Miguel A. Morales

mmorales@metocean.com

Summary

An environment, outside of Target Track, where smoke tests and regression tests can be automated.

jalapeno Documentation

TG-100 and TG-300 Test Automation Tool

Contents

[1. Jalapeno 2](#_Toc491772613)

[1.1. Project Location 2](#_Toc491772614)

[1.2. Development Environment 2](#_Toc491772615)

[1.3. Version History 2](#_Toc491772616)

[1.4. Purpose 2](#_Toc491772617)

[2. Serial Connection 3](#_Toc491772618)

[2.1. Session and SerialLink 3](#_Toc491772619)

[2.2. Connection Tasks (DoConnectionTasks) 3](#_Toc491772620)

[2.2.1. Receiving Messages (DoReceiving) 3](#_Toc491772621)

[2.2.2. Sending Messages (DoSending) 5](#_Toc491772622)

[2.2.3. Keep Alive Packets 5](#_Toc491772623)

[3. User Configuration 6](#_Toc491772624)

[3.1. Reading Configurations 6](#_Toc491772625)

[3.2. Sending Configurations 7](#_Toc491772626)

[3.3. Configuration Items 8](#_Toc491772627)

[3.3.1. Mode 9](#_Toc491772628)

[3.3.2. Geofence 9](#_Toc491772629)

[3.3.3. Timefence 9](#_Toc491772630)

[3.3.4. Notifications 9](#_Toc491772631)

[3.3.5. IO Settings 9](#_Toc491772632)

[3.3.6. Iridium 9](#_Toc491772633)

[3.3.7. Cell 9](#_Toc491772634)

[3.4. Editing Configuration items 9](#_Toc491772635)

[4. Messaging 10](#_Toc491772636)

[4.1. Message Base Class 10](#_Toc491772637)

[4.2. Packetizer Class 10](#_Toc491772638)

[4.3. New Messages 11](#_Toc491772639)

[5. *MessageListener* Util Class 11](#_Toc491772640)

[5.1. Flagging Messages 11](#_Toc491772641)

[5.2. *MsgTypeReceived* Events 11](#_Toc491772642)

[5.3. Subscribers 11](#_Toc491772643)

[6. NUnit 12](#_Toc491772644)

[6.1. How-to-start 12](#_Toc491772645)

[6.2. Test Fixtures 12](#_Toc491772646)

[6.3. Test Cases 12](#_Toc491772647)

[6.3.1. Configuration Test Cases 12](#_Toc491772648)

# Jalapeno

## Project Location

The project is found at the following location:

<http://dev.metoceandata.local:8080/svn/Product_Verification/trunk/Jalapeno>

## Development Environment

Jalapeno was built on the .NET framework Version 4.6.01055 using the C# programming language and the Visual Studio 2013 IDE.

## Version History

The Jalapeno program is an automation tool for TG-100 and TG-300 testing. The program’s purpose is to provide an environment, outside of Target Track, where smoke tests and regression tests can be automated. The following table outlines the functions of the program added throughout each iteration.

Table 1: Version History

|  |  |  |  |
| --- | --- | --- | --- |
| **Version #** | **Date** | **Features Added** | **Notes** |
| J-Alpha | 22/08/17 | Serial Connection   * Connects gracefully * Disconnects gracefully   Configurations   * Full device configuration can be read form the device * Single-item configurations can be performed   NUnit   * Test cases added for each configuration item | Visual Studio 2013, NUnit 3 Test Adapter Extension |
| J-Alpha | 29/08/17 | MessageListener Util implemented  Messages added   * RealTimeTrackingMessage * RequestPositionMessage * DiagnosticStatusMessage | Visual Studio 2013, NUnit 3 Test Adapter Extension |

## Purpose

The purpose of this documentation is to provide the necessary insight into the Jalapeno program needed to develop NUnit test cases. This document will outline the methods through which the program establishes a USB connection with the TG device. Furthermore, it will go over the architecture used to send and receive configurations. There will be a focus on writing configurations, as this is the primary objective of the program.

