DashBoard

openGear™ Software Development Guide
For DashBoard v9.5

ROSS LIVING LIVE!

Thank You for Choosing Ross

You've made a great choice. We expect you will be very happy with your purchase of Ross Technology.

Our mission is to:

1. Provide a Superior Customer Experience
   - offer the best product quality and support
2. Make Cool Practical Technology
   - develop great products that customers love

Ross has become well known for the Ross Video Code of Ethics. It guides our interactions and empowers our employees. I hope you enjoy reading it below.

If anything at all with your Ross experience does not live up to your expectations be sure to reach out to us at *solutions@rossvideo.com*.

David Ross
CEO, Ross Video
dross@rossvideo.com

Ross Video Code of Ethics

Any company is the sum total of the people that make things happen. At Ross, our employees are a special group. Our employees truly care about doing a great job and delivering a high quality customer experience every day. This code of ethics hangs on the wall of all Ross Video locations to guide our behavior:

1. We will always act in our customers' best interest.
2. We will do our best to understand our customers' requirements.
3. We will not ship crap.
4. We will be great to work with.
5. We will do something extra for our customers, as an apology, when something big goes wrong and it's our fault.
6. We will keep our promises.
7. We will treat the competition with respect.
8. We will cooperate with and help other friendly companies.
9. We will go above and beyond in times of crisis. *If there's no one to authorize the required action in times of company or customer crisis - do what you know in your heart is right. (You may rent helicopters if necessary.)*

openGear™ Software Development Guide

- Ross Part Number: **8200DR-006-9.5**
- Release Date: **November 8, 2022**.
- DashBoard Software Issue: **9.5**

The information contained in this Guide is subject to change without notice or obligation.

Copyright

Patents

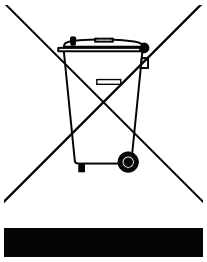
Ross Video products are protected by patent numbers US 7,034,886; US 7,508,455; US 7,602,446; US 7,802,802 B2; US 7,834,886; US 7,914,332; US 8,307,284; US 8,407,374 B2; US 8,499,019 B2; US 8,519,949 B2; US 8,743,292 B2; GB 2,419,119 B; GB 2,447,380 B. Other patents pending.

Environmental Information

The equipment that you purchased required the extraction and use of natural resources for its production. It may contain hazardous substances that could impact health and the environment.

To avoid the potential release of those substances into the environment and to diminish the need for the extraction of natural resources, Ross Video encourages you to use the appropriate take-back systems. These systems will reuse or recycle most of the materials from your end-of-life equipment in an environmentally friendly and health conscious manner.

The crossed-out wheeled bin symbol invites you to use these systems.



If you need more information on the collection, reuse, and recycling systems, please contact your local or regional waste administration.

You can also contact Ross Video for more information on the environmental performances of our products.

Company Address

Ross Video Limited
8 John Street
Iroquois, Ontario
Canada, K0E 1K0

Ross Video Incorporated
P.O. Box 880
Ogdensburg, New York
USA 13669-0880

General Business Office: (+1) 613 • 652 • 4886
Fax: (+1) 613 • 652 • 4425

Technical Support: (+1) 613 • 652 • 4886
After Hours Emergency: (+1) 613 • 349 • 0006

E-mail (Technical Support): techsupport@rossvideo.com
E-mail (General Information): solutions@rossvideo.com
Website: http://www.rossvideo.com

Contents

OGP Messaging 5

OGP Reference 6

File Upload 7

Introduction

About this Guide

This guide is part of the openGear Development Guide. There are two parts to the guide:

- ***openGear Development Guide Part I – Hardware Specifications (8200DR-005-xx)***
- ***openGear Development Guide Part II – Software (8200DR-006-xx)*** (this guide)

This guide describes the software elements of the openGear ecosystem. The following major subjects are covered:

- **DashBoard Connect** – A specification for connecting networked devices together, allowing control, setup and monitoring through DashBoard client software.
- **openGear Protocol (OGP)** – A protocol for parameter reporting, control and alarm reporting.
- **openGear XML** – A specification for defining all parameters for an openGear device in XML notation.
- **DashBoard JSON** – A specification for defining all parameters, reporting, control and alarm reporting in JSON notation.

Related Documentation

Documents and resources that describe technologies related to the openGear ecosystem include the following:

- ***openGear Development Guide, Part I – Hardware Specifications (8200DR-005-xx)***.

  Describes electrical and mechanical hardware specifications for openGear cards and the openGear frame.
- ***DashBoard CustomPanel Development Guide (8351DR-007- xx)***.

  Provides information about creating CustomPanels, including coding CustomPanels using OGLML and ogScript.
- ***Introduction to the Controller Area Network (CAN)***. Texas Instruments Application Report SLOA101, August 2002.

  An overview of the CAN hardware and protocol specification.
- ***The CAN Protocol***. Kvaser Technical Document. See http://www.kvaser.com/can/protocol/.

  An overview of the CAN hardware and protocol specification.
- ***Controller Area Network***. ISO 11898.

  An overview of the CAN hardware and protocol specification.
- ***Atmel AT90CAN128 Datasheet, Rev 7679E-CAN 07/07***.

  Describes CAN implementation on the Atmel AT90CAN128 microcontroller (used on many openGear cards).
- ***Essential SNMP (2nd edition)*** by Mauro, D.R & K.J. Schmidt. O'Reilly, 2009.

  A good introduction to the Simple Network Management Protocol (SNMP).

System Architecture

Overview

The openGear ecosystem provides an environment where devices can be controlled and monitored in a unified environment using DashBoard. The environment consists of clients, such as DashBoard, and devices. Communication is achieved primarily through use of openGear Protocol (OGP).

Device and Slot Model

OGP was originally designed as a binary protocol to communicate between DashBoard and openGear frames and cards. As such, OGP is built on a model of devices within devices. Each sub-device has a unique address within the parent device. These correspond to the slots within an openGear frame, with each sub-device being the individual cards within the frame. The frame itself is addressed at slot 0 (the address of the Frame Network Controller). Most DashBoard Connect devices need only be represented as a device with a single sub-device at address 0.

DashBoard Connect

DashBoard Connect is a specification for how any networked device communicates with DashBoard in the openGear ecosystem. It outlines methods for device discovery, connection, session management and data transport.

There are several components that may participate in the system:

- DashBoard – control and monitoring client
- DashBoard Connect devices – any networked device presented in DashBoard
- openGear Frames
- openGear Cards

openGear Communication Protocol (OGP)

OGP a simple messaging protocol which supports control, status monitoring, and maintenance (e.g. software upgrade) of connected devices. Each device must implement this protocol in order to take advantage of the remote control capabilities provided by DashBoard.

OGP may be implemented using either the legacy binary protocol or via JSON representation. Details of both implementations are documented in this guide.

The openGear frame includes a Frame Network Controller; this card manages the TCP/IP connection to DashBoard, as well as provides a simple Controller Area Network bus interface (CANbus) to openGear cards installed within that frame. Dashboard Connect devices must implement their own OGP server to communicate with DashBoard via TCP/IP.

DashBoard

DashBoard is a user interface application which provides an environment for openGear and DashBoard Connect devices to interact with the user. It is an open platform for facility control and monitoring that enables users to quickly build unique, tailored CustomPanels to make complex operations simple.

There may be multiple DashBoard clients active within a system. Each device is responsible for maintaining communication with all connected DashBoard clients and must broadcast all parameter changes to each.

DashBoard Connect Devices

Any TCP/IP-connected device may interface with DashBoard using DashBoard Connect. This guide provides more details on implementation of DashBoard Connect. There are specific methods for discovery, connection and communication that DashBoard Connect devices must follow in order to properly interface with DashBoard. DashBoard Connect devices maintain an internal data model and use OGP (either binary or JSON format) for reporting and updating of that model.

openGear Frames

openGear Frames are 2-RU rack-mounted frames which can house up to 20 openGear cards. There are several models of openGear frame, including:

- DFR-8310 – 10-slot frame (legacy)
- DFR-8320 / DFR-8321 – 20-slot frame (legacy)
- OG3-FR – 20-slot high-density modular frame
- OGX-FR – 20-slot high density modular frame, adapted to UHD and IP infrastructures

Each frame includes a Frame Network Controller in addition to the openGear card slots. The number of cards that may be installed in a frame may be fewer than the specified maximum due to physical, thermal, or power limitations. The frames feature a modular back panel, allowing custom I/O rear modules to be installed for each openGear card.

The Frame Network Controller is installed in slot 0. It provides an external Ethernet connection for the frame, and is the master of the internal CANbus. The openGear cards within the frame communicate to the Frame controller via the CANbus. The Frame Network Controller incorporates an application called TransCAN, which acts as a bridge between CANbus-connected openGear cards and the external network.

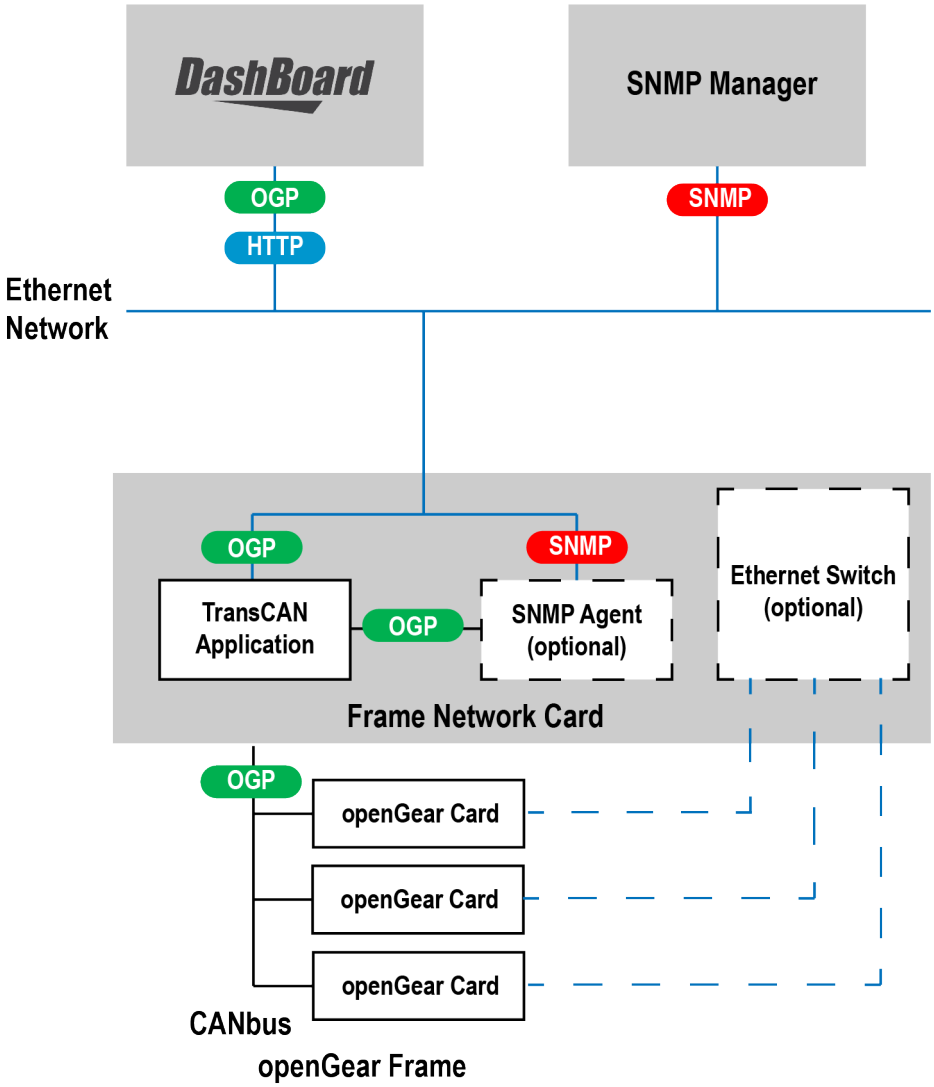


Figure 1.1 *- openGear Frame*

Each OGP client connects to TransCAN with a persistent TCP connection, and then listens to incoming messages to determine what devices are currently online. The openGear cards broadcast status information periodically, and TransCAN forwards all OGP messages that it receives over the CANbus to every connected OGP client.

