detector. The 8-Hour Rule enforces a 30-minute rest period after 8-hours of driving.

This same functionality is available with a previously released FMI protocol named "Driver 8-Hour Rule Enable Protocol" as described in Section 6.5.8.

Note: This rule only applies to "Property Carrying" vehicles.

*Note: This setting is a Client setting, which would affect all logged-in drivers on the Client. The default feature state is "**Enabled**" and would enforce the 8-Hour Rule.*

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x1500 – HOS Setting	hos_settings_data_type
1	Client to Server	0x1501 – HOS Setting Receipt	hos_settings_receipt_data_type

Server to **Client** *HOS Settings Protocol* packet for "Driver 8-Hour Rule Enable" is shown below:

```
typedef struct  /*  D619  */
    {
    uint16   setting_selector;   /* 1 = Driver 8-Hour Rule Enable */
    uint16   reserved;           /* Unused */
    boolean  enable_state;       /* 0 = disable, 1 = enable feature */
    } hos_settings_data_type;
```

Client to **Server** *HOS Settings Protocol Receipt* packet for "Driver 8-Hour Rule Enable" is shown below:

```
typedef struct  /*  D619  */
    {
    uint16   setting_selector;   /* 1 = Driver 8-Hour Rule Enable */
    uint16   Reserved;           /* Unused */
    boolean  enable_state;       /* Current feature state, 0 = disabled, 1 = enabled */
    uint8    result_code;        /* Result code from Client */
    } hos_settings_receipt_data_type;
```

Result Code (Decimal)	Meaning
0	Success
255	Internal error

Go to HOS Settings selection table, click here>6.5.10

6.5.10.3 Periodic Driver Status

This protocol allows the Server to enable or disable the Periodic Driver Status.

*Note: This setting is a Client setting, which would affect all logged-in drivers on the Client. The default feature state is "**Enabled**" with a "**900 second**" (15-minute) setting value.*

Default Setting State	Default Setting Value
Enabled	900 seconds (15 minutes)

Server to **Client** *HOS Settings Protocol* packet for "Periodic Driver Status" is shown below:

```
typedef struct  /*  D619  */
    {
    uint16   setting_selector;   /* 2 = Periodic Driver Status */
    uint16   settings_value;     /* 60 to 65,535 (seconds) */
    boolean  enable_state;       /* 0 = disable, 1 = enable feature */
    } hos_settings_data_type;
```

Client to **Server** *HOS Settings Protocol Receipt* packet for "Periodic Driver Status" is shown below:

```
typedef struct  /*  D619  */
    {
    uint16   setting_selector;   /* 2 = Periodic Driver Status */
    uint16   settings_value;     /* Value of setting saved in seconds */
    boolean  enable_state;       /* Current feature state, 0 = disabled, 1 = enabled */
    uint8    result_code;        /* 0 = Success */
    } hos_settings_receipt_data_type;
```

Result Code (Decimal)	Meaning
0	Success
1	Threshold too low, so now set to “60”
2	Threshold too high, so now set to “65,535”
255	Internal error

Go to HOS Settings selection table, click here> 6.5.10

6.6 Deprecated Protocols

The following protocols are deprecated, and may no longer be supported in the future. It is highly recommended that other existing protocols be used for new development, as they provide functionality equivalent to or better than the protocols in this section.

6.6.1 Text Message Protocols (Deprecated)

6.6.1.1 A603 Client to Server Open Text Message Protocol (Deprecated)

This text message protocol is used to send a simple text message from the Client to the Server. When the Server receives this message, it is required to send a message receipt back to the Client. This protocol is only supported on Clients that report A603 as part of their protocol support data. The packet sequence for the Client to Server open text message is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Client to Server	0x0024 – Client to Server Open Text Message Packet ID	client_to_server_open_text_msg_data_type
1	Server to Client	0x0025 – Client to Server Text Message Receipt Packet ID	client_to_server_text_msg_receipt_data_type

The type definition for the *client_to_server_open_text_msg_data_type* is shown below. This data type is only supported on Clients that report D603 as part of their protocol support data.

```
typedef struct  /* D603 */
    {
    time_type   origination_time;
    uint32      unique_id;
    uchar_t8    text_message[/* variable length, null-terminated string, 200 bytes max */];
    } client_to_server_open_text_msg_data_type;
```

The *origination_time* is the time that the text message was sent from the Client. The *unique_id* is the unsigned 32-bit unique identifier for the message.