# Serial Connection

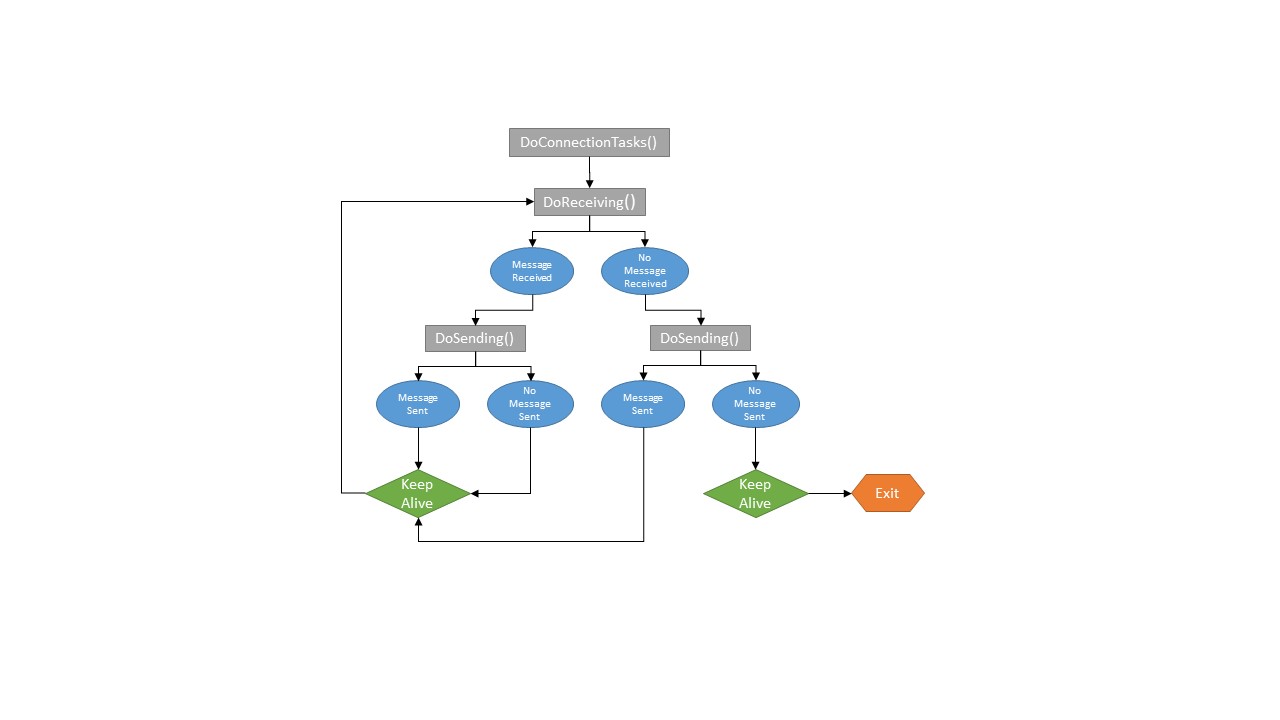
## Session and SerialLink

An instance of *SerialLink* can establish a serial connection via USB using the method *ConnectSerialPort(String ComPortName), where ComPortName* is the name of the COM port (e.g. COM21). *SerialLink* is mostly event-driven, with threads being created as these events are handled. Note that all events are queued in an event queue thread which deploys event threads, this is done using the UtilThread class.

This 2 second timer is controlled by an instance of *UtilTimer*, which is simply a counting thread which publishes the *TimerElapsed* event. Once published, *SerialLink* will attempt to perform its “Connection Tasks”.

## Connection Tasks (DoConnectionTasks)

*SerialLink* performs three main tasks, receiving incoming messages, sending messages, and sending *KeepAwake* packets. A successful completion of one of these tasks is a confirmation that a serial connection is still present. These tasks are performed within the *DoConnectionTasks()* method, which is evoked every two seconds. The method algorithm is represented in the following figure:



*Figure 1: DoConnectionTasks() Flowchart*

### Receiving Messages (DoReceiving)

The serial stream from USB can be decoded into packets with the following format:

Table 2: Serial Packet Format

|  |  |  |
| --- | --- | --- |
| ***FIELD*** | ***SIZE/TYPE*** | ***DESCRIPTION*** |
| **<SOF>** | U8[2] | Start Of Frame Indicator = [ 0xAA, 0x55 ] |
| <**MODEL\_NUM**> | U16 | Binary number should match either the device model number or the broadcast value (0xFFFF) |
| <**SERIAL\_NUM**> | U16 | Binary number should match either the device serial number or the broadcast value (0xFFFF) |
| <**PACKET\_TYPE**> | U8 | Type of data contained in the PACKET\_DATA field |
| **<PACKET\_DATA\_LENGTH>** | U16 | Number of bytes in <PACKET\_DATA> |
| **<PACKET\_DATA>** | U8[PACKET\_DATA\_LENGTH] | Packet data defined by the PACKET\_TYPE field |
| **<CRC>** | U16 | CRC of all data bytes in between <SOF> and <CRC>  Fields, excluding <SOF> and <CRC> fields |
| **<EOF>** | U8[2] | End Of Frame Indicator = [ 0xFF, 0xCC ] |