The OG3-FR frame may optionally incorporate an Ethernet Switch on the Frame Network Controller, allowing dedicated Ethernet ports on each card to be routed to the single external port on the frame.

SNMP support and an integrated Ethernet switch are optional features, dependent upon the model of frame and Frame Controller. On supported Frame Controllers, SNMP is an optional software-enabled feature.

Frame	Controller	SNMP	Ethernet Switch
DFR-8310 DFR-8320 DFR-8321	MFC-8310	No	No
	MFC-8310-N	Optional	No
	MFC-8310-NS	Yes	No
	MFC-8320-S	No	No
	MFC-8320-N	Optional	No
	MFC-8320-NS	Yes	No
OG3-FR	MFC-8322-S	No	No
	MFC-8322-N	Optional	Yes
	MFC-8322-NS	Yes	Yes
	MFC-OG3-N	Optional	Yes
	MFC-OG3-NS	Yes	Yes

For additional details about the openGear frame, see ***openGear Development Guide, Part I – Hardware Specifications (8200DR-005-xx)***.

openGear Cards

openGear cards are housed within an openGear frame. All cards have access to the internal CANbus in the frame, which uses binary openGear protocol (OGP) for communication. Cards may also, optionally, provide a dedicated Ethernet port for additional communication. Cards may, through this port, connect directly to DashBoard using DashBoard Connect.

Any card may include a physical Ethernet port on the card's rear module. The OG3-FR optionally includes an Ethernet switch on the Frame Network Controller, allowing Ethernet traffic from compatible openGear cards in the frame to be routed to the single Ethernet jack on the rear of the OG3-FR frame.

For further details about the openGear frame, see ***openGear Development Guide, Part I – Hardware Specifications (8200DR-005-xx)***.

SNMP Manager

An optional feature of openGear frames is an integrated SNMP Agent. Third-party SNMP management software provides an industry-standard monitoring solution for openGear frames and cards. In addition to the feature-rich DashBoard interface, openGear frames may communicate via SNMP to an SNMP manager. OGP provides methods for openGear cards to interface with the SNMP agent. DashBoard Connect does not provide an interface for SNMP for DashBoard Connect devices.

Connectivity & Protocols

The following figure outlines the basic connectivity.

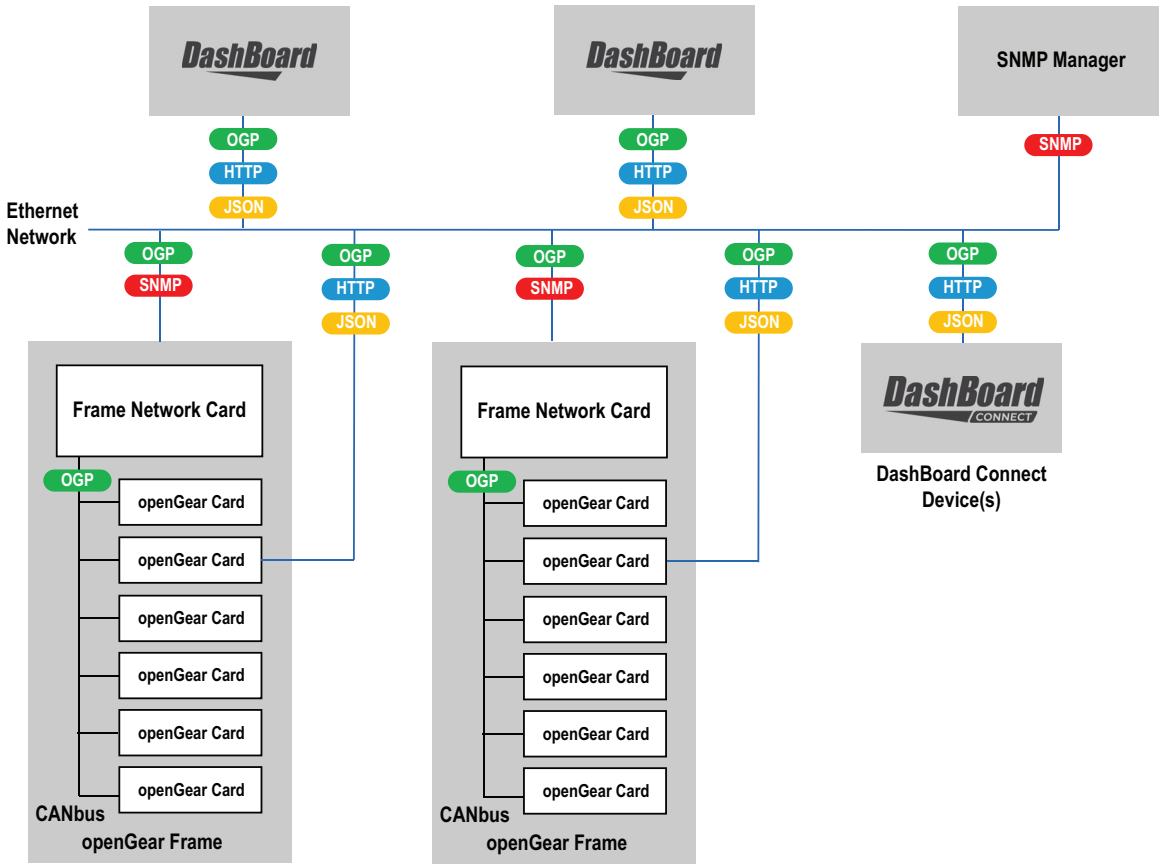


***Figure 1.2** - openGear Connectivity Architecture*

The Dashboard Connect environment utilizes a number of protocols to communicate between devices. The primary protocols are:

- Binary openGear Protocol (OGP) — See "openGear Protocol (OGP)" on page 1–5.
- openGear JSON Protocol — See "openGear JSON Protocol (OGP-JSON)" on page 1–5.
- Hypertext Transport Protocol (HTTP and HTTPS) — See "Hypertext Transport Protocol (HTTP and HTTPS)" on page 1–6.
- Service Location Protocol (SLP) — See "Service Location Protocol (SLP)" on page 1–6.
- Simple Network Management Protocol (SNMP) — See "Simple Network Management Protocol (SNMP)" on page 1–6.

openGear Protocol (OGP)

OGP is the heart of the openGear ecosystem. It is a simple message-passing protocol which enables devices to publish and allow manipulation of device parameters. It also provides for basic menu-building within DashBoard. It is a binary protocol between devices and DashBoard. DashBoard communicates to DashBoard connect devices (including openGear frames) using OGP over TCP/IP. The openGear Frame Network Controller acts as a bridge between Ethernet and its internal CANbus. openGear cards with dedicated Ethernet ports may optionally connect to DashBoard directly via TCP/IP.

openGear JSON Protocol (OGP-JSON)

DashBoard Connect devices may communicate with DashBoard utilizing a JSON (JavaScript Object Notation) representation of openGear Protocol. This allows devices to communicate using human-readable messaging which can leverage standard software libraries for implementation.

See also "JSON Reference" on page 4–1.

Hypertext Transport Protocol (HTTP and HTTPS)

DashBoard Connect utilizes Hypertext Transport Protocol as a general-purpose transport protocol to deliver documents and files between devices. Content may include simple web pages, configuration files, software updates, openGear Layout Markup Language (OGLML) documents, and images. Both unsecured (HTTP) and secure (HTTPS) connections are supported.

Service Location Protocol (SLP)

SLP is an Internet standard defined in RFC 2608 and 3224. DashBoard utilizes SLP as an automatic discovery method for DashBoard Connect devices on TCP/IP networks. SLP support is not mandatory for DashBoard Connect devices, but is highly recommended. DashBoard also provides a manual discovery mechanism.

See also "Discovery" on page 2–11.

Simple Network Management Protocol (SNMP)

openGear frames optionally support SNMP monitoring of the openGear frame and cards through standard SNMP management software. DashBoard Connect does not provide an interface for SNMP for DashBoard Connect devices.

See also "SNMP" on page 9–1.

Getting Started

The Basic Steps

Devices must go through a number of steps in order to connect and sustain a presence in Dashboard.

- **Device Discovery** — See "Device Discovery" on page 1–6.
- **Connection** — See "Connection" on page 1–6.
- **Device Parameter Query** — See "Device Parameter Query" on page 1–7.
- **Message Handling** — See "Message Handling" on page 1–7.
- **Heartbeat** — See "Heartbeat" on page 1–7.

Device Discovery

Before a device connects to DashBoard, it must first be discovered. Devices on a local network may be discovered automatically through SLP. A user may also manually discover a device by manually entering an IP address or hostname in DashBoard. In this situation the device delivers a connection properties XML file via a web server. Either via an SLP response or through the XML file, the device publishes the type of interfaces and connection methods which can be used to connect to the device.

openGear cards within an openGear frame do not need to go through an explicit discovery phase; they need only start sending parameter updates to the Frame Network Controller. The openGear card will then automatically be added as child to the frame's Tree node in DashBoard.

See also "Discovery" on page 2–11.

Connection

Using the data retrieved in the discovery process, DashBoard can connect to the device. If DashBoard is set to automatically connect, it will attempt to connect to all discovered devices periodically. Initially, it will attempt to connect every few seconds, but will back off to about once every 30 seconds if connection attempts fail. Alternatively, a user may manually connect to a discovered device via menu selection drop-down or double-clicking the node in DashBoard.

See also "Initiating Connection to OGP Devices" on page 5–1.

Device Parameter Query

After initial connection, DashBoard sends a handshake to the device. When DashBoard receives a message from the device it requests a list of all parameters.Once DashBoard has obtained all of the necessary information from a device, it relies on the device to keep it up to date by broadcasting any changes. Similarly, DashBoard may send parameter changes to the device in response to user interaction in DashBoard.

Message Handling

DashBoard listens to incoming messages and synchronizes its internal model of the device appropriately. Clients can also invoke changes by sending messages to the device (e.g. to change the value of a parameter). All such changes are communicated through a request-response transaction. Binary OGP protocol requires strict adherence to the request-response model, thus an explicit response is required to every message from DashBoard. JSON messaging uses an asynchronous model, where the device need only send parameter update messages to DashBoard when they change. Note that both messaging protocols require the device to provide a heartbeat message at regular intervals in order to keep the connection to DashBoard alive.

Response messages must be forwarded to all connected DashBoard clients, so that all clients are informed of changes invoked by other clients. If devices receive parameter changes through other methods (for example front panel control or other control protocol), these changes must be broadcast via an asynchronous parameter reporting message to all connected DashBoard clients.

***Note**: openGear cards within an openGear frame need only respond to requests from the Frame Network Controller. The Frame Network Controller manages connection and propagation of parameter updates to all connected DashBoard clients.*

OGP messages are designed to be short, simple, and stateless. Each request message is fully self-contained to ensure that the request will always have the same effect. This simplifies the design of the communications on each openGear device. The device can receive, process, and respond to each message as it is received without having to consider reception order or related factors.

However, there are situations where the UI state may depend upon parameter updates, thus it is the responsibility of the device to sequence the transmission of such messages to ensure the UI responds in a consistent and predictable manner.

Heartbeat

DashBoard clients have a 10-second timeout on the communication channel with each device. If no messages are received within the timeout window, DashBoard closes the connection to the device. Therefore, it is recommended that each openGear card should send at least one message every 5 seconds as a keep-alive or heartbeat.

Connection and Transport

In This Chapter

This chapter describes mechanisms that can be used connect to devices and present them in DashBoard, and how to customize the appearance of devices in DashBoard.

The following topics are discussed:

- **Transport Layer** — See "Transport Layer" on page 2–1.
- **Device Discovery and Integration** — See "Device Discovery and Integration" on page 2–6.
- **Device Connection** — See "Device Connection" on page 2–20.
- **Methods to Speed up Device Initialization** — See "Methods to Speed up Device Initialization" on page 2–21

Transport Layer

This section contains the following topics:

Transport Mechanisms

OGP may be transported via the following methods:

- **Binary OGP over TCP/IP** — See "Binary OGP via TCP/IP" on page 2–1.
- **JSON Netstrings over TCP/IP** — "JSON Over TCP/IP" on page 2–2
- **Binary OGP over CANbus** — "OGP via CAN" on page 2–2

TCP/IP transports sit atop a standard TCP/IP socket connection stack. OGP over CANbus sits atop a CAN 2.0B stack running at 1.0Mbit/s.

