Garmin Fleet Management Interface Control Specification

May 4, 2011
Drawing Number: 001-00096-00 Rev. G

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.

Technical Support and Feedback:

For technical support or to provide feedback please visit Garmin's Fleet Management web page at http://www.garmin.com/solutions/.

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.

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 for transferal 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

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. Features 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 1 indicates that the feature should be enabled and a 0 indicates that the feature should be disabled. 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

* 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.

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 Product Request Type	2
3-4	Fleet Management Packet ID	0 (decimal)

5	feature_count	2 (decimal)
6	reserved	0 (decimal)
7-8	Enable driver passwords	0x800A (hexadecimal)
9-10	Disable multiple drivers	0x000B (hexadecimal)
11	Checksum	2's complement of the sum of all bytes from byte 1 to byte 4
12	DLE	16 (decimal)
13	ETX	3 (decimal)

The server is required to send the enable fleet management protocol request if any of the following conditions occurs:

1. The cable connecting the server to the client is disconnected and then reconnected

2. Whenever the client powers on

3. Whenever the server powers on

Any attempts to access the fleet management protocols will be ignored by the client until it receives the enable fleet management protocol request after one of the conditions mentioned above occurs. It is possible that the client will not be listening to the communication line, so the server should continue to send the enable fleet management request packet to the client until it gets an ACK for the packet back from the client.

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.

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 to use to send a text message to the client depending 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.

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

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

```
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 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. When the 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.

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 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	A604_server_to_client_open_text_msg_data_type
3	Client to Server	0x0020 – Text Message Acknowledgment Packet ID	text_msg_ack_data_type
4	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. This data type is only supported on clients that report D604 as part of their protocol support data.

```
typedef struct  /*  D604  */
    {
    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 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. This data type is only supported on clients that report D604 as part of their protocol support data.

```
typedef struct  /*  D604  */
    {
    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 that 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 that is not on the client.
4	Canned Response List database full	Wait for the Text Message Acknowledgement packet from a previous Server to Client Canned Response Text Message, then restart the protocol.

The packet ID and type definition for the A604 Server to Client Open Text Message are described in section 5.1.5.1.1 above.

The type definition for the text_msg_ack_data_type is shown below. This data type is only supported on clients that report D602 or D604 as part of their protocol support data.

```
typedef struct  /* D602/D604 */
    {
    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. This data type is only supported on clients that report D604 as part of their protocol support data.

```
typedef struct  /* D604 */
    {
    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.

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

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

```
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. This protocol is only supported on clients that report A607 as part of their protocol support data. 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. This data type is only supported on clients that report D607 as part of their protocol support data.

```
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. This data type is only supported on clients that report D607 as part of their protocol support data.

```
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 5.1.5.1.2).

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.

This protocol is only supported on clients that report A604 as part of their protocol support data. 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. This data type is only supported on clients that report D604 as part of their protocol support data.

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

```
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.

This protocol is only supported on clients that report A604 as part of their protocol support data. 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. This data type is only supported on clients that report D604 as part of their protocol support data.

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

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

```
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. This protocol is only supported on clients that report A607 as part of their protocol support data. 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. This data type is only supported on clients that report D607 as part of their protocol support data.

```
typedef struct /* D603 */
    {
    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.

This protocol is only supported on clients that report A604 as part of their protocol support data. 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. This data type is only supported on clients that report D604 as part of their protocol support data.

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

```
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.

This protocol is only supported on clients that report A604 as part of their protocol support data. 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. This data type is only supported on clients that report D604 as part of their protocol support data.

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

```
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 Packet ID	None
1..n		(Set Canned Message protocols)	

5.1.5.7 Other Text Message Protocols (Deprecated)

The following text message protocols are deprecated. Although they will continue to be supported on client devices, it is highly recommended that the protocols described in section 5.1.5.1 be used for new development, as they provide functionality equivalent to or better than the protocols in this section.

5.1.5.7.1 A603 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. 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.

5.1.5.7.2 A602 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 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.

5.1.5.7.3 Server to Client Simple Okay Acknowledgement Text Message Protocol

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 that is 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

5.1.5.7.4 Server to Client Yes/No Confirmation Text Message Protocol

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.

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 two protocols available to the server for sending Stops to the client, the A603 Stop protocol and the A602 Stop protocol. The A603 Stop protocol allows the client to report the status of a Stop back to the server, while the A602 Stop protocol does not have the Stop status reporting capability. 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.

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. This protocol is only supported on clients that report A603 as part of their protocol support data. 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. 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;
    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 A602 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 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.

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. This protocol is only supported with Stops that were sent to the client using the A603 Stop protocol. This protocol is only supported on clients that report A603 as part of their protocol support data. 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. This data type is only supported on clients that report D603 as part of their protocol support data.