The point of interest for all packets will be the PACKET\_DATA received. The structure of this data is described in Table 3:

Table 3: PACKET\_DATA Format

|  |  |  |  |
| --- | --- | --- | --- |
|  | ***FIELD*** | ***SIZE/TYPE*** | ***DESCRIPTION*** |
| **<PACKET\_DATA>** | <**CMD\_TYPE**> | U8 | Bit 7 (MSB): Reserved  Bits 0-6: define Command Type |
| **<CMD\_SUB\_TYPE>** | U8 | Bit 7 (MSB): Command (0) / Response (1)  Bits 0-6: defines Command Sub-Type |
| **<PAYLOAD>** | U8[PAYLOAD\_SIZE] | Contents of the payload are defined by Command Type and Command Sub-Type pair. |

#### Complete Messages

All messages received from the TG300 are automatically queued into the *ReceivedData* byte array queue through an event driven sequence. These messages are parsed one at a time by the *DoReceiving()* method. If the *ReceivedData* queue contains a message, a *MessageHandler* object is created to depacketize the received byte array. *MessageHandler* will parse header packet data and confirm the following:

**Complete Message Conditions:**

1. Start of Frame (SOF) is received
2. Correct CRC is received
3. End of Frame (EOF) is received

If these conditions are satisfied, *MessageHandler* will then parse the received data and triage the message with reference to its PACKET\_TYPE:

Table 4: Message PACKET\_TYPE

| **PACKET\_TYPE** | **NUM** | **SOURCE** | **DATA** |
| --- | --- | --- | --- |
| **APP\_DATA** | 4 | Any | One full message from Messaging Layer. |
| **KEEP\_ALIVE** | 5 | Software | None |
| **CONFIRMATION** | 6 | Device | None |
| **NOTIFICATION** | 9 | Device | A notification message (See “Notifications” section) |

#### Incomplete Messages

If an SOF is detected but no EOF is detected, this is determined to be an incomplete message. *Messages* are sometimes received in various packets either intentionally or unintentionally; *SerialLink* will attempt to stitch these message “fragments” together. When the first fragment (SOF fragment) is received, a flag is raised to alert *SerialLink* that it should listen for other fragments. The following situations may arise:

1. The next message received meets the complete message conditions (Section 2.2.1.1); it is not a fragment. Message is consumed, data is parsed. Incomplete message flag is still raised.
2. The next message received does not begin with an SOF and does not end with an EOF; this message is determined to be a fragment. Fragment is stored in thequeue and is not parsed. Incomplete message flag is still raised.
3. The next message received does not begin with an SOF but does end with an EOF. This message is determined to be the EOF fragment. An EOF fragment indicates that the entire fragmented message has been received. Fragments are stitched together into a single array, the message meets the complete message conditions (Section 2.2.1.1), the message is consumed, and the data is parsed. Incomplete message flag is lowered.
4. The next message received does not begin with an SOF but does end with an EOF. This message is determined to be the EOF fragment. An EOF fragment indicates that the entire fragmented message has been received. Fragments are stitched together into a single array; the message does NOT meet the complete message conditions (Section 2.2.1.1). The message is discarded. Incomplete message flag is lowered.

### Sending Messages (DoSending)

*SerialLink* contains a queue named *MessagesToSend* which is checked during connection tasks. If a message is pushed to this queue (from anywhere), it will be sent to the device in a FIFO sequence. The ZylSerialPort DLL is used to send these messages by evoking the *SendByteArray()* method. All messages sent follow the Message Packet Format.

### Keep Alive Packets

When the serial connection is idle for more than 5 seconds the device goes to sleep mode. *SerialLink* is responsible for keeping this serial connection alive. *SerialLink* does this by sending *KeepAliveMessage* (derived from the *Message* class)every 2 seconds. In turn, the device sends confirmation packets to indicate that these KeepAlive messages have been received. If three consecutive message have not received confirmation, the serial connection is severed.