The payload of Binary OGP over TCP/IP and CANbus is the same; however each transport provides a unique wrapper.

Binary OGP via TCP/IP

Transport of OGP between DashBoard and device is via TCP/IP. The Network model is as follows:

Layer	Description
Application	openGear Protocol
Presentation	OGP TCP/IP Wrapper
Session	Socket Interface
Transport	TCP
Network	IP
Data Link	Ethernet
Physical	

OGP Transport over TCP/IP follows typical client/server architecture. The client opens a TCP connection to the device. The socket connection should remain open as long as DashBoard is up, to allow it to receive asynchronous messages from the device.

The default openGear service port is **5253**; however a device may listen on any port, provided the listening port is properly disclosed during device discovery. Devices using binary OGP must report their **equipmentType** as `opengear` during device discovery.

openGear messages are encoded for transmission via TCP/IP in a simple wrapper. Multi-byte values are transmitted in standard network order (i.e. most-significant byte first):

Field	Offset	Length	Format	Description
sync	0	4	uint32	Start of message sync = 0xBAD2ACE5
source	4	1	uint8	Source address
dest	5	1	uint8	Destination address
msgType	6	1	uint8	Message type
length	7	2	uint16	Content length (maximum 8192)
content	9	length	char[]	Message content (maximum 8192 bytes)

Important: *This encoding is subject to change without notice. Future versions may include additional fields to encode user permissions, or may be encrypted via SSL for improved security.*

JSON Over TCP/IP

OGP may alternatively be transported between DashBoard and a TCP/IP device via JavaScript Object Notation (JSON) format. The Network model is as follows:

Layer	Description
Application	openGear Protocol
Presentation	JSON Netstrings
Session	Socket Interface
Transport	TCP
Network	IP
Data Link	Ethernet
Physical	

JSON Transport over TCP/IP follows typical client/server architecture. The client opens a TCP connection to the device. The socket connection should remain open as long as DashBoard is up, to allow it to receive asynchronous messages from the device.

The default openGear JSON service port is **5254**; however a device may listen on any port, provided the listening port is properly disclosed during device discovery. Devices using JSON messaging must report their **equipmentType** as `opengear-json` during device discovery.

OGP via CAN

Overview

Within the openGear frame, OGP is carried from the Frame Network Controller to each slot via the CANbus on the midplane. The CAN bus is a two-wire 1-Mbps serial bus shared by up to 22 separate cards:

- openGear cards in slots 1 through 20.
- Frame Network Controller (MFC-83xx or MFC-OG3). The Frame Network Controller provides a common access point through which users can monitor and control all cards via Ethernet. The Frame Network Controller also hosts the optional SNMP agent, which allows monitoring and control via SNMP.

- Frame Control card. This is responsible for monitoring and controlling the frame hardware (power supplies, fans, and LEDs). On the MFC-8310-N card, it is implemented using a separate processor (and separate CAN connection). On subsequent controllers, the Frame Network Controller incorporates this functionality.

Communications within the frame follow a simplified OSI model with four layers:

Layer	Description
Application	openGear Protocol
Transport	openGear CAN transport
Data Link	Extended CAN 2.0B
Physical	CANbus (ISO 11898)

The physical characteristics of the CANbus are specified by ISO 11898. Each card may access the bus through pins 11 and 13 of the MEC8 midplane connector as defined in the ***openGear Development Guide Part I – Hardware Specifications (8200DR-005-xx)***. Each card may determine its slot number (for addressing purposes) by reading the value of the AN_SlotID resistor on pin 6 of the MEC8 connector as outlined in the ***openGear Development Guide Part I – Hardware Specifications (8200DR-005-xx)***.

Low level messaging on the bus (the data link layer) conforms to the extended CAN protocol defined by CAN Standard 2.0B. Messages are broadcast as frames, each comprising a 29-bit identification header and up to 8 bytes of payload data. The CAN protocol includes mechanisms for conflict resolution, frame validation, and error-handling.

openGear CAN Transport

The CAN protocol provides reliable broadcast of short (8-byte) frames, but does not provide a mechanism for sending messages to specific recipients or for reliably sending longer messages. The openGear CAN transport layer extends the CAN 2.0B protocol to provide message transport services similar to UDP, capable of reliably sending addressed messages up to 260 bytes long.

As outlined above, each openGear message consists of four fields:

- source address
- destination address
- message type
- message content (an array of bytes and its length)

Each card is assigned a specific address; see "Addressing Conventions" on page 5–2 for details. Source and destination addresses are included in each CAN frame to allow messages to be sent to a specific device. Addresses and message type are encoded in the CAN frame identification bits as outlined in the next section. The message content is encoded in the 8-byte CAN frame data field. Messages longer than 8 bytes are fragmented into multiple frames for transmission, and reassembled by the receiver. Content encoding is described in detail below.

***Note**: openGear CAN Transport supports card-to-controller addressing and controller-to-card addressing. Card-to-card addressing is not supported.*

Definition of CAN Frame ID

All messages carried on the openGear CAN bus shall conform to Extended CAN Standard 2.0B (having a 29-bit frame ID header). The CAN standard allows the ID header to be defined as appropriate for the application domain. For openGear, the ID header bits are defined as follows:

High Word:

Priority		Version		Source Address						Destination Address					
P1	P0	V1	V0	S5	S4	S3	S2	S1	S0	D5	D4	D3	D2	D1	D0

Low Word:

Message Type								Flags					N/A		
T7	T6	T5	T4	T3	T2	T1	T0	F4	F3	F2	F1	F0	xx	xx	xx

The functions of the fields are as follows:

Field	Length	Description
Priority	2 bits	Message Priority. Currently not implemented. • 00 – System Message • 01 – High priority application message • 10 – Low priority application message • 11 – Background priority
Version	2 bits	Protocol version. Currently 0.
Source	6 bits	Address of the sender
Destination	6 bits	Address of the recipient
Message Type	8 bits	Message type specified by the application level
Flags	5 bits	Control Flags: • F4 – Reserved • F3 – Reserved • F2 – Reserved • F1 – First frame in message • F0 – Last frame in message

Content Encoding

Each openGear message may contain up to 260 bytes of content. The message content is encoded in the CAN frame data field. The maximum CAN frame payload is 8 bytes, so messages longer than 8 bytes must be fragmented into 8-byte chunks for transmission, then reassembled by the receiver. The content length and 16-bit CRC are sent along with the content to allow the receiver to verify that the message has been received and reassembled correctly.

The first frame (fragment) of a message is formatted as follows:

Field	Length	Description
Header	29 bits	Source / Destination / MsgType / Flags (see section above). F1 Flag set.
length	16 bits	Message content length (maximum 260).
CRC	16 bits	CRC for the message content, computed using the algorithm in “Appendix E: CRC Computation” on page 11–12.
content	4 bytes	First 4 bytes of message content.

Subsequent frames (fragments) are formatted as follows:

Field	Length	Description
Header	29 bits	Source / Destination / MsgType / Flags (see section above).
content	8 bytes	Next 8 bytes of message content.

Subsequent frames must each contain 8 bytes of content in the content field, except the final frame of the message, which may contain fewer than 8 bytes.

The source address, destination address, and message type must be the same for each fragment.

Fragments must be sent in order. The first and last frame of each message must be flagged using bits F1 and F0 in the frame identifier header. Intermediate frames must have F1 = F0 = 0.

The receiver must compute the CRC for the reassembled content and verify it against the CRC sent in the first frame. If the CRC does not match, the message is corrupt and must be discarded.

Single-frame messages: any message having a content length of 8 bytes or less must be sent as a single CAN frame with flags F1 and F0 both set, without the added message length and CRC. Because there is no fragmentation and reassembly, the CRC is not required in this case; correct delivery of the single frame is ensured by the CAN controller hardware.

Fragmentation Example

This example shows the CAN frames required to send the following multi-frame message:

- source address: 0x15 (slot 5)
- destination address:0x02
- message type: 0x37
- message content: “This fragmented message” (24 bytes including terminating null)

Four frames are required to send the message (24 + 2 + 2 = 28 bytes total).

The data field of the first frame includes the content length (0x0018) and CRC (0x644D):

Frame	Frame ID Header	Length	Data							
1	05 42 37 10	8	00	18	64	4D	54 T	68 h	69 i	73 s
2	05 42 37 00	8	20 _	66 f	72 r	61 a	67 g	6D m	65 e	6E n
3	05 42 37 00	8	74 t	65 e	64 d	20 _	6D m	65 e	73 s	73 s
4	05 42 37 08	4	61 a	67 g	65 e	00 \0	n/a	n/a	n/a	n/a

Hardware / Data Link Requirements

To avoid interfering with other communications on the CAN bus, each openGear card must satisfy the following requirements:

1. Each card must respect the differential CAN signal levels specified in ISO 11898. This is readily achieved by connecting to the bus through a compliant CAN transceiver (such as TI SN65HVD233).

2. All transmission on the bus must conform to CAN standard 2.0B (extended ID headers). This is readily achieved by using a compliant CAN controller (such as Philips SJA1000).
3. All transmission on the bus must be at 1 Mbit/s.

Device Addressing

Each device connected to DashBoard consists of a device address and a slot ID. The device address is a network name or address determined during device discovery; DashBoard connects via TCP/IP to the device address. Additionally, DashBoard uses a slot ID to access multiple sub-devices within a device address. Every node in DashBoard must specify a slot ID. DashBoard Connect devices usually need only present a single node with Slot ID 0. openGear frames present the Frame Network Controller with Slot ID 0, and each openGear card installed in the frame takes an address between 1 and 20 (corresponding to the physical slot the in which the card is installed).

Device Discovery and Integration

This chapter contains the following topics:

Getting Started

In order to present a device in DashBoard, there are several steps that must be followed. Due to the flexible nature of DashBoard, the tasks at each step will vary depending upon the desired implementation strategy. However, the basic steps are the same regardless of implementation:

1. Decide upon how the device will be presented in DashBoard.

   Implementation can include web pages, openGear Protocol controls, or advanced layouts with custom scripting. A device may include a single interface or multiple interfaces, presented in a hierarchical tree.

   For more information, see "Device Presentation" on page 2–7.

2. Choose the connection interface.

   Devices may connect via CANbus in an openGear Frame, or directly to DashBoard via TCP/IP. DashBoard Connect devices may connect via TCP/IP using either binary OGP or JSON messaging mechanism.

   For more information, see "Connection Interfaces" on page 2–10.

3. Implement device discovery.

   TCP/IP-connected devices may support automatic discovery, manual discovery or both. In all cases, the device must publish information to DashBoard in order to be presented in the DashBoard tree.

   For more information, see "Discovery" on page 2–11.

4. Create the device definition file.

   In the final, optional, part of the discovery process, DashBoard will read a description of the device properties, its connection methods and presentation layout in the DashBoard Tree View. This is typically only required for complex devices with multiple tree nodes which are beyond the scope of what may be presented in device discovery phase. The properties are typically defined in a DashBoard Connect Device Properties XML file.

   For more information, see "Device Definition" on page 2–15.

5. Implement the configuration interfaces.

The implementation will vary greatly depending upon the device. Implementation may involve an openGear Protocol (OGP) server, OGLML (openGear Layout Markup Language) server and/or a web server.

Further Reading

DashBoard features a rich alarms, status and monitoring interface enabled by openGear Protocol (OGP), described later in this document.

Advanced control interfaces may be implemented by supplementing OGP with openGear Layout Markup Language (OGLML) and ogScript. ogScript documentation is available in the DashBoard online help.

Device Presentation

DashBoard presents devices to the user in a tree view. Each device creates one or more nodes in the tree to provide an interface for control and monitoring. Each node may launch an editor where information can be reported, and the user may interact with the device. Each node may also have child nodes, allowing multiple levels of hierarchy in the tree.

This section contains the following topics:

The DashBoard Device Tree

The tree view allows devices to be shown in the list, and also allows the creation of child objects, so that a hierarchical tree can be created. For example, a Video Server may present "Channel 1" and "Channel 2" as children, each child presenting its own editor.

Figure 2.1 shows the DashBoard tree view.