The type definition for the *client_to_server_text_msg_receipt_data_type* is shown below. This data type is only supported on Clients that report D603 as part of their protocol support data.

```
typedef struct  /* D603 */
    {
    uint32      unique_id;
    } client_to_server_text_msg_receipt_data_type;
```

The *unique_id* is the unsigned 32-bit unique identifier for the message that the Client sent to the Server.

6.6.1.2 A602 Server to Client Open Text Message Protocol (Deprecated)

This text message protocol is used to send a simple text message from the Server to the Client. When the Client receives this message, it will notify the user and allow the message to be displayed. No additional action will be required from the Client after receiving the text message. This protocol is only supported on Clients that report A602 as part of their protocol support data.

This protocol does not have the capability to report the text message status back to the Server. So, it is recommended you use the A604 Server to Client Open Text Message Protocol if a Client supports both A602 and A604. The packet sequence for the Server to Client open text message is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0021 – Server to Client Open Text Message Packet ID	A602_server_to_client_open_text_msg_data_type

The type definition for the *A602_server_to_client_open_text_msg_data_type* is shown below. This data type is only supported on Clients that report D602 as part of their protocol support data.

```
typedef struct  /* D602 */
    {
    time_type   origination_time;
    uchar_t8    text_message[/* variable length, null-terminated string, 200 bytes max */];
    } A602_server_to_client_open_text_msg_data_type;
```

The *origination_time* is the time that the text message was sent from the Server.

6.6.1.3 Server to Client Simple Okay Acknowledgement Text Message Protocol (Deprecated)

This text message protocol is used to send a simple okay acknowledgement text message from the Server to the Client. When the Client receives this message, it will notify the user and allow the message to be displayed. When the message is displayed, the Client will also display an "Okay" button that the user is required to press after reading the text message.

Once the "Okay" button is pressed, the Client will send an "Okay" acknowledgement message to the Server. This protocol is only supported on Clients that report A602 as part of their protocol support data. The packet sequence for the Server to Client simple okay acknowledgement text message is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0022 – Server to Client Simple Okay Acknowledgment Text Message Packet ID	server_to_client_ack_text_msg_data_type
1	Client to Server	0x0020 – Text message Acknowledgment Packet ID	text_msg_ack_data_type

The type definition for the *server_to_client_ack_text_msg_data_type* is shown below. This data type is only supported on Clients that report D602 as part of their protocol support data.

```
typedef struct  /* D602 */
    {
    time_type   origination_time;
    uint8       id_size;
    uint8       reserved[3];        /* set to 0 */
    uint8       id[/* 16 bytes */];
    uchar_t8    text_message[/* variable length, null-terminated string, 200 bytes max */];
    } server_to_client_ack_text_msg_data_type;
```

The *origination_time* is the time that the text message was sent from the Server. The *id_size* determines the number of characters used in the *id* member. An *id size* of zero indicates that there is no *message id*. The *id* member is an array of 8-bit integers that could represent any type of data.

The type definition for the *text_msg_ack_data_type* is shown below. This data type is only supported on Clients that report D602 as part of their protocol support data.

```
typedef struct  /* D602 */
    {
    time_type   origination_time;
    uint8       id_size;
    uint8       reserved[3];        /* set to 0 */
    uint8       id[/* 16 bytes */];
    uint32      msg_ack_type
    } text_msg_ack_data_type;
```

The *origination_time* is the time that the text message was acknowledged on the Client. The *id_size* and *id* will match the *id_size* and *id* of the applicable text message. The *msg_ack_type* will depend on the type of text message being acknowledged. The table below defines the different values for *msg_ack_type*.