# User Configuration

## Reading Configurations

Configurations are read from the device by sending the following command:

Table 5: Read Configuration Message Format \_

| **Description** | **<CMD\_TYPE>** | **<CMD\_SUB\_TYPE>** | **<PAYLOAD>** |
| --- | --- | --- | --- |
| Read Configuration | 0x05 | 0x01 (command) | None |
| 0x05 | 0x81 (response) | CONFIGURATION\_PACKET\_DATA |

Once the command is received successfully, the device will send its configuration over the serial connection. The configuration will be sent over multiple packets, each following this following format:

Table 6: Write Configuration Message Format

| **Description** | **<CMD\_TYPE>** | **<CMD\_SUB\_TYPE>** | **<PAYLOAD>** |
| --- | --- | --- | --- |
| Write Configuration | 0x05 | 0x02 (command) | CONFIGURATION\_PACKET\_DATA [N] |
| 0x05 | 0x82 (response) | U8 ResultCode  CONFIGURATION\_PACKET\_DATA [N] |

Table 7: CONFIGURATION\_PACKET\_DATA Format

|  |  |  |
| --- | --- | --- |
| CONFIGURATION\_PACKET\_DATA | | |
| U32 | ConfigCRC | 16 bit CRC stored in lowest half of a U32 field, significant bytes are zeroes |
| U16 | Offset | Byte offset of the starting point in the T\_FULL\_CONFIGURATION structure that the Data begins at. |
| U8 | Length | Length of the configuration Data field. |
| U8[] | Data | Data to be copied to the cache |
| Size = 7 + Length bytes | | |

It has been observed that these configuration packets all arrive as fragments. Therefore, *MessageHandler* stitches them together into a large “message” which contains various configuration messages. See the following example; observe the various SOF and EOF.