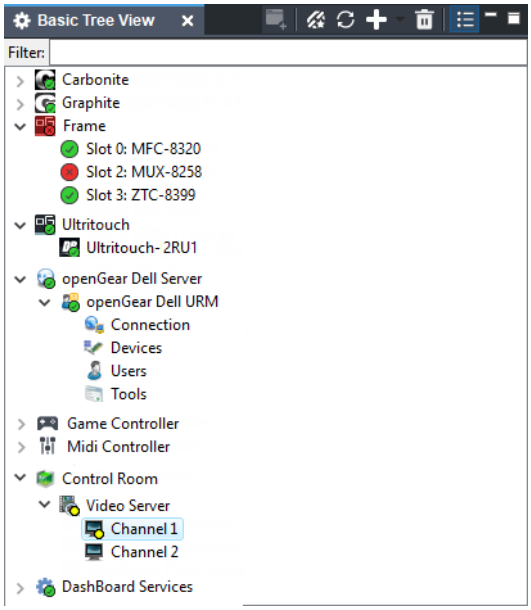


***Figure 2.1** - The DashBoard Tree View*

Each node in the tree may represent one of the following:

- Device editor page
- Parent of child nodes.

When a user opens a node in the tree, the defined editor appears. If the user opens a parent node, the tree expands to display any child nodes.

Devices may present an interface using several different methods. It is also possible to create several configuration pages, each of which may use a different interface type. The interface types are:

- openGear Protocol (OGP) default interface
- openGear Layout Markup Language (OGLML) view
- Standard Web page (web-generic) view

openGear Protocol – Default Layout

Binary openGear Protocol and openGear JSON Protocol provide access to DashBoard's detailed alarms, instant system-wide updates of current parameter information, and a pre-configured uniform user interface.

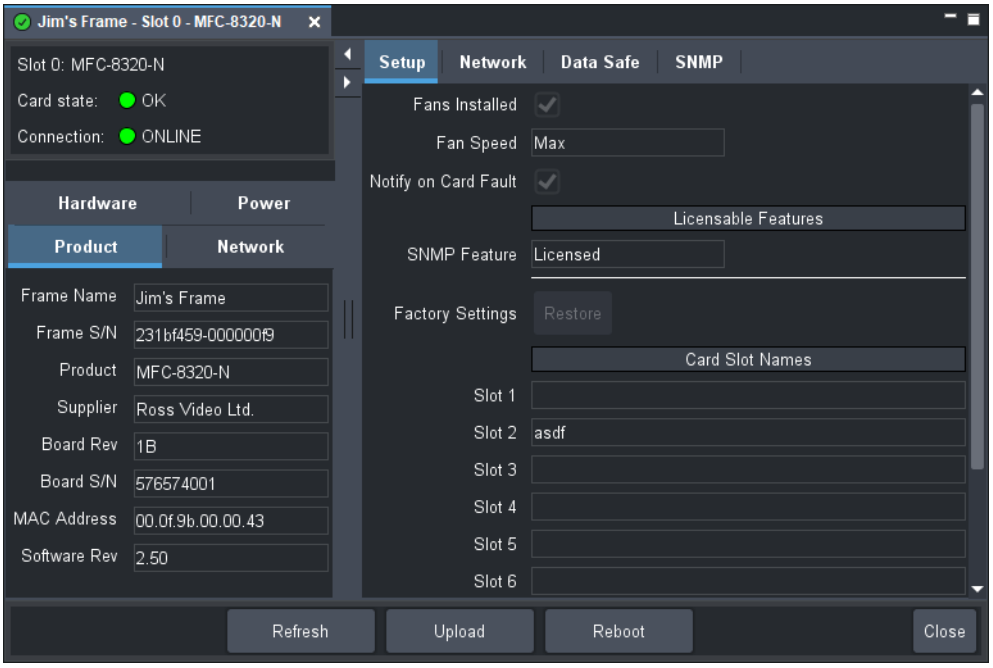


***Figure 2.2** - Example of a Pre-Configured Editor*

DashBoard uses the protocol to send updates to parameter values to the device and the device sends updated values to all connected DashBoard instances. Devices can also provide an arbitrary number of alarm parameters and update them to change the reported status of the device in DashBoard.

openGear Layout Markup Language (OGLML)

openGear Layout Markup Language extends the capability of OGP-connected devices, allowing them more freedom to define custom interface layouts within DashBoard.

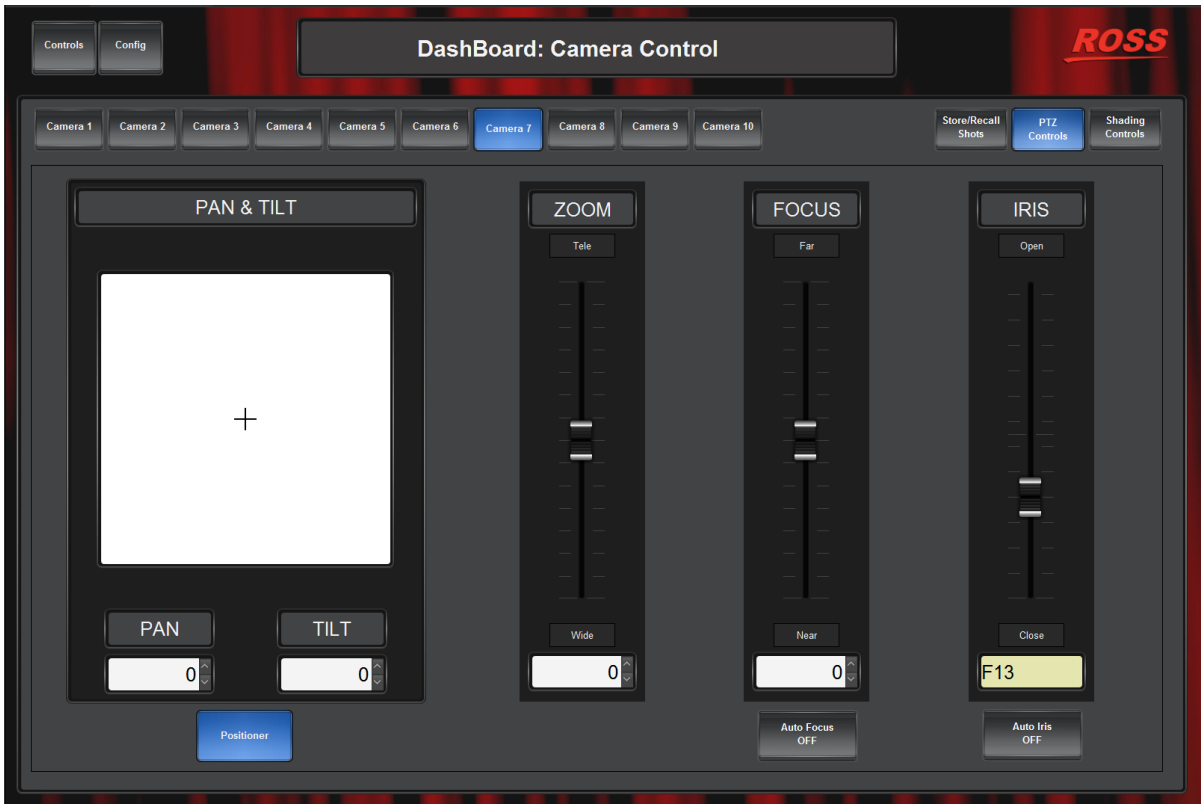


Figure 2.3 *- Example of an OGLML-defined Interface*

OGLML-based layouts still use OGP as the underlying protocol to communicate between the device and DashBoard, but the layout of the interface is defined by the OGLML document, rather than the default OGP layout. OGLML is an XML-based markup language which defines the exact placement and style to be used for OGP items to be arranged in any manner within the device editor. Additionally, OGLML interfaces allow the inclusion of ogScript, a powerful scripting language, allowing more advanced control interface behaviors.

OGLML documents can be interactively created within DashBoard using the Panel Builder feature.

Web Page Interface (web-generic)

DashBoard Connect allows web-enabled devices to appear in the DashBoard Tree View. If your product has an existing web-interface and does not require Java Applets, the product can be properly added to DashBoard with addition of an XML file. These devices are referred to as `web-generic` devices in DashBoard. Note that web-generic devices are considered dumb nodes, and cannot take advantage of DashBoard's global parameter management, alarms and other built-in features. Furthermore, web-generic elements of a device's UI cannot be used in DashBoard Custom Panels.

This mechanism uses a DashBoard plug-in to read a DashBoard Connect XML and use it to add nodes to the DashBoard tree view. Each node in the tree can point to a web page. When the user opens the node, a web browser is opened in the DashBoard control area and the selected web page is loaded.

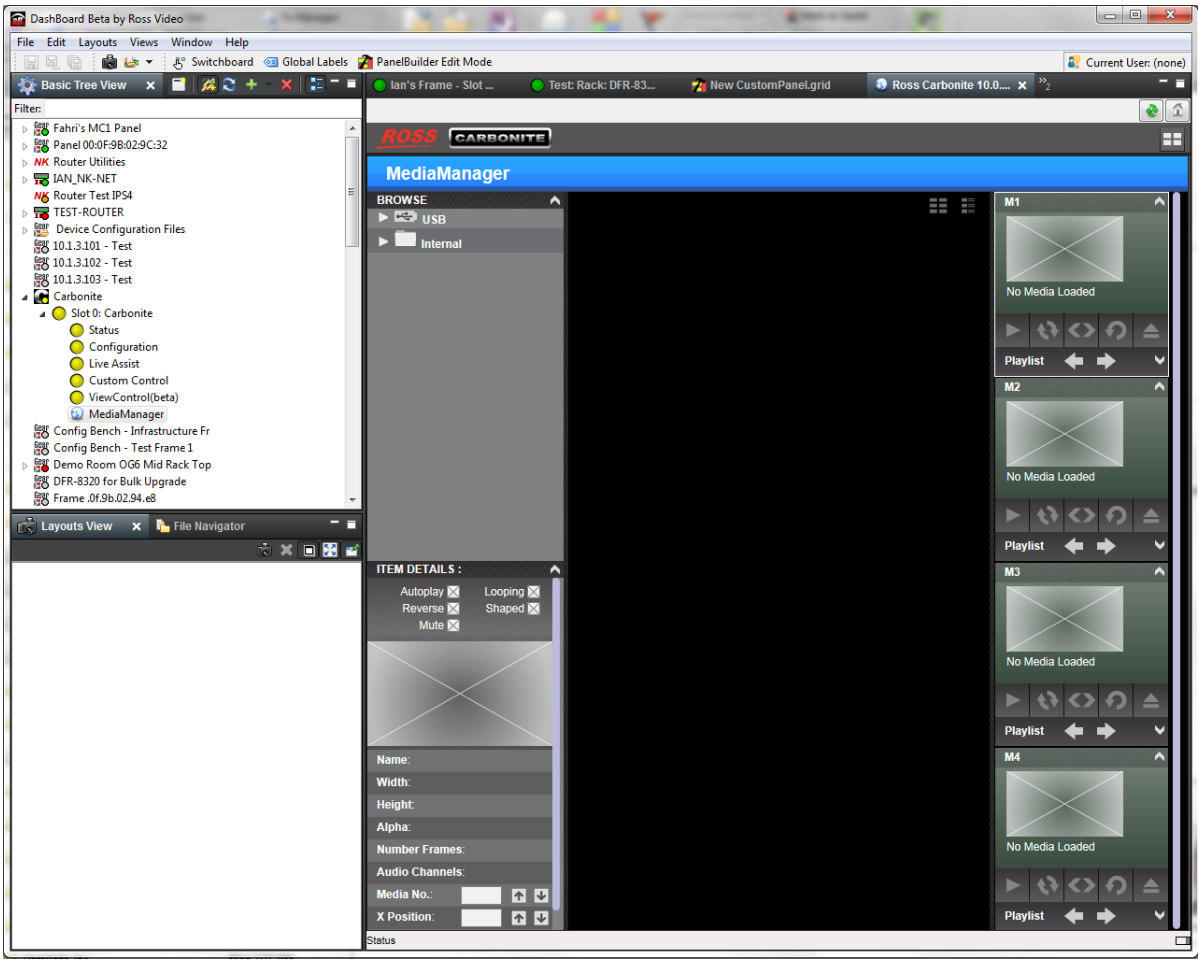


***Figure 2.4** -Example of a Web-generic Tree Node*

DashBoard has no control over the browser version and the availability of third-party browser plugins for features such as Flash, ActiveX, and PDF. Note that Java plug-ins are not supported in web-generic pages.