```
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 that is 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. This data type is only supported on clients that report D603 as part of their protocol support data.

```
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 destination. This protocol is only supported on clients that report A603 as part of their protocol support data. 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. This data type is only supported on clients that report D603 as part of their protocol support data.

```
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. This data type is only supported on clients that report D603 as part of their protocol support data.

```
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. This protocol is only supported on clients that report A603 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	0x0220 – Auto-Arrival Data Packet ID	auto_arrival_data_type

The type definition for the auto_arrival_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      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. This protocol is only supported on clients that report A607 as part of their protocol support data. 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. This data type is only supported on clients that report D607 as part of their protocol support data.

```
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 that is 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 type definition for waypoint_receipt_data_type is shown below. This data type is only supported on clients that report D607 as part of their protocol support data.

```
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. This data type is only supported on clients that report D607 as part of their protocol support data.

```
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. This data type is only supported on clients that report D607 as part of their protocol support data.

```
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. This protocol is only supported on clients that report A607 as part of their protocol support data. 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. This data type is only supported on clients that report D607 as part of their protocol support data.

```
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. This protocol is only supported on clients that report A607 as part of their protocol support data. 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. This data type is only supported on clients that report D607 as part of their protocol support data.

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

The cat 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. This data type is only supported on clients that report D607 as part of their protocol support data.

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

The cat 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. This protocol is only supported on clients that report A607 as part of their protocol support data. 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. This data type is only supported on clients that report D607 as part of their protocol support data.

```
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. This data type is only supported on clients that report D607 as part of their protocol support data.

```
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. This protocol is only supported on clients that report A607 as part of their protocol support data. 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. This data type is only supported on clients that include D607 in their protocol support data.

```
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, 30 bytes max */
    } driver_id_d607_data_type;
```

The change_id is a unique number per driver that is 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. This data type is only supported on clients that include D604 in their protocol support data.

```
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 that is being acknowledged. The result_code indicates whether the update was successful. This will be true if the update was successful, false otherwise (for example, the driver_idx is out of range). The driver_idx is the zero-based index of the driver updated. Note that 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. This protocol is only supported on clients that report A607 as part of their protocol support data. 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. These data types are only supported on clients that include D607 in their protocol support data. 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)

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.

5.1.12.2.1 A604 Server to Client Driver ID Update Protocol

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 that is being acknowledged. The result_code indicates whether the update was successful. This will be true if the update was successful, false otherwise.

5.1.12.2.2 A604 Client to Server Driver ID Update Protocol

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 5.1.12.2.1. These data types are only supported on clients that include D604 in their protocol support data.

5.1.12.2.3 A604 Server to Client Driver ID Request Protocol

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 5.1.12.2.1. These data types are only supported on clients that include D604 in their protocol support data.

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 A604 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	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 that is being acknowledged. The result_code indicates whether the update was successful. This will be true if the update was successful, false otherwise (for example, the driver_status is not on the client).

5.1.12.3.1 A604 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 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 5.1.12.5.1. These data types are only supported on clients that include D604 in their protocol support data.

A604 Server to Client Driver Status Request Protocol (see section 5.1.12.5.1) to indicate that the status has not been set.

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

```
typedef struct
    {
    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
    {
    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; false otherwise (for example, the status list already contains the maximum number of items).

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

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

The status_id identifies the list item that is to be deleted.