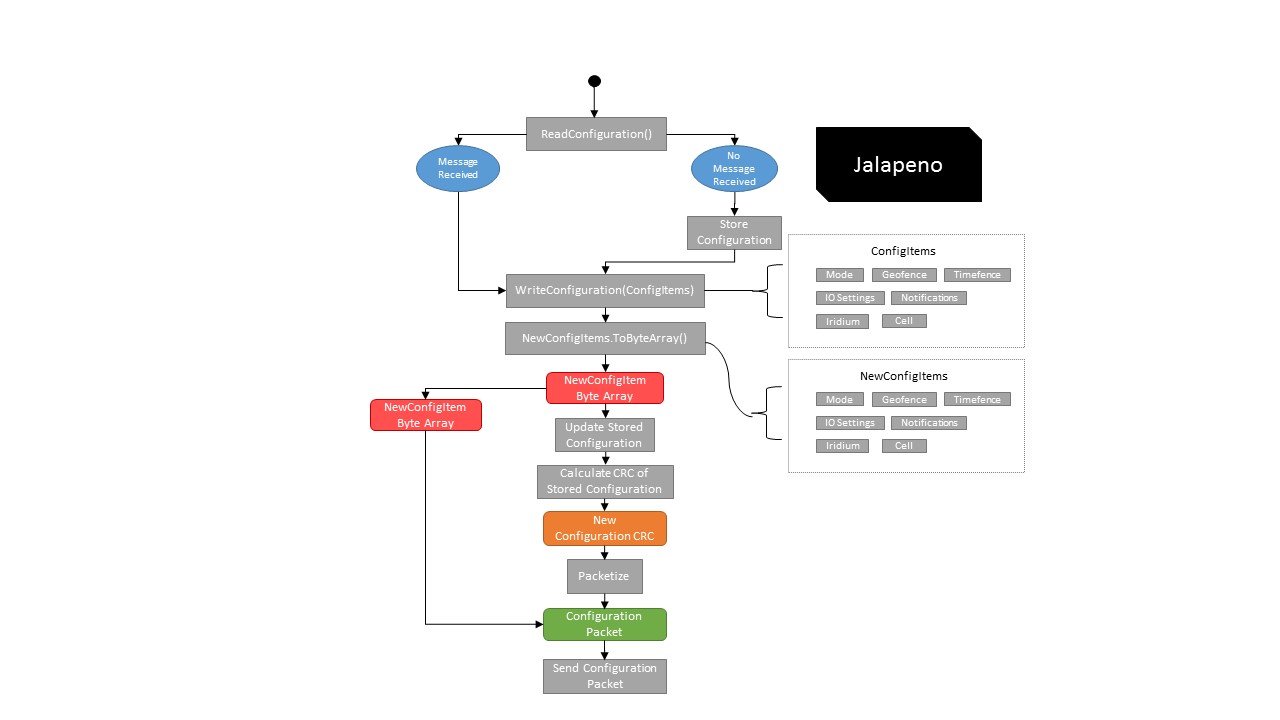
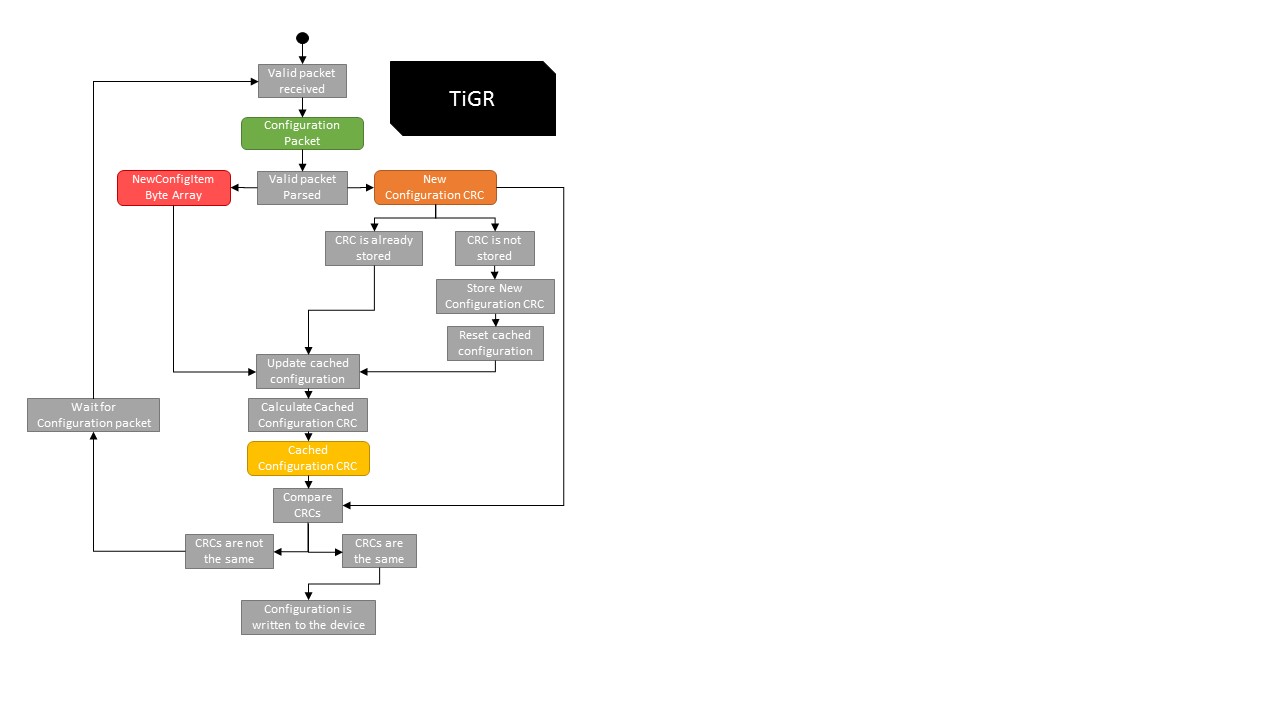
Received:AA-55-05-00-0C-00-04-9E-00-05-81-83-49-00-00-00-00-2C-01-00-00-00-00-00-00-2C-01-00-00-58-02-00-00-40-02-FF-01-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-3C-00-00-00-00-00-00-00-2C-01-00-00-00-00-00-00-40-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-3C-00-00-00-00-00-00-00-2C-01-00-00-00-00-00-00-40-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-3C-00-00-00-00-00-00-00-2C-01-00-00-6C-57-FF-CC-AA-55-05-00-0C-00-06-00-00-E2-B1-FF-CC-AA-55-05-00-0C-00-04-9E-00-05-81-83-49-00-00-96-00-00-00-00-00-40-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-3C-00-00-00-00-00-00-00-2C-01-00-00-00-00-00-00-40-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-3C-00-00-00-00-00-00-00-2C-01-00-00-00-00-00-00-40-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-54-65-73-74-00-00-00-00-02-5D-87-05-0A-6F-40-79-04-24-BE-29-0A-10-4E-7C-AA-EF-FF-CC-

*Messagehandler* recognizes these messages as configuration messages and parses through the large “message” in order to separate it into the expected packets seen in Figure 1. This is done within the *MessagePacketData* class, instanced within *MessageHandler* as *PacketData*. The payload data is extracted from each of these packets and then stitched together using the *MessagePayloadData* class. The CRC of this entire configuration is verified. *MessageHandler* will then pass the configuration byte array held by *PayloadData* back to *SerialLink* so that it may update its *CurrentConfiguration* object.