Connection Interfaces

There are two physical interfaces which may be used by a device to connect to DashBoard. These are TCP/IP and CANbus.

TCP/IP

For most devices, communication with DashBoard is performed via TCP/IP using either OGP or JSON messaging. In order to initiate this connection, the device must go through device discovery in order to appear in the DashBoard Tree.

CANbus

Cards installed in an openGear frame communicate with DashBoard with OGP via CANbus. In this scenario, the card need only begin sending messages to the frame controller via CANbus, and the Frame Network Controller will add the card as a child node in the DashBoard device tree based upon the card's slot in the frame. DashBoard then begins to query the device's parameters. No explicit discovery mechanism is required. The Frame controller handles all of the discovery and tree management with DashBoard on behalf of the card.

If the user opens the card's node in the tree, the pre-configured interface is displayed. The card may additionally specify OGLML or web-generic interfaces through implementation of an independent TCP/IP web server (if the card has an independent Ethernet port) or through the use of External Objects.

Note that if the card is removed or stops responding, the card remains in the tree, and is shown in a disconnected state; to remove the node from the tree, the user must explicitly remove it.

Discovery

In order for a TCP/IP-connected device to appear in the DashBoard Tree View, it must first be discovered. There are two discovery mechanisms supported:

- Automatic discovery via SLP
- Manual discovery using an XML file delivered via a web server.

Devices must support at least the manual discovery process, but it is recommended that both methods be implemented.

***Note**: Cards in an openGear frame do not need to implement a discovery mechanism, as the frame controller manages this on the card's behalf.*

During discovery, device properties are defined in two steps: Connection Properties and Device Properties. Both of these phases use a common language for property definition.

Connection Properties

Connection properties are specified upon initial discovery of the device to provide basic connection information about the device. This phase of discovery provides information about how DashBoard connects and communicates with the device. These properties are specified in an SLP response or a `connection-props.xml` file. The `connection-props.xml` file should provide the same information as an SLP discovery response.

Device Index URL

The DashBoard Connect Device Properties XML file allows for more complex presentation of the device in DashBoard. It allows complex device hierarchies to be specified, and nodes of mixed types. This XML file may be served up by a web server on the device, or specified in OGP with reserved OID 0xFF0D. This step is not mandatory, and generally is not required for devices that present a simple single node in DashBoard.

***Note**: A property can be declared in both the index-url and connection-props files but ONLY if their values match.*

Automatic Discovery via SLP

DashBoard uses Service Location Protocol (SLP) to search the network for devices. SLP broadcasts a specific query over a network and waits for responses. The nature of the broadcast message means that automatic discovery will only work within a subnet, and can often be blocked by forwarding routers. For more information about SLP, visit the following page on the Internet Engineering Task Force (IETF) website: http://www.ietf.org/rfc/rfc2608.txt.

A device must respond to an SLP query with the following information:

- serviceURL
- equipmentType
- port
- node-id
- node-name
- additional properties (optional)

  Valid properties are outlined in "Device Definition" on page 2–15.

The SLP response takes the following form:

```
service:broadcast-equipment:equipmentType://address:port
(property_name=property_value),(property2_name=property2_value),…
```

The `serviceURL` must include the service type broadcast-equipment and the address (ip address or hostname) of the device and port to connect. The `serviceURL` should also specify the `equipmentType` property appended to the service type. For example, a device with `equipmentType opengear` could respond with a `serviceURL` of:

```
service:broadcast-equipment:opengear://hostname:5253
```

The value of the `equipmentType` property determines how DashBoard connects to the device. DashBoard supports the following `equipmentType` values by default, but other `equipmentType` may be added through custom plug-ins:

equipmentType	Function
opengear	DashBoard connects to the device using binary OGP messaging.
opengear-json	DashBoard connects to the devices using JSON messaging.
opengear-oglml	DashBoard connects using OGP, and will render an enhanced interface defined by an OGLML document.
web-generic	DashBoard connects to the device as a generic webserver using HTTP.
structure	Used to create a generic node to contain other nodes and add to the overall hierarchy. These can be used to separate/contain other nodes in the tree.

In general, the `equipmentType` should only be set to `web-generic` for simple non-OGP-capable devices that only provide a simple web page interface. Other devices should report `equipmentType` as `opengear` or `opengear-json`, even if the device contains a mix of web-generic and OGP nodes.

The node-id must be a unique identifier within DashBoard. It is recommended that the format *Product Name-Serial Number* be used to ensure a unique node-id. The node-name is an identifier that will be displayed in the DashBoard Tree View, and need not be unique. The node-id and node-name must be passed as an attribute in the SLP response. For example:

```
service:broadcast-equipment:opengear://hostname:5253
(node-id=ProductName-123456789),(node-name=My OGP Device)
```

The above response would report to DashBoard a device with hostname `hostname` is equipmentType `opengear`. DashBoard should communicate with the device on port 5253 (the default port for OGP). Its node-id is `ProductName-123456789` and its display name is `My OGP Device`.

The device may optionally also specify additional properties in the SLP response.

The index-url property tells DashBoard to fetch a DashBoard Connect Device Properties file via an HTTP request to further describe the device. For example:

```
service:broadcast-equipment:opengear://hostname:5253
(node-id=DeviceName-123456789),(node-name=My OGP Device),
(index-url=http://hostname:80/ogp/ogp.xml)
```

In this example, DashBoard will connect to a web server on port 80 of the device and read the DashBoard Connect Device Properties XML file `ogp/ogp.xml` to complete the definition of the device.

For web-generic devices, the config-url property tells DashBoard the URL to retrieve when the node is opened. A simple web server bookmark can easily be discovered via SLP. For example:

```
service:broadcast-equipment:web-generic://192.168.0.29:80
(node-id=ProductName-12345678),(node-name=MyTestNode),
(config-url=http://192.168.0.29/default.html)
```

The equipmentType is set to `web-generic` to indicate that DashBoard should connect to this device using the web browser plug-in. So, in this example, DashBoard would create a node in the Tree View which, when opened, will retrieve the page `default.html` from the web server at 192.168.0.29 (port 80) and launch it in a browser window within DashBoard.

Valid properties are outlined in *"Device Definition" on page 2–15*.

Additional Notes:

- SLP uses UDP broadcast messages and only works within a network subnet
- SLP Servers must listen for broadcast messages on port 427
- DashBoard queries for services of type `broadcast-equipment`
- Services in DashBoard must also provide node-id and node-name
- Many Linux distributions come with an SLP Server from OpenSLP called with a name of "slpd" and SLP registrations can easily be managed via the "slptool" command. OpenSLP is also available for Microsoft Windows and Mac OS X. Visit http://www.openslp.org.

***Note**: Because automatic discovery can fail for a variety of reasons (different subnet, firewalls, etc.) products in DashBoard must not rely on automatic discovery alone. It is strongly recommended that devices also implement manual discovery.*

Manual Discovery

In the event that automatic discovery fails (or the user has disabled it), all products in DashBoard should also provide a form of manual discovery where users type in a known hostname or IP address to identify the device. DashBoard will then attempt to retrieve an XML file hosted in a known location on a web server. The XML file must contain the same information required in the SLP response of automatically-discovered devices.

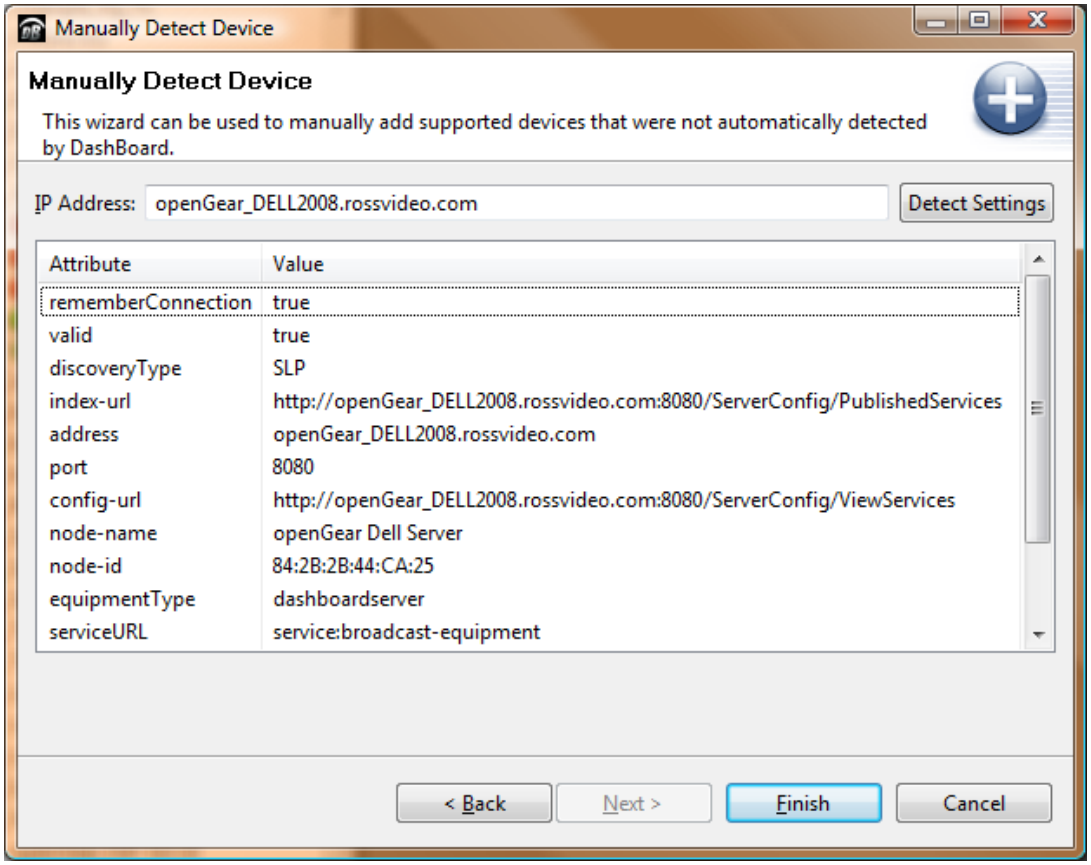


***Figure 2.5** - Manual Detect Device Dialog Box*

To manually search for a device

1. Click the green "+" icon in the DashBoard tree view.
2. Choose the **Manual Connection** option.
3. Enter the host name or IP address of the device.
4. Click **Detect Settings**.

5. When you click the **Finish** button, the connection information is sent to the DashBoard tree view just as if it were automatically detected via SLP.

When **Detect Settings** is clicked, DashBoard reads a connection properties file, `connection-props.xml`, from the device. The device must provide a web server running on either port 80 or port 8080. The file containing the connection settings must be located at:

```
http://[host or ip]/connect/connection-props.xml
```

The connection-props.xml File

At a minimum, the connection-props.xml file must specify the following properties:

- serviceUrl
- equipmentType
- node-id
- node-name
- address
- port
- additional properties (optional)

The serviceUrl, equipmentType, address, and port properties would normally be part of the SLP service URL and must be explicitly stated in the `connection-props.xml` file. The remaining properties are duplicates of the SLP attributes the product would typically send.

Commonly, a device would also specify a DashBoard Connect Device Properties file to provide more detailed description of the device using the index-url property. For example:

```
<?xml version="1.0" encoding="UTF-8" standalone=”yes”?>
<properties version="1.0">
   <comment>DashBoard Device Connection Settings</comment>
   <entry key="serviceUrl">service:broadcast-equipment</entry>
   <entry key="equipmentType">opengear</entry>
   <entry key="address">my-host-name</entry>
   <entry key="port">5253</entry>
   <entry key="node-id">MyOGPDevice-123456789</entry>
   <entry key="node-name">My OGP Device</entry>
   <entry key="index-url">http://my-web-server/ogp/index.xml</entry>
</properties>
```

DashBoard will then fetch the file `ogp/index.xml` from the device’s web server (http://my-web-server) to complete the definition of the device.

***Note**: The device may specify the index-url via reserved OID 0xFF0D instead of specifying an* `index-url` *property during device discovery.*

It is recommended that only basic connection settings are placed in the connection-props.xml file and the full device definition be placed in the Device Properties XML file. Reasons for this include:

1. The `connection-props.xml` file cannot specify any hierarchy, so no child nodes may be specified.
2. The properties in the Device Properties XML file specified by the index-url property are periodically re-read by Dashboard, as specified by the refresh-interval property. The connection-props.xml file is only read once when the initial discovery of the device is performed.