Value (Decimal)	Acknowledgment Type
0	Simple Okay Acknowledgement
1	Yes Acknowledgment
2	No Acknowledgment

6.6.1.4 Server to Client Yes/No Confirmation Text Message Protocol (Deprecated)

This text message protocol is used to send a Yes/No confirmation text message from the Server to the Client. When the Client receives this message, it will notify the user and allow the message to be displayed. When the message is displayed, the Client will also display two buttons (Yes and No). The user is required to press one of the two buttons after reading the text message. Once the user presses one of the two buttons, the Client will send an acknowledgement message to the Server.

This protocol is only supported on Clients that report A602 as part of their protocol support data. The packet sequence for the Server to Client Yes/No confirmation text message is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0023 – Server to Client Yes/No Confirmation Text Message Packet ID	server_to_client_ack_text_msg_data_type
1	Client to Server	0x0020 – Text message Acknowledgment Packet ID	text_msg_ack_data_type

The type definition for the *server_to_client_ack_text_msg_data_type* is shown below. This data type is only supported on Clients that report D602 as part of their protocol support data.

```
typedef struct  /* D602 */
    {
    time_type   origination_time;
    uint8       id_size;
    uint8       reserved[3];        /* set to 0 */
    uint8       id[/* 16 bytes */];
    uchar_t8    text_message[/* variable length, null-terminated string, 200 bytes max */];
    } server_to_client_ack_text_msg_data_type;
```

The *origination_time* is the time that the text message was sent from the Server. The *id_size* determines the number of characters used in the *id* member. An *id size* of zero indicates that there is no *message id*. The *id* member is an array of 8-bit integers that could represent any type of data.

6.6.1.5 StreetPilot Text Message Protocol (Deprecated)

This Protocol was developed on the StreetPilot 3 and StreetPilot 2610\2620 to allow the Server to send a simple text message to the Client. Garmin products which **do not** report A607 or higher as part of their protocols support data will continue to support this protocol if the Server chooses to use it. Unlike the text message protocols described in Section 5.1.5, the Client is not required to have fleet management enabled (See Section 5.1.2) to receive text messages using this protocol.

When the Client receives this message, it will display the text message to the user. The message is removed from the Client once the user is done reviewing it. The packet sequence for the Legacy text message is shown below:

N	Direction	Packet ID	Data Type
0	Server to Client	136 – Legacy Text Message Packet ID	legacy_text_msg_type

The type definition for the *legacy_text_msg_type* is shown below.

```
typedef struct
    {
    char        message[ /* variable length, null-terminated string, 200 characters max */ ];
    } legacy_text_msg_type;
```

6.6.2 Driver ID Monitoring Protocols (Deprecated)

The following driver ID protocols are deprecated. Although they will continue to be supported on Client devices, it is highly recommended that the protocols described in Section 5.1.12.1 be used for new development, as they provide functionality equivalent to or better than the protocols in this section.

6.6.2.1 A604 Server to Client Driver ID Update Protocol (Deprecated)

The A604 Server to Client Driver ID Update Protocol is used to change the driver ID of the current driver on the Client device. This protocol is only supported on Clients that report A604 as part of their protocol support data. The packet sequence for the A604 Server to Client Driver ID Update Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0811 – Server to Client Driver ID Update Packet ID	driver_id_data_type
1	Client to Server	0x0812 – Driver ID Receipt Packet ID	driver_id_receipt_data_type

The type definition for the *driver_id_data_type* is shown below. This data type is only supported on Clients that include D604 in their protocol support data.

```
typedef struct
    {
    uint32     status_change_id;
    time_type  status_change_time;  /* timestamp of status change */
    uchar_t8   driver_id[];         /* variable length, null terminated string, 50 bytes max */
    } driver_id_data_type;
```

The *status_change_id* is a unique number used to identify this status change request. The *status_change_time* is the timestamp when the specified driver ID took effect.

The type definition for the *driver_id_receipt_data_type* is shown below. This data type is only supported on Clients that include D604 in their protocol support data.

```
typedef struct
    {
    uint32   status_change_id;
    boolean  result_code;
    uint8    reserved[3];     /* Set to 0 */
    } driver_id_receipt_data_type;
```

The *status_change_id* identifies the driver ID update being acknowledged. The *result_code* indicates whether the update was successful. This will be *true* if the update was successful or *false* otherwise.