*CurrentConfiguration* will only overwrite its stored configuration if the configuration byte array is the expected length and does not match the stored configuration. *CurrentConfiguration* will parse through the data and populate its configuration item objects with the appropriate data. A copy of the entire byte array is also stored within *ConfigConfiguration*.

## Sending Configurations

Configurations can be done in two different scopes. The first is a simple single-item configuration, performed with a single packet. The TiGR accepts configurations based on configuration session CRCs; the device has a buffer with a copy of its stored configuration as well as the CRC for this configuration. Single-item configurations can be done by sending the byte sequence for the specific changes necessary. This single-item configuration must be accompanied by the length of the data to be written, the position of this data within the configuration array, and the updated configuration CRC. The device will then take this configuration data, overwrite the data within its buffer at the appropriate position, and recalculate the configuration CRC. If this calculated configuration CRC and the received configuration CRC are identical, the configuration changes will be performed. If the CRC does not match, the configuration is unchanged but the configuration change remains within the device buffer. The following flowcharts depict the process followed when writing a configuration, both for Jalapeno and the TG-100 or TG-300.

*Figure 2: Writing Configurations (Jalapeno) Figure 3: Writing Configurations (TiGR)*

When a multi-item configuration is performed, the process is similar. Packets for each configuration item change must be sent to the device. Each packet must contain the length of the data to be written, the position of this data within the configuration array, and the updated configuration CRC calculated after all items are updated. The configuration CRC attached to multi-item configuration packets will be referred to as the configuration session CRC.

If two items must be configured, the following steps will be followed:

1. First configuration packet is sent with the configuration session CRC attached.
2. The device will receive the first packet, updates its buffer to include the new changes, and calculates the configuration CRC. This configuration CRC does not match the received configuration session CRC. The configuration changes are not performed. The buffer is not reset. The configuration session CRC is stored.
3. Second configuration packet is sent with the configuration session CRC attached.
4. The device will receive the second packet. Since the configuration CRC attached matches the stored configuration session CRC, the device recognize that the packet is part of a multi-item configuration. The device buffer is updated to include the new changes, and calculates the configuration CRC. This configuration CRC now matches the received configuration session CRC. The configuration changes are now performed. The buffer now contains a copy of the current configuration. The configuration is successfully written.

## Configuration Items

Within the *Jalapeno.Config* namespace, all configuration item classes are held. They are as follows:

Table : Configuration Items Within *FullConfiguration*

|  |  |  |
| --- | --- | --- |
| FullConfiguration (TG-300) | | |
| **Configuration Item Class Name** | **Object Name within Stored Config.** | **Item Byte Array Size** |
| Mode | DefaultMode | 46 bytes |
| Mode | AdvancedModes 0 - 4 | 46 \* 5 = 230 bytes |
| Geofence | ConfigGeofence 0 - 9 | 25 \* 10 = 250 bytes |
| Timefence | ConfigTimefence 0 - 2 | 17 \* 3 = 51 bytes |
| IOSettings | ConfigIOSettings | 2 bytes |
| Notifications | ConfigNotifications | 11 bytes |
| Iridium | ConfigIridium | 32 bytes |
| Cell | ConfigCell | 186 bytes |
| U8[20] | Reserved | Reserved bytes. Set all to zeroes. |
| **Size = 828 bytes** | | |

The Jalapeno representation of these configuration items is meant to mirror the structure detailed in the ICD. The following section will detail the Jalapeno representation of these configuration items.

Each configuration item contains a method called *ToByteArray()* which returns the configuration item byte array. In addition, each configuration item has a read method (e.g. *readTimefence(), readGeofence()*) which will take a configuration byte array and populate the configuration item parameters within the progam. ToByteArray() is usually used when preparing a write configuration message (which will be sent to the device), and read methods are usually used when reading a configuration from the device.

### Mode

See *Section 4.1 Operating Modes* in the ICD for details on the data structure that defines this configuration item.

### Geofence

See *Section 4.3 Geofences* in the ICD for details on the data structure that defines this configuration item.