Device Definition

The definition of a device is specified through a series of properties. These properties may be supplied via SLP broadcast, XML configuration file, OGP external objects, or any combination of these methods.

See also "Discovery" on page 2–11.

The property defined in an XML entry is treated the same as a property defined in an SLP-Broadcast message. In certain cases, a property can define a value also provided by an openGear Protocol reserved OID. Whenever a property is defined in multiple places (openGear XML, SLP, or reserved OID), they must all report the same value otherwise DashBoard may not behave correctly.

Device Definition XML file Structure

The DashBoard Connect Device XML file uses a series of key-value pairs to specify properties for the device structure. Properties are specified in the following format:

```
<entry key="property-name">property-value</entry>
```

For example:

```
<entry key="equipmentType">web-generic</entry>
```

Hierarchy can be specified by using the `<children>` and `<child>` tags. It is possible to mix nodes of different `equipmentType` within the same device definition; for example a device may have a child which presents a default OGP view, another child with an OGLML view, and a third that opens a web interface.

The basic XML file structure is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<properties version="1.0">
   <entry key="parent-property1-name">property1-value</entry>
   <entry key="parent-property2-name">property2-value</entry>
   <children>
      <child>
         <entry key="child-id">child1-id</entry>
         <entry key="child1-property1-name">property1-value</entry>
         <entry key="child1-property2-name">property2-value</entry>
      </child>
      <child>
         <entry key="child-id">child2-id</entry>
         <entry key="child2-property1-name">property1-value</entry>
         <entry key="child2-property2-name">property2-value</entry>
      </child>
   </children>
</properties>
```

Example

For example, the following tree view of a Video server device has 2 child nodes. Channel 1 is a web-generic configuration view, and channel 2 is an OGLML view:

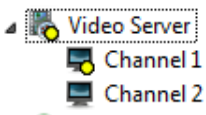


***Figure 2.6** - Tree Elements Created by XML File*

The following XML file will generate the view in **Figure 2.6**, above:

```
<?xml version="1.0" encoding="UTF-8"?>
<properties version="1.0">
   <comment>DashBoard Device Connection Settings</comment>
   <entry key="base-url">http://my-host-name/device-info</entry>
   <entry key="node-id">MyVideoServer-123456789</entry>
   <entry key="node-name">Video Server</entry>
   <entry key="node-image-url">images/video_image_16.png</entry>
   <entry key="status-level">WARN</entry>
   <entry key="status-message">No input for Channel 1</entry>
   <entry key="refresh-interval">30000</entry>
   <children>
      <child>
         <entry key="equipmentType">web-generic</entry>
         <entry key="node-name">Channel 1</entry>
         <entry key="child-id">chan-1</entry>
         <entry key="config-url">config/chan1.html</entry>
         <entry key="node-image-url">images/still_frame_16.png</entry>
         <entry key="status-level">WARN</entry>
         <entry key="status-message">No input for Channel 1</entry>
      </child>
      <child>
         <entry key="equipmentType">opengear-oglml</entry>
         <entry key="node-name">Channel 2</entry>
         <entry key="config-url">config/chan2.xml</entry>
         <entry key="child-id">chan-2</entry>
         <entry key="node-image-url">images/still_frame_16.png</entry>
         <entry key="status-level">OK</entry>
         <entry key="status-message"></entry>
      </child>
   </children>
</properties>
```

URL Types

URLs specified in property values may be specified in a number of ways, supporting different transport methods. The following formats are supported:

Transport	Description
http://	HTTP connection to a web server. Defaults to port 80.
https://	Secure HTTP connection to a web server. Defaults to port 443.
file://	Specified a file on the local file system.
eo://	Links to an External Object. The OID of the External Object in Hex format is specified. For example: `eo://0x0209`

Device Definition Property Reference

Name	OID	Function
Address		IP address or network name of the device.
base-url		This property allows all other URLs to be relative to this value.
child-id		Unique identifier for child nodes. Appends to parent's node-id. Has no impact on openGear cards.
config-url	0xFE08	URL of document to open when user opens node.
equipmentType		Determines the type of broadcast equipment that has been detected. Has no impact on openGear Frames. Has no impact on openGear cards.
hardware-id	0xFF04	Should be set to same as the node-id. Has no impact on openGear cards.
index-url	0xFF0D	URL of a DashBoard Connect Device Properties XML file.
node-id	0xFF04	Unique identifier for the device. Has no impact on openGear cards.
node-category	0xFE05	Items sharing the same category are kept together. Has no impact on openGear cards.
node-image-url	0xFF0C 0xFE06	The URL of an image for the icon in the DashBoard tree view.
node-name		Display name for this element in the tree view. Has no impact on openGear cards.
Port		Port number to connect to the device.
refresh-interval		Time, in milliseconds, after which DashBoard will recheck the definition XML file.
serviceUrl		Set to "service:broadcast-equipment" for DashBoard discovery.
status-level		Current status of this element. Has no impact on openGear Frames. Has no impact on openGear cards.
status-message		Status message for this element. Has no impact on openGear Frames. Has no impact on openGear cards.
tool-tip		Tool tip text for this element in the tree view. Has no impact on openGear Frames. Has no impact on openGear cards.
urm-required		DashBoard will pass URM information to config-url.

Property Descriptions

address

This property is the hostname or IP address of the device. Dashboard attempts to connect via the address and port specified in the address and port properties

base-url

This property allows all other URLs to be relative to this value. If defined, it must be a valid URL beginning with `http://` or `https://`, or `eo://`. The base-url property should be defined at the same level as any dependent relative URLs. A child inherits the base-url from its parent, unless one is specifically defined for the child.

child-id

For lower-level nodes in the tree view, this value defines a suffix to be appended to the parent element's node-id. The combination of parent node-id with this child-id becomes the composite node-id for this element. For example, a child-id of `Page1,` with parent with node-id `MyDevice` would have a complete node-id of `MyDevice/Page1.`

This value should only be specified for child nodes and is ignored if node-id is explicitly defined.

config-url

For `web-generic` and `opengear-oglml` equipmentTypes, this is the URL of a web page or OGLML page to open when the user drags this node to the editor area or selects the node and clicks 'open'. This must either be a valid full URL or a path relative to the value of base-url. It is strongly recommended that configuration pages avoid the use of browser plug-ins such as PDF and Flash. Java Applets are not supported at all and must not be used.

equipmentType

Tells DashBoard the type of broadcast equipment that has been detected, so that DashBoard knows how to interface with it. This attribute is required by all top-level nodes in the DashBoard tree view. The following `equipmentType` values are supported by DashBoard's default plug-ins:

equipmentType	Function
opengear	Node will be created as a default-view OGP device using OGP messaging.
opengear-json	Node will be created as a default-view OGP device using json messaging.
opengear-oglml	Node will present an interface defined by an OGLML document.
web-generic	Node will present a web page interface.
structure	Node expands to reveal child nodes. It facilitates a hierarchical node structure.

Other values of `equipmentType` are valid; however require a custom plug-in for support.

If this node is also reported via SLP, the `equipmentType` property must match the trailing part of the service URL E.g. for `service:broadcast-equipment:opengear` the equipmentType is `opengear.`

Note that this property has no impact on openGear Cards.

hardware-id

Manual openGear Frame connections use the frame IP address and port as the value for the node-id field. The hardware-id value is used for storing/retrieving the correct permissions from User Rights Management and to detect if the frame at a given IP address has changed.

In general, it is unnecessary to define this value for anything other than an openGear frame and, if it is defined for an openGear frame, it should contain the same value as the node-id.

index-url

This is the URL of a DashBoard Connect XML file. This must either be a full URL or a path relative to the value of base-url. DashBoard will read the contents of the XML file pointed to by `index_url` to continue the definition of the device. If it is used in the top level, all properties defined in both files (the one defining `index-url` and the one to which the `index-url` points) should be the same.

node-category

This is used to control grouping of similar devices in the DashBoard tree view. This value is only used by top-level elements in the DashBoard tree view. Items sharing the same category are kept together.

Each `equipmentType` defines its own default category. The openGear equipment type uses a category of `"openGear Devices"`. This value defines a top-level node for User Rights Management and is also used to group similar devices in the DashBoard tree view.

By specifying a custom node-category, it allows devices of similar type or manufacturer to be grouped together in the DashBoard tree.

node-id

The unique identifier for the device. This is required by any top-level node in the DashBoard tree view. A node-id must be globally unique within DashBoard. node-id should not be specified for child nodes. Although devices may define any node-id they wish, the recommended value for node-id is:

```
productName-serialNumber
```

Note that the node-id is assigned when the node is created in the DashBoard Tree and cannot be changed. It is recommended that this property be only defined in an SLP broadcast `or connection-props.xml` file.

***Note**: openGear frames detected via SLP have the frame serial number assigned as the node-id. openGear frames added manually using the "TCP/IP openGear Frame" option will have a node-id set to the IP address. However if the "Detect Frame Information" option is selected, the frame will populate the node-id with the frame's serial number.*

node-name

The display name for this element in the tree view.

node-image-url

The URL of an image to use as the icon for the node in the DashBoard tree view. Note that status information (green, yellow or red dot) can be overlaid on top of the icon. This attribute should be used only for web-generic and SLP devices. For openGear cards, the 0xFF0C OID must be used. For openGear Frames and DashBoard Connect Devices, OID 0xFE06 is used.

Images should be of the following format:

- PNG, GIF, JPEG
- Size: 16x16 pixels

port

This property is the TCP port number where the device will accept incoming connections. Dashboard will attempt to connect via the address and port specified in the address and port properties.

refresh-interval

Defines how frequently DashBoard should check for changes to the XML file. The value is provided in milliseconds:

- Default: 10000ms (10 seconds).
- Minimum: 5000ms (5 seconds).

serviceUrl

SLP Broadcasts are used to locate network devices. DashBoard will search for any service reporting a service URL of `service:broadcast-equipment`. serviceUrl should always be `service:broadcast-equipment`.

status-level

For `web-generic` nodes only.

Defines the current status of this element. status-level no meaning for OGP devices. OGP devices must report status level through OGP alarm parameters; the worst-case active alarm is reported in the DashBoard tree.

Valid values are as follows:

status-level	Function
OK	No status icon is overlaid.
WARN	A yellow warning icon is overlaid on the node's icon.
ERROR	A red error icon is overplayed on the node's icon.

status-message

For `web-generic` nodes only.

Defines the status message for this element. If no tool-tip is defined for the node, the status message will be used as the tool-tip. Has no meaning for OGP devices. OGP devices must report status message through OGP alarm parameters; the worst-case active alarm is reported in the DashBoard tree.

tool-tip

For `web-generic` nodes only.

Defines the tool tip text for this element in the tree view. If this is not defined but status-message is, the status-message value is used as the tool tip.

urm-required

When set to true, DashBoard passes user ID information to a device editor. DashBoard injects two parameters to http requests made to config-url:

- trusted=true
- username=`[current user ID]`

Only applies to elements that define config-url.

Device Connection

TCP/IP Connection Handshake

Once a device has been discovered, DashBoard connects to each node of the device that was created during the device discovery process. This begins with a connection handshake message. The connection handshake between DashBoard and a device allows the device to refuse connections from a DashBoard client. This allows a device to limit the number of active connections it maintains to provide a minimum quality of service to connected DashBoard clients.

***Note**: The connection handshake is only performed with devices that communicate directly using DashBoard over TCP/IP. Individual cards within an openGear frame do not need to concern themselves with this, as the frame controller handles it.*

By default, DashBoard automatically discovers devices on the network (via SLP) and establishes connection to each. In a large network, behavior may be undesirable. Individual devices may not have the capacity to handle multiple simultaneous DashBoard connections.

Although it would be quite simple for the device to simply refuse to accept new connections, this does not stop DashBoard from periodically attempting to re-connect to the device. As well, it is important to provide feedback to the user, so they can distinguish a truly offline device, from one that has simply reached its maximum connection limit.

To support this “intentional rejection of connection”, a device must accept all DashBoard connections, and then must send a specific message, to inform DashBoard that the connection is refused before closing the TCP connection. DashBoard will then stop trying to reconnect to the device.

See also:

- “JSON Reference” on page 4–1
- “OGP Reference” on page 6–1

Keeping the Connection Alive

DashBoard Connect devices are responsible for maintaining a live connection with DashBoard. DashBoard will timeout and close a connection if there is no message received after 10 seconds. Once the initial socket connection has been established, it is recommended that each the device send a message to DashBoard at least once every 5 seconds. Devices should implement a heartbeat to guarantee the connection is kept alive.

Devices may support connection to multiple DashBoard clients simultaneously. It is the responsibility of the device to broadcast any state change to all connected DashBoard clients to ensure they all stay in sync. Any client-specific session data must also be managed by the device. It is also the responsibility of the device to refuse connections via the handshake response if it does not support multiple connections.

Methods to Speed up Device Initialization

OGP-Binary developers may wish to use one of the three methods described below to handle the initialization of devices with many parameters quickly and efficiently. This involves bypassing the typical DashBoard response message to each device query that is described in the device communication model.

The three methods are listed here to help you choose which method will work best for your needs:

- OGP Blast Mode —This method is recommended when a device is running its own OGP TCP/IP server, or is in an openGear Frame with relatively low CAN Bus utilization.
- XML Side Loading of the Device Description — This method requires dynamic regeneration of an XML document from the OGP device and requires the generation of a potentially large XML document.
- Instantiate a New Connection to the Device — This method involves an extra TCP/IP connection, and the loss of DataSafe and SNMP usage capabilities provided by the openGear Frame.

  Note: This method allows the dedicated connection to use either OGP-Binary or OGP-JSON. If you choose to continue using OGP-Binary, you can pair this with OGP blast mode.

OGP Blast Mode

OGP Blast Mode allows a device to indicate to DashBoard that it is about to initiate a “blast” of messages, instead of a typical device request/response message flow. In this case, DashBoard will allow the OGP-Binary device to send a number of response messages without waiting for their requests. To use OGP Blast Mode, it is required that the OGP device preemptively sends its responses to DashBoard’s typical request query in advance, since the DashBoard client still needs this information.

You can configure blast mode by turning it on or off for your device. It must default to ‘off’.

OGP_GET_NUMPARAMS Response

Message type: 0xC5

Message length: 3*

Response required: none

Description: This is the required response to an OGP_GET_NUMPARAMS request. It provides the number of parameters supported by the device.

Field	offset	length	format	Description
rc	0	1	uint8	return code OGP_OK - normal return
numPar	1	2	uint16	number of parameters for this device
xmlDescriptionLen	3	1	uint8	length of xmlDescriptionURL (including null-terminator) to follow
xmlDescriptionURL	4	*	string	URL of a device definition XML file (if a URL is provided, the remainder of the parameter descriptor and menu queries are skipped by DashBoard)
numStringOIDs	4 + xmlDescriptionURL Length	4	int32	number of parameters within String-based OIDs for this device
blastSupportFlag	8 + xmlDescriptionURL Length	1	uint8	Flag to indicate whether this device supports OGP_BLAST_REQUEST message

OGP_START_INIT_BLAST Request

Message type: 0x4E

Message length: 2

Response required: ***OGP_START_INIT_BLAST Response***

Description: This informs the device that it is free to broadcast its responses to the standard DashBoard initialization queries immediately. This allows the device to avoid round-trip delays by sending its parameter descriptors, parameter values, and menus before DashBoard requests them.

Field	offset	length	format	Description
spare	0	1	uint8	placeholder for return code
state	1	1	uint8	OGP_START (0x00)

OGP_START_INIT_BLAST Response

Message type: 0xCE

Message length: 2

Response required: none

Description: This is the required response to OGP_START_INIT-BLAST. It indicates that a blast will start or that a blast has been denied.

Field	offset	length	format	Description
rc	0	1	uint8	return code **OGP_OK** - normal return (OGP Blast will follow) **OGP_UNSUPPORTED (0X01)** - OGP Blast disabled or not supported **OGP_RESPONSE_DENIED (0X03)** - OGP Blast already in progress
state	1	1	uint8	OGP_START (0x00) Or OGP_COMPLETE (0x01)

To implement the OGP_START_INIT_BLAST RESPONSE:

1. An initialization blast consists of the following messages:

```
TX: OGP_START_INIT_BLAST Response with OGP_START (0xCE)
TX: OGP_GET_DESCRIPTOR_Response (0xC7) for first OID
TX: OGP_GET_PARAM Response (0xC9) for first OID
```

2. Make sure to repeat for each OID in the param list in order, as shown below.

Note: You can also (optionally) pre-send your responses to the menu requests.

```
TX: OGP_GET_MENUSET_NAME Response (0xD0) for menu set 0
TX: OGP_GET_MENU_COUNT Response (0xD1) for menu set 0


TX: OGP_GET_MENU_NAME Response (0xD2) for first menu
TX: OGP_GET_MENU_OIDS Response (0xD3) for first menu
TX: OGP_GET_MENU_STATE Response (0xDA) for first menu
```

3. Repeat for each menu in the menu set, as shown below.

Tip: This includes menu sets 1 and 2.

```
TX: OGP_START_INIT_BLAST Response with OGP_COMPLETE (0xCE)
```

✶ IMPORTANT: A device responding to an initialization blast through an openGear Frame must limit its response rate to avoid interfering with normal communications on the CAN bus. Aresponse rate of 80 messages per second (or less) is required.

If additional OGP_START_INIT_BLAST messages are received during the blast, an OGP_START_INIT_BLAST Response must be sent with OGP_RESPONSE_DENIED

If the device receives a request message, it must respond to it even if it has already sent an appropriate response as part of its blast.

XML Side Loading of the Device Description

This method of speeding up device initialization involves adding an extra bit of information to the device response for the number of OIDs (*OGP_GET_NUMPARAMS* Response). It involves providing a URL for an XML device description. If it is provided, then DashBoard stops asking for the device description, and instead retrieves it from the specified XML document. It must contain all the device descriptor information including menu groups, menus, parameters, parameter values, and constraints, along with any other information typically provided in the device description.

You will need to dynamically generate the XML document from the OGP device to ensure the parameter values are read by the device. Normal parameter query messages are skipped, and DashBoard will not request the parameter value from the device. Instead, DashBoard will use the "value" attribute of the XML parameter tags as the parameter values, until additional updates to the value are sent asynchronously via OGP-Binary by the device.

To create an XML document from the OGP device

1. In the DashBoard tree view, right click on the OGP device.
2. Select **Save Configuration to File**.

   An example of what the XML device definition would look like for your device will be saved as an openGear Device ("OGD") file in the location of your choice.

You can use this XML file to prevent DashBoard from sending requests for the device description, but the XML file must contain all the definitions typically provided in the device description.

Troubleshooting Tip: If the URL is defined, DashBoard will attempt to fetch the XML document and use it to initialize the device. If the ethernet settings are known to be incorrect, the device should not provide this URL. If the XML cannot be reached, the device will fail to initialize using the XML side load method. As of DashBoard 9.5, if the XML file can not be accessed, DashBoard will default to using the OGP-Binary querying method to initialize the device. You can choose to continue device initialization using the OGP-Binary querying method, or update your ethernet settings to resume using the XML side loading initialization method.

Below is an outline of the XML file structure:

```
<?xml version="1.1" encoding="UTF-8"?>
<frame>
       <card>
              <params>
                 <param/>
                 <param/>
                . . .
              </params>
              <statusmenu>
                 <menu>
                     <param/>
                     <param/>
                     . . .
              </menu>
              <menu/>
              . . .
       </statusmenu>
       <configmenu>
              <menu>
                     <param/>
                     <param/>
                     . . .
              </menu>
              <menu/>
              . . .
       </configmenu>
       <menugroup>
              <menu>
                     <param/>
```

```
                        <param/>
                        . . .
                    </menu>
                    <menu/>
                    . . .
                </menugroup>
                <menugroup/>
                . . .
            </card>
            <card/>
            . . .
        </frame>
```

See below for an example of an OGD file based on the “Zeus Test Card” ZTC-8399:

```
<?xml version="1.1" encoding="UTF-8"?>
<card online="true" slot="3"
     status="0"
     statustext="OK" version="2.0">
     <params>
          <param access="0" maxlength="20" name="Product" oid="0x105"
               type="STRING" value="ZTC-8399" widget="default"/>
          <param access="0" maxlength="32" name="Supplier" oid="0x102"
               type="STRING" value="Ross Video Ltd." widget="default"/>
          <param access="0" maxlength="16" name="Board Rev" oid="0x108"
               type="STRING" value="0 " widget="default"/>
          <param access="0" maxlength="20" name="Serial Number"
               oid="0x106" type="STRING" value="562616007" widget="default"/>
          <param access="0" maxlength="20" name="Software Rev"
               oid="0x10B" type="STRING" value="2.01" widget="default"/>
          <param access="0" constrainttype="ALARM_TABLE"
               name="HW status" oid="0x202" precision="0" type="INT16"
               value="1" widget="default">
               <constraint key="0" severity="0">HW A-OK</constraint>
               <constraint key="1" severity="1">HW warning</constraint>
               <constraint key="2" severity="2">HW error 1</constraint>
               <constraint key="3" severity="2">HW error 2</constraint>
          </param>
          <param access="0" constrainttype="ALARM_TABLE"
               name="Signal status" oid="0x203" precision="0"
               type="INT16" value="0" widget="default">
               <constraint key="1" severity="1">????????</constraint>
               <constraint key="2" severity="1">No reference</constraint>
               <constraint key="3" severity="2">No input signal</constraint>
          </param>
```

```
<param access="0" constrainttype="INT_NULL"
     name="Voltage (mV)" oid="0x204" precision="0"
     type="INT16" value="12000" widget="default"/>
<param access="0" constrainttype="INT_NULL"
     name="Current (mA)" oid="0x205" precision="0"
     type="INT16" value="239" widget="default"/>
<param access="0" maxlength="32" name="Rear Module"
     oid="0x207" type="STRING" value="0.2 " widget="default"/>
<param access="0" constrainttype="INT_CHOICE" name="Fan Door"
     oid="0x54E" precision="0" type="INT16" value="2" widget="default">
     <constraint key="1">Closed</constraint>
     <constraint key="2">Open</constraint>
</param>
<param access="0" constrainttype="INT_NULL"
     name="CPU headroom" oid="0x209" precision="0"
     type="INT16" value="875" widget="default"/>
<param access="0" constrainttype="INT_NULL"
     name="RAM available" oid="0x20A" precision="0"
     type="INT16" value="121" widget="default"/>
<param access="0" constrainttype="INT_NULL" name="Boot Count"
     oid="0x20C" precision="0" type="INT16" value="472" widget="text"/>
<param access="0" maxlength="20" name="Total Power (W)"
     oid="0x301" type="STRING" value="2.4" widget="default"/>
<param access="0" constrainttype="INT_NULL"
     name="Load current (mA)" oid="0x302" precision="0"
     type="INT16" value="10" widget="default"/>
<param access="0" constrainttype="INT_NULL"
     name="Logic current (mA)" oid="0x303" precision="0"
     type="INT16" value="171" widget="default"/>
<param access="0" constrainttype="INT_NULL" name="Temp_1 (C)"
     oid="0x304" precision="0" type="INT16" value="41" widget="default"/>
<param access="0" constrainttype="INT_NULL" name="Temp_2 (C)"
     oid="0x305" precision="0" type="INT16" value="43" widget="default"/>
<param access="0" constrainttype="INT_NULL" name="Temp_3 (C)"
     oid="0x306" precision="0" type="INT16" value="40" widget="default"/>
<param access="0" constrainttype="INT_NULL" name="Temp_4 (C)"
     oid="0x307" precision="0" type="INT16" value="37" widget="default"/>
<param access="0" constrainttype="INT_CHOICE"
     name="Reference_1" oid="0x308" precision="0" type="INT16"
     value="0" widget="default">
     <constraint key="0">None</constraint>
     <constraint key="1">NTSC</constraint>
     <constraint key="2">PAL</constraint>
```