6.6.2.2 A604 Client to Server Driver ID Update Protocol (Deprecated)

The A604 Client to Server Driver ID Update Protocol is used to notify the Server when the driver changes the driver ID via the user interface on the Client. This protocol is only supported on Clients that report A604 as part of their protocol support data. The packet sequence for the A604 Client to Server Driver ID Update Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Client to Server	0x0811 –Client to Server Driver ID Update Packet ID	driver_id_data_type
1	Server to Client	0x0812 – Driver ID Receipt Packet ID	driver_id_receipt_data_type

The type definitions for the *driver_id_data_type* and *driver_id_receipt_data_type* are described in Section 6.6.2.1. These data types are only supported on Clients that include D604 in their protocol support data.

6.6.2.3 A604 Server to Client Driver ID Request Protocol (Deprecated)

The Server to Client Driver ID Request Protocol is used by the Server to obtain the driver ID currently stored in the device. If no driver ID has been set, a zero length string will be returned in the *driver_id_data_type*. This protocol is only supported on Clients that report A604 as part of their protocol support data. The packet sequence for the Server to Client Driver ID Request Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0810 – Request Driver ID Packet ID	None
1	Client to Server	0x0811 – Client to Server Driver ID Update Packet ID	driver_id_data_type
2	Server to Client	0x0812 – Driver ID Receipt Packet ID	driver_id_receipt_data_type

The type definitions for the *driver_id_data_type* and *driver_id_receipt_data_type* are described in Section 6.6.2.1. These data types are only supported on Clients that include D604 in their protocol support data.

6.6.3 Driver Status Monitoring Protocols (Deprecated)

The following driver status protocols are deprecated. Although they will continue to be supported on Client devices, it is highly recommended that the protocols described in Section 5.1.12.3 be used for new development, as they provide functionality equivalent to or better than the protocols in this section.

6.6.3.1 A604 Server to Client Driver Status Update Protocol (Deprecated)

The Server to Client Driver Status Update Protocol is used to change the status of the current driver on the Client device. This protocol is only supported on Clients that report A604 as part of their protocol support data. The packet sequence for the Server to Client Driver Status Update Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0821 – Server to Client Driver Status Update Packet ID	driver_status_data_type
1	Client to Server	0x0822 – Driver Status Receipt Packet ID	driver_status_receipt_data_type

The type definition for the *driver_status_data_type* is shown below. This data type is only supported on Clients that include D604 in their protocol support data.

```
typedef struct
    {
    uint32     status_change_id;    /* unique identifier */
    time_type  status_change_time;  /* timestamp of status change */
    uint32     driver_status;       /* ID corresponding to the new driver status */
    } driver_status_data_type;
```

The *status_change_id* is a unique number which identifies this status update message. The *status_change_time* is the timestamp when the specified driver status took effect.

The type definition for the *driver_status_receipt_data_type* is shown below. This data type is only supported on Clients that include D604 in their protocol support data.

```
typedef struct
    {
    uint32  status_change_id;          /* timestamp of status change */
    boolean result_code;
    uint8  reserved[3]                 /* Set to 0 */
    } driver_status_receipt_data_type;
```

The *status_change_id* identifies the status update being acknowledged. The *result_code* indicates whether the update was successful. This will be *true* if the update was successful or *false* otherwise (for example, the *driver_status* is not on the Client).

6.6.3.2 A604 Client to Server Driver Status Update Protocol (Deprecated)

The Client to Server Driver Status Update Protocol is used to notify the Server when the driver changes the driver status via the user interface on the Client. This protocol is only supported on Clients that report A604 as part of their protocol support data. The packet sequence for the Client to Server Driver Status Update Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Client to Server	0x0821 – Client to Server Driver Status Update Packet ID	driver_status_data_type
1	Server to Client	0x0822 – Driver Status Receipt Packet ID	driver_status_receipt_data_type

The type definitions for the *driver_status_data_type* and *driver_status_receipt_data_type* are described in Section 6.6.3.1. These data types are only supported on Clients that include D604 in their protocol support data.