### Timefence

See *Section 4.4 Timefences* in the ICD for details on the data structure that defines this configuration item.

### Notifications

See *Section 4.5 Notifications* in the ICD for details on the data structure that defines this configuration item.

### IO Settings

See *Section 4.6 I/O Subsystem* in the ICD for details on the data structure that defines this configuration item.

### Iridium

See *Section 4.7 Iridium Subsystem* in the ICD for details on the data structure that defines this configuration item.

### Cell

See *Section 4.9 Cell Subsystem* in the ICD for details on the data structure that defines this configuration item.

## Editing Configuration items

Configuration items to be changed are edited by changing the configuration item objects within the *NewConfigurationItems* class instanced within *FullConfiguration* as *ConfigItems*. Whenever configuration item changes are sent, the configuration item will be a copy of the configuration items within *ConfigItems*. Here is an example of a geofence configuration being sent:

1. Location GeoLoc = **new** Location(78.134493, -14.944785, -9.492188, 79.804688);
3. Geofence NewGeofence = **new** Geofence("NewTest", Geofence.GeofenceTypeEnum.GEOFENCE\_DISABLED, GeoLoc);
5. SerialLink.CurrentConfiguration.ConfigItems.NewConfigGeofence = NewGeofence;
7. Program.CurrentSession.WriteConfig(WriteConfigurationMessage.ConfigurationItemsEnum.GEOFENCE\_0);

As can be observed above, the appropriate *ConfigItems* object (*NewGeofence*) is being changed before writing the desired geofence to the device.

# Messaging

## Message Base Class

All message classes/objects are derived from the *Message* base class. All messages queued in the outgoing message queue, using the method described in Section 2.2.2, must be derived from the *Message* class to be sent properly. *Messag*e utilizes a *Packetizer* object to build message packets that conform to the necessary format.

1. **public** **class** Message
2. {
3. **public** Packetizer MessagePacketizer;
4. **public** **byte**[] MessagePacket {**get**; **set**;}
6. **public** Message()
7. {
8. MessagePacketizer = **new** Packetizer();
9. }
10. }

## Packetizer Class

*Packetizer* is a class which should be evoked whenever an outgoing packet is created. This class provides the tools to create a message in a modular fashion. *Packetizer* builds messages using a byte array queue, pushing pieces of the message to the queue until the message is complete. Once the message is finished, the queue is compiled into a single byte array. See below for some examples of the *Packetizer* method calls used to build *ReadConfigurationMessage*:

1. **public** **class** ReadConfigurationMessage : Message
2. {
3. **public** ReadConfigurationMessage(): **base**()
4. {
5. MessagePacketizer.PacketType = MessageHandler.PacketTypeEnum.AppData;
6. MessagePacketizer.CommandType=MessagePacketData.CommandTypeEnum.READ\_CONFIGURATION;
8. MessagePacketizer.addCommandType();
9. MessagePacket = MessagePacketizer.CompleteMessage();
10. }
11. }

The *Packetizer* constructor will add the “start” of the packet header (Seen in Table 2), which includes the SOF, broadcast values, a *PacketType* placeholder, and a *DataLength* placeholder.

Within this derived class, the *PacketType* is defined. If the *PacketType* is App Data, the *CommandType* is also defined. Using *Packetizer*, the *CommandType* is added to the message.

For this message, this will be all that is required; no further message data is need. The message is completed by calling *Packetizer*’s *CompleteMessage()* method. *CompleteMessage* will add the “end” of the packet header, which includes the CRC placeholder and the EOF. The message byte array is then compiled. *CompleteMessag*e will retroactively add the *DataType*, the calculated *DataLength*, and the CRC to the message byte array. A complete message byte array is returned.

## New Messages

In order to create new message classes, this same inheritance can be used. It must be noted that since *Packetizer* uses a queue, an understanding of packet order is needed to make the appropriate method calls in the correct order. See the *WriteConfigurationMessage* class, also derived from *Message*, to see a similar implementation. If a new App data message is being added, changes to the *addCommandType()* method within *Packetizer* must be done to include it.

Note: To recognize new incoming messages, *MessageHandler*’s *AssessPacketType()* method must be updated to triage these messages accordingly. This is done by changing the enumerations within *MessagePacketData* and the *getCommandType* method.

ADD SESSION METHOD

# *MessageListener* Util Class

In order to be able to confirm the success receipt of certain messages, the *MessageListener* class was developed to listen for message replies.