The type definition for the driver_status_list_item_receipt_data_type is defined in section 5.1.12.3.2 and repeated below. This data type is only supported on devices that report D604 as part of their protocol support data.

```
typedef struct
    {
    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).

5.1.12.3.4 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 /* D604 */
    {
    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 that is being acknowledged. The result_code indicates whether the update was successful. This will be true if the update was successful, 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 that 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)

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.

5.1.12.5.1 A604 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	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 that is being acknowledged. The result_code indicates whether the update was successful. This will be true if the update was successful, false otherwise (for example, the driver_status is not on the client).

5.1.12.5.2 A604 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 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 5.1.12.5.1. These data types are only supported on clients that include D604 in their protocol support data.

5.1.12.5.3 A604 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 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 5.1.12.5.1,. These data types are only supported on clients that include D604 in their protocol support data.

5.1.13 File Transfer Protocols

The following protocols are used to transfer files from the server to the client, and allow the server to obtain information about the files on the client device. Currently, only GPI files may be transferred.

The GPI file may be deleted from the client using the Data Deletion Protocol. See section 5.1.14 for details.

5.1.13.1 GPI File Transfer Protocol

This protocol is used to send a GPI file from the server to the client. In this protocol, the server divides the file into small packets and sends them, one by one, to the client. The client saves the data to a temporary file as it is received, and acknowledges when the data has been received and written. When the entire file has been sent, the client verifies the integrity of the file by verifying the CRC32 checksum and by ensuring the file can be opened.

Please note that locked GPI files are not supported.

For more information about GPI files, see http://www.garmin.com/products/poiloader/ .

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	0x0400 – GPI File Transfer Start Packet ID	gpi_file_info_data_type
1	Client to Server	0x0403 – GPI File Start Receipt Packet ID	gpi_file_receipt_data_type
2..n-3	Server to Client	0x0401 – GPI File Data Packet ID	gpi_file_packet_data_type
3..n-2	Client to Server	0x0404 – GPI Packet Receipt Packet ID	gpi_packet_receipt_data_type
n-1	Server to Client	0x0402 – GPI File Transfer End Packet ID	gpi_file_end_data_type
n	Client to Server	0x0405 – GPI File End Receipt Packet ID	gpi_file_receipt_data_type

The type definition for the gpi_file_info_data_type is shown below. 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  reserved[3]; /* Set to 0 */
    uint8  file_version[/* 16 bytes */];
    } gpi_file_info_data_type;
```

The file_size is the size of the GPI file that will be transferred, in bytes. The maximum file size is limited by the amount of available space on the device. All Garmin devices will have at least 2MB space available for the GPI file. 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 GPI File Information Protocol described in section 5.1.13.2 can be used to retrieve this field from the client. The file_version_length indicates the number of bytes of file_version that are valid.

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

```
typedef struct  /*  D604  */
    {
    uint8  result_code;
    uint8  reserved[3]; /* Set to 0 */
    } gpi_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 (Decimal)	Meaning	Recommended Response
0	Success	None
1	CRC error	The server should restart the entire file transfer.
2	Insufficient space on device	The user of the device should remove any unnecessary files to make space available.
3	Unable to open GPI file	Verify that the file can be opened when transferred to a device using a local connection such as USB or an SD card.
4	No transfer in progress	The server should resend the GPI File Transfer Start packet.

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

```
typedef struct  /*  D604  */
    {
    uint32 offset;           /* offset of this data from the beginning of the file */
    uint8  data_length;      /* length of file_data (0..245)   */
    uint8  reserved[3];      /* Set to 0 */
    uint8  file_data[];      /* file data */
    } gpi_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_data is the actual data to be written to the file.

Note that 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.

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

```
typedef struct /* D604 */
    {
    uint32 offset;           /* offset of data received */
    uint32 next_offset;      /* offset of next data the server should send */
    } gpi_packet_receipt_data_type;
```

The offset is the offset of the file packet received; it is the same as the offset of the corresponding gpi_file_packet_data_type. The next_offset indicates the offset that the server should use when sending the next file packet. 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 gpi_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 gpi_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.

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

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

5.1.13.2 GPI File Information Protocol

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	gpi_file_info_data_type

The type definition for the gpi_file_info_data_type is introduced in section 5.1.13.1 above, and repeated below. 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  reserved[3]; /* Set to 0 */
    uint8  file_version[/* 16 bytes */];
    } gpi_file_info_data_type;
```

The file_size is the size of the file that is 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. This data type is only supported on clients that report D603 or D604 as part of their protocol support data.