6.6.3.3 A604 Server to Client Driver Status Request Protocol (Deprecated)

The Server to Client Driver Status Request Protocol is used by the Server to obtain the driver status currently stored in the device. If no driver status has been set, an ID of 0xFFFFFFFF will be returned as the driver status. This protocol is only supported on Clients that report A604 as part of their protocol support data. The packet sequence for the Server to Client Driver Status Request Protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0820 – Request Driver Status Packet ID	None
1	Client to Server	0x0821 – Client to Server Driver Status Update Packet ID	driver_status_data_type
2	Server to Client	0x0822 – Driver Status Receipt Packet ID	driver_status_receipt_data_type

The type definitions for the *driver_status_data_type* and *driver_status_receipt_data_type* are described in Section 6.6.3.1. These data types are only supported on Clients that include D604 in their protocol support data.

6.6.4 Stop Message Protocols (Deprecated)

6.6.4.1 A602 Stop Protocol (Deprecated)

This protocol is used to send Stops or destinations from the Server to the Client. When the Client receives a Stop, it will display it to the user and give the user the ability to start navigating to the Stop location. The Client will not report Stop status (unread, read, active…) for Stops it received using this protocol. This protocol is only supported on Clients that report A602 as part of their protocol support data. The packet sequence for the A602 Stop protocol is shown below:

N	Direction	Fleet Management Packet ID	Fleet Management Packet Data Type
0	Server to Client	0x0100 – A602 Stop Protocol Packet ID	A602_stop_data_type

The type definition for the *A602_stop_data_type* is shown below. This data type is only supported on Clients that report D602 as part of their protocol support data.

```
typedef struct  /* D602 */
    {
    time_type         origination_time;
    sc_position_type  stop_position;
    uchar_t8          text[/* variable length, null-terminated string, 51 bytes max */];
    } A602_stop_data_type;
```

The *origination_time* is the time that the Stop was sent from the Server. The *stop_position* is the location of the Stop. The *text* member contains the text that will be displayed on the Client's user interface for this Stop.

6.6.4.2 StreetPilot Stop Message Protocol (Deprecated)

This Protocol was developed on the StreetPilot 3 and StreetPilot 2610\2620 to allow the Server to send Stop or destination messages to the Client. Garmin devices which **do not** report A607 or higher as part of their protocols support data will continue to support this protocol if the Server chooses to use it. Unlike the Stop protocols described in Section 5.1.5.7, the Client is not required to have fleet management enabled (See Section 5.1.2) to receive Stops using this protocol. When the Client receives a Stop, it will display the Stop to the user and give the user the option to either Save the Stop or start navigating to the Stop. The packet sequence for the Legacy Stop message is shown below:

N	Direction	Packet ID	Data Type
0	Server to Client	135 – Legacy Stop Message Packet ID	legacy_stop_msg_type

The type definition for the *legacy_stop_msg_type* is shown below.

```
typedef struct
    {
    sc_position_type   stop_position;
    char               name[ /* variable length, null-terminated string, 51 characters max */ ];
    } legacy_stop_msg_type;
```

The *stop_position* member is the location of the Stop. The name member will be used to identify the destination through the Client's user interface.

7 Frequently Asked Questions

7.1 Fleet Management Support on Garmin Devices

Q: How can a Partner observe the Garmin FMI protocols without first implementing solutions on a Server?

A: Use the Garmin "Fleet Management Controller" tool (also known as the "FMC" or "PC App") for initial development, which simulates Server connectivity, and can be connected to a Garmin FMI Client device to observe format and sequence of protocol packets outlined in this document. This free tool can be found in the Fleet Management Interface Developer Kit at: http://developer.garmin.com/lbs/fleet-management/

Q: What Garmin devices support the fleet management protocol described in this document?

A: Please visit http://www.garmin.com/solutions/ for the complete list of Garmin devices that support the fleet management protocol described in this document.

Q: My Garmin device displays a message that says "Communication device is not responding. Please check connections".

A: This means that your Garmin device lost connection to the Server and is waiting for the Server to send an Enable Feet Management protocol request. For more information on the Enable Fleet Management protocol request, see Section 5.1.2 of this document.