## Flagging Messages

Messages can be flagged by setting *PacketTypeFlagged* or *CommandTypeFlagged* to the desired enumeration. Although these can be changed directly, it is better to flag these messages by calling the *ListenForCommandType*() and *ListenForCommandType*() methods that belong to *Session*.

## *MsgTypeReceived* Events

*MessageListener* works by subscribing to *MsgTypeReceived* events, which are raised whenever a message is received. This event is accompanied by information concerning the packet type and command type of the received message.

## Subscribers

MessageListener has two methods subscribed to the *MsgTypeReceived* events, OnPacketTypeReceived() and OnCommandTypeReceived(). When the user asks *MessageListener* to listen for a message type, these functions assess each received event and determine if the desired command type or packet type was received.

Here is an example of a simple implementation of *MessageListener:*

1. Program.CurrentSession.RequestDiagnostics();
2. Program.CurrentSession.ListenForCommandType(Jalapeno.Messaging.MessagePacketData.CommandTypeEnum.DIAGNOSTIC\_STATUS\_REPLY);

# NUnit

NUnit is a unit-testing framework for all .NET languages. It is used with the Jalapeno project to create test fixtures, as well as test cases.

## How-to-start

For NUnit to integrate with the Visual Studio IDE, it is necessary to install the NUnit 3 Test Adapter Extension. Once this is done, the test cases designed within the Jalapeno project will appear within the Test Window.

Note: NUnit does not use *Main()* as an entry point. This means that an instance of Session must be declared as a class member within Program in order to have an existing serial connection when running tests. See below:

1. **namespace** Jalapeno
2. {
3. **public** **class** Program
4. {
5. **public** **static** Session CurrentSession = **new** Session("COM7");
7. **public** **static** **void** Main(**string**[] args)
8. {
9. **while** (CurrentSession.TG300.Connected) ;
10. }
11. }
12. }

## Test Fixtures

Test fixtures, within NUnit, mark a class that contains tests and, optionally, setup or teardown methods. It is defined by using the attribute [TextFixture].

## Test Cases

Test cases are defined as methods, within a [TextFixture] class, that have a “[Test]” attribute.

### Configuration Test Cases

Test cases involving configuration changes must follow the steps outlined below:

1. Read a valid configuration (Done through the *ReadConfig()* method within *the TestTools namespace*).
2. Assign the desired configuration changes to *ConfigItem* within *FullConfiguration*.
3. Write the configuration using Session’s *WriteConfig()* method.
4. Repeat step 1.
5. Perform the assertion by using the *StructuralComparisons.StructuralEqualityComparer.Equals()* method and evoking the *ToByteArray()* for the expected configuration item (*ConfigItem*) and the actual configuration item (*FullConfiguration* class members).

The following is an example:

1. [TestFixture (Category = "Mode")]
2. **class** ModeSmokeTests
3. {
4. [SetUp]
5. **public** **void** RunBeforeAnyTests()
6. {
7. TestTools.ReadConfig();
8. }
9. [Test]
10. **public** **void** CanWriteMode()
11. {
12. FullConfiguration.ConfigItems.NewConfigMode.setOperationFlags(Mode.GPSModeEnum.MOTION\_ONLY, **false**, **true**, Mode.CellModeEnum.STANDBY);
13. FullConfiguration.ConfigItems.NewConfigMode.setGpsFixInterval(300);
14. FullConfiguration.ConfigItems.NewConfigMode.setMailboxCheckInterval\_NoMotion(300);
15. FullConfiguration.ConfigItems.NewConfigMode.setMailboxCheckInterval\_Motion(600);
16. FullConfiguration.ConfigItems.NewConfigMode.setTransmitInterval(300);
18. Program.CurrentSession.WriteConfig(WriteConfigurationMessage.ConfigurationItemsEnum.DEFAULT\_MODE);
20. TestTools.ReadConfig();
22. Assert.AreEqual(**true**, StructuralComparisons.StructuralEqualityComparer.Equals(FullConfiguration.ConfigItems.NewConfigMode.ToByteArray(), Program.CurrentSession.TG300.CurrentConfiguration.DefaultMode.ToByteArray()));
23. }

In the above example, the [SetUp] attribute is used to ensure that all [Tests] within a specific [TextFixture] read a valid configuration.

Note: Some tests might require a wait period before any operations can be performed. This is because the device takes a second or two to establish a steady, responsive, serial connection.