```
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 all data relating to fleet management on the client, and disables the fleet management interface on the client.	D604
8	Delete all waypoints created through the Create Waypoint Protocol on the client.	D607

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

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

```
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	0x0134	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.
4095	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. This type is only supported on clients that report D605 as part of their protocol support data.

```
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. This data type is only supported on clients that report D606 as part of their protocol support data.

```
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. This data type is only supported on clients that report D606 as part of their protocol support data.

```
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 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. 2000MPH value, means that 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;
```

5.2 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, please 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 that is 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;
```

Please note that 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 */
    };
```

5.2.5 Legacy Text Message Protocol

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;
```

5.2.6 Legacy Stop Message Protocol

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 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 5.1.6, 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.

6 Appendices

6.1 Packet IDs

A packet ID is an 8-bit unsigned integer type that is used to identify what type of packet that is been 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
Legacy Stop message	135	See Section 5.2.6
Legacy text message	136	See Section 5.2.5
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 that is used to identify what type of fleet management related data that is been 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.

Fleet Management Packet Type	**Value (hexadecimal)**	**Description**	**Server to Client**	**Client to Server**
Enable Fleet Management Protocol Request	0x0000	See Section 5.1.2	X	
Product ID and Support Request	0x0001	See Section 5.1.3	X	
Product ID Data	0x0002	See Section 5.1.3		X
Protocol Support Data	0x0003	See Section 5.1.3		X
Unicode Support Request	0x0004	See Section 5.1.4		X
Unicode Support Response	0x0005	See Section 5.1.4	X	
Text Message Acknowledgement	0x0020	See Section 5.1.5.1		X
Text Message (A602 Open Server to Client)	0x0021	See Section 5.1.5.7.1	X	
Text Message (Simple Acknowledgement)	0x0022	See Section 5.1.5.7.3	X	
Text Message (Yes/No Confirmation)	0x0023	See Section 5.1.5.7.4	X	
Text Message (Open Client to Server)	0x0024	See Section 5.1.5.3		X
Text Message Receipt (Open Client to Server)	0x0025	See Section 5.1.5.3	X	
A607 Client to Server Text Message	0x0026	See Section 5.1.5.5		X
Set Canned Response List	0x0028	See Section 5.1.5.1.2	X	
Canned Response List Receipt	0x0029	See Section 5.1.5.1.2		X
Text Message (A604 Open Server to Client)	0x002a	See Section 5.1.5.1.1	X	
Text Message Receipt (A604 Open Server to	0x002b	See Section 5.1.5.1.1		X

Fleet Management Packet Type	Value (hexadecimal)	Description	Server to Client	Client to Server
Client)				
Text Message Ack Receipt	0x002c	See Section 5.1.5.1.2	X	
Set Canned Response	0x0030	See Section 5.1.5.4.1	X	
Delete Canned Response	0x0031	See Section 5.1.5.4.2	X	
Set Canned Response Receipt	0x0032	See Section 5.1.5.4.1		X
Delete Canned Response Receipt	0x0033	See Section 5.1.5.4.2		X
Request Canned Response List Refresh	0x0034	See Section 5.1.5.4.3		X
Text Message Status Request	0x0040	See Section 5.1.5.2	X	
Text Message Status	0x0041	See Section 5.1.5.2		X
Set Canned Message	0x0050	See Section 5.1.5.6.1	X	
Set Canned Message Receipt	0x0051	See Section 5.1.5.6.1		X
Delete Canned Message	0x0052	See Section 5.1.5.6.2	X	
Delete Canned Message Receipt	0x0053	See Section 5.1.5.6.2		X
Refresh Canned Message List	0x0054	See Section 5.1.5.6.3		X
A602 Stop	0x0100	See Section 5.1.6.2	X	
A603 Stop	0x0101	See Section 5.1.6.1	X	
Sort Stop List	0x0110	See Section 5.1.10	X	
Sort Stop List Acknowledgement	0x0111	See Section 5.1.10		X
Create Waypoint	0x0130	See Section 5.1.11.1	X	
Create Waypoint Receipt	0x0131	See Section 5.1.11.1		X
Delete Waypoint	0x0132	See Section 5.1.11.3	X	
Waypoint Deleted	0x0133	See Section 5.1.11.2	X	X
Waypoint Deleted Receipt	0x0134	See Section 5.1.11.2	X	
Delete Waypoint by Category	0x0135	See Section 5.1.11.4	X	
Delete Waypoint by Category Receipt	0x0136	See Section 5.1.11.4		X
Create Waypoint Category	0x0137	See Section 5.1.11.5	X	
Create Waypoint Category Receipt	0x0138	See Section 5.1.11.5		X
ETA Data Request	0x0200	See Section 5.1.8	X	
ETA Data	0x0201	See Section 5.1.8		X
ETA Data Receipt	0x0202	See Section 5.1.8	X	
Stop Status Request	0x0210	See Section 5.1.7	X	
Stop Status	0x0211	See Section 5.1.7		X
Stop Status Receipt	0x0212	See Section 5.1.7	X	
Auto-Arrival	0x0220	See Section 5.1.9	X	
Data Deletion	0x0230	See Section 5.1.14	X	
User Interface Text	0x0240	See Section 5.1.15	X	
User Interface Text Receipt	0x0241	See Section 5.1.15		X
Message Throttling Command	0x0250	See Section 5.1.17.1	X	
Message Throttling Response	0x0251	See Section 5.1.17.1		X
Message Throttling Query	0x0252	See Section 5.1.17.2	X	
Message Throttling Query Response	0x0253	See Section 5.1.17.2		X
Ping (Communication Link Status)	0x0260	See Section 5.1.16	X	X
Ping (Communication Link Status) Response	0x0261	See Section 5.1.16	X	X
GPI File Transfer Start	0x0400	See Section 5.1.13.1	X	
GPI File Data Packet	0x0401	See Section 5.1.13.1	X	
GPI File Transfer End	0x0402	See Section 5.1.13.1	X	
GPI File Start Receipt	0x0403	See Section 5.1.13.1		X
GPI Packet Receipt	0x0404	See Section 5.1.13.1		X
GPI File End Receipt	0x0405	See Section 5.1.13.1		X
GPI File Information Request	0x0406	See Section 5.1.13.2	X	

Fleet Management Packet Type	**Value (hexadecimal)**	**Description**	**Server to Client**	**Client to Server**
GPI File Information	0x0407	See Section 5.1.13.2		X
Set Driver Status List Item	0x0800	See Section 5.1.12.3.2	X	
Delete Driver Status List Item	0x0801	See Section 5.1.12.3.3	X	
Set Driver Status List Item Receipt	0x0802	See Section 5.1.12.3.2		X
Delete Driver Status List Item Receipt	0x0803	See Section 5.1.12.3.3		X
Driver Status List Refresh	0x0804	See Section 5.1.12.3.4		X
Request Driver ID	0x0810	See Section 5.1.12.1	X	
Driver ID Update	0x0811	See Section 5.1.12.1	X	X
Driver ID Receipt	0x0812	See Section 5.1.12.1	X	X
A607 Driver ID Update	0x0813	See Section 5.1.12.1.1	X	X
Request Driver Status	0x0820	See Section 5.1.12.4	X	
Driver Status Update	0x0821	See Section 5.1.12.4	X	X
Driver Status Receipt	0x0822	See Section 5.1.12.4	X	X
A607 Driver Status Update	0x0823	See Section 5.1.12.4.1	X	X
FMI Safe Mode	0x0900	See Section 5.1.18	X	
FMI Safe Mode Receipt	0x0901	See Section 5.1.18		X
Speed Limit Alert Setup	0x1000	See Section 5.1.19	X	
Speed Limit Alert Setup Receipt	0x1001	See Section 5.1.19		X
Speed Limit Alert	0x1002	See Section 5.1.19		X
Speed Limit Alert Receipt	0x1003	See Section 5.1.19	X	

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 CRC32 Algorithm

The following is the CRC32 algorithm used in 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() */
```

7 Frequently Asked Questions

7.1 Fleet Management Support on Garmin Devices

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 fleet management protocol request. For more information on the enable fleet management protocol request, see Section 5.1.2 of this document.