```
    </params>
    <statsmenu menuid="0" name="Status">
        <menu menuid="0" menustate="2" name="Product" staticid="0">
            <param access="0" name="Product" oid="0x105" type="MENU_OID"/>
            <param access="0" name="Supplier" oid="0x102" type="MENU_OID"/>
            <param access="0" name="Board Rev" oid="0x108" type="MENU_OID"/>
            <param access="0" name="Serial Number" oid="0x106" type="MENU_OID"/>
            <param access="0" name="Software Rev" oid="0x10B" type="MENU_OID"/>
        </menu>
        <menu menuid="1" menustate="2" name="Hardware" staticid="1">
            <param access="0" name="HW status" oid="0x202" type="MENU_OID"/>
            <param access="0" name="Signal status" oid="0x203" type="MENU_OID"/>
            <param access="0" name="Voltage (mV)" oid="0x204" type="MENU_OID"/>
            <param access="0" name="Current (mA)" oid="0x205" type="MENU_OID"/>
            <param access="0" name="Rear Module" oid="0x207" type="MENU_OID"/>
            <param access="0" name="CPU headroom" oid="0x209" type="MENU_OID"/>
            <param access="0" name="RAM available" oid="0x20A" type="MENU_OID"/>
            <param access="0" name="Boot Count" oid="0x20C" type="MENU_OID"/>
        </menu>
        <menu menuid="2" menustate="2" name="Test" staticid="2">
            <param access="0" name="Total Power (W)" oid="0x301" type="MENU_OID"/>
           <param access="0" name="Load current (mA)" oid="0x302" type="MENU_OID"/>
            <param access="0" name="Logic current (mA)" oid="0x303"
type="MENU_OID"/>
            <param access="0" name="Temp_1 (C)" oid="0x304" type="MENU_OID"/>
            <param access="0" name="Temp_2 (C)" oid="0x305" type="MENU_OID"/>
            <param access="0" name="Temp_3 (C)" oid="0x306" type="MENU_OID"/>
            <param access="0" name="Temp_4 (C)" oid="0x307" type="MENU_OID"/>
            <param access="0" name="Reference_1" oid="0x308" type="MENU_OID"/>
        </menu>
    </statsmenu>
    <configmenu menuid="1" name="Config">
        <menu menuid="256" menustate="2" name="Setup" staticid="256">
            <param access="0" name="Boot Count" oid="0x20C" type="MENU_OID"/>
        </menu>
        <menu menuid="262" menustate="2" name="Progress" staticid="262">
            <param access="0" name="Boot Count" oid="0x20C" type="MENU_OID"/>
        </menu>
    </configmenu>
</card>
```

Instantiate a New Connection to the Device

This method involves instantiating a new connection to the device directly using the Device Index URL reserved OID and creating an additional node to create and access this connection under your device's original node in the DashBoard Tree View.

The XML document specified by the Device Index URL reserved OID allows additional properties or entirely new nodes/levels of hierarchy to be specified under a device node in DashBoard's tree view. If one of the new nodes under the device is a connection to an OGP server, DashBoard will connect to this new server and place a "frame" node under the device node.

Note: The device connection is separate and is applicable if you are using an OGP binary device that lives inside of the OpenGear frame talking over the CAN bus. If you already have a dedicated connection, use either OGP Blast Mode or XML Side Loading of the Device Description to speed up device initialization, or switch into OGP-JSON.

To create a new device node using the Device Index URL

1. Create the XML file on the OGP device containing the information in the string below:

```
<?xml version="1.0" encoding="UTF-8"?>
<properties version="1.0">
     <children>
          <child>
               <entry key="equipmentType">editor-link</entry>
               <entry key="child-id">basic-menu</entry>
               <entry key="node-name">Default Menu System</entry>
          </child>
          <child>
<comment>DashBoard Device Connection Settings</comment>
<entry key="serviceUrl">service:broadcast-equipment</entry>
<entry key="equipmentType">opengear</entry>
<entry key="address">172.16.9.31</entry>
<entry key="port">5253</entry>
<entry key="node-id">device</entry>
<entry key="node-name">Jim's Frame</entry>
</child>
     </children>
</properties>
```

2. Populate the value of the Device Index URL into the reserved OID 0xFF0D in DashBoard.

   This will create a device node under the CAN bus which contains two nodes within it:

   Editor Link: This node points to the interface for the device node and provides access to the device parameters served over the CAN bus.

   Device Connection Settings: This node contains the connection information which establishes a dedicated connection to the device in a new frame node.

Additional Navigation Improvement

Once the new frame node is created, there are two additional reserved OIDs that can be specified for ease of navigation. By specifying these OIDs, you can turn the frame node into a click-able leaf node, which will allow you to navigate directly to the operator interface by double-clicking on the frame.

To turn the frame node into a click-able leaf node

1. Navigate to the reserved OID CONFIG_SLOT with the value FE07 and enter the value of the slot to be opened when the frame is opened.
2. Navigate to the reserved OID DISPLAY_OPTIONS with the value FF0B.
3. Under DISPLAY_OPTIONS_INDEX_HIDDEN, enter the same slot value.

Completing these steps will ensure that double clicking on the frame node will allow you to navigate directly to the operator interface.

Data Model

In This Chapter

This chapter describes the underlying data model for openGear and DashBoard Connect devices.

The following subjects are discussed:

Device Data Model

DashBoard stores a device's data representation in an object hierarchy:

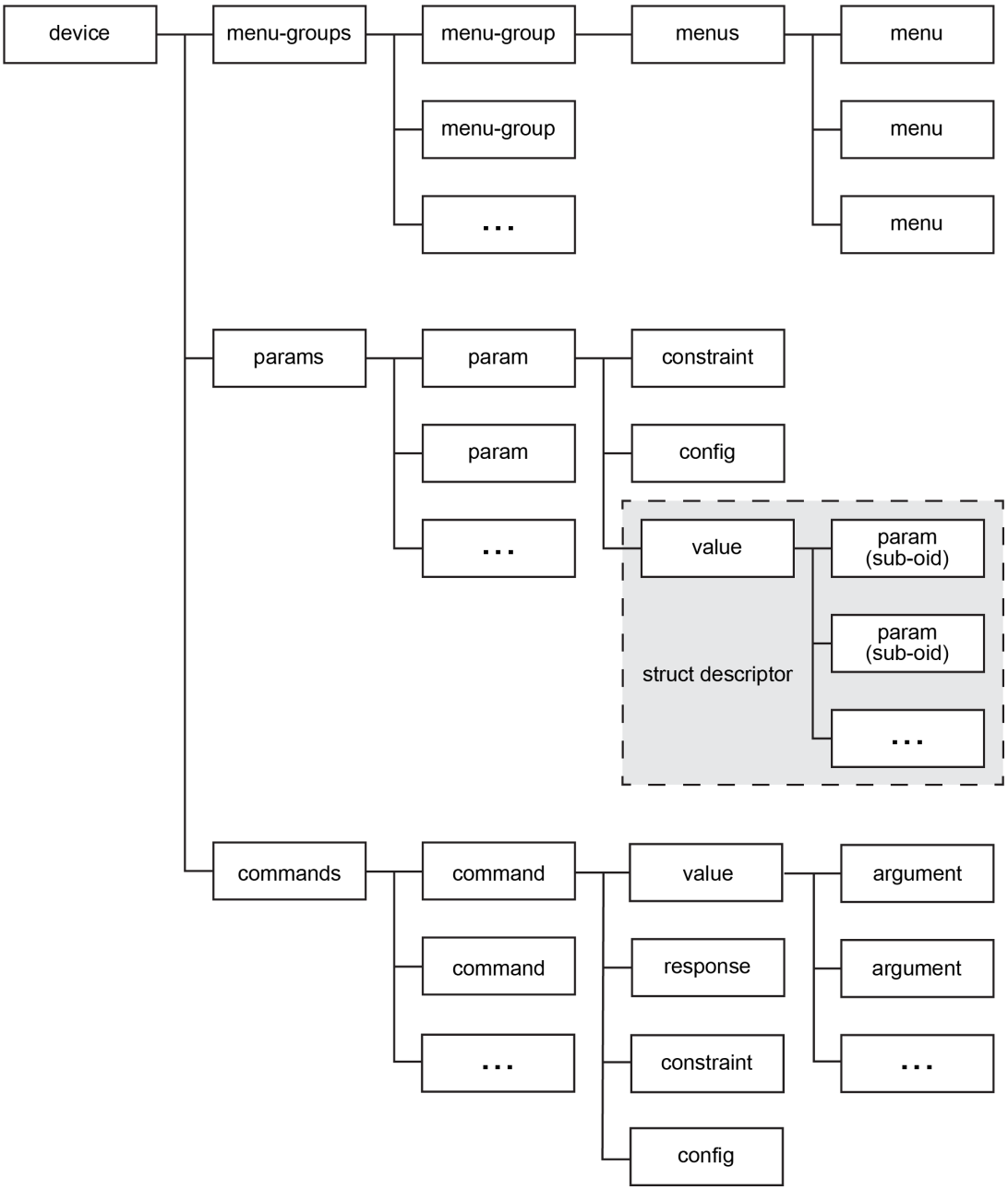


***Figure 3.1** - DashBoard Data Object Hierarchy*

This hierarchy is explicitly exposed in the JSON protocol and XML representation. Binary OGP does not explicitly reference the data through the object hierarchy, but individual data elements may be accessed via their OIDs.

All information regarding a device is encapsulated within the **device object**. Each node in the DashBoard tree is treated as an independent device object. The device object contains a list of parameters, menu-groups, and device commands (if any).

This section describes the device data model, and includes the following topics:

Parameters

The configuration and state of any device can be represented by a list of parameters holding information about the device, including:

- identification: device type and supplier name, software revision, etc.
- status: alarms, voltage, current, temperature, input signal presence and format, etc.
- configuration: user-specified setup parameters (gain, delay, output video format, etc.)

Each parameter is identified with an Object Identifier (OID), and consists of two parts: the descriptor and the value. The descriptor defines the structure of the data, and the value is the content, which is dependent on the descriptor. The descriptor may also specify a constraint, which limits the value to a certain set of valid values.

Object Identifiers (OIDs)

Each parameter is identified by a unique object ID (OID). Two types of OIDs are supported: numeric and string. The use of meaningful string OIDs is strongly recommended for new designs, as it clarifies code and simplifies the development of DashBoard Custom Panels. Handling of Numeric and String OID parameters use different message types. Devices implementing String OIDs must support both message types.

***Note**: OGP via CANbus does not support string OIDs. These may only be used on TCP/IP connections.*

Numeric OIDs

Numeric OIDs are 2-byte integers, and referenced in this document as a 16-bit hex value, for example: `0x0105`. In JSON messaging, numeric OIDs are encoded as strings. For example, the OID `0x0105` is encoded as the string `"0x105"`.

String OIDs

String OIDs allow text-based parameter identifiers, and must follow the following encoding rules:

- Must not contain spaces
- May only contain the following characters: a-z A-Z 0-9 .(dot) _ (underscore)
- Are case sensitive
- There is no set limit to the String OID identifier length; however string OIDs over 255 characters cannot be carried over CAN or TCP/IP binary protocol.
- Shall not begin with a number
- Shall not end with `.number`

A string OID identifier should not be confused with the parameter name. A string ID is the variable name, the Parameter Name is the display name for the parameter. For example a parameter may have the OID “mle.2.keyer.3.ckey-state” and the parameter name could be “Chroma Key”. Software refers to the value “mle.2.keyer.3.ckey-state”, but the default label on the DashBoard GUI would be “Chroma Key”.

Reserved OIDs

There are a number of reserved numeric OIDs which must not be used for other purposes. For a complete list of such OIDs, see “Reserved OIDs” on page 11–6.

Descriptors

Each parameter is defined using a descriptor containing its name, data type, data length, constraint (set of permitted values) and other information. This information is used to create an appropriate user interface for the device and to properly interpret and display parameter values reported by the device.

In OGP, when DashBoard first contacts a device, it requests the list of parameters for that device, and the descriptor for each parameter.

In JSON messaging, the device pushes the full set of parameter descriptors through the `device` object. Descriptor objects are identified by the naming convention `_d_oid`.

The descriptor for each parameter contains the following fields:

Field	Description
oid	Object Identifier for this Parameter.
version	Version of the descriptor. The current version is 2. Permitted versions are 0, 1 and 2. Versions 0 and 1 are identical to version 2, except that widget hints are ignored.
name	Parameter name to be displayed in a user interface This field provides the parameter name to be displayed in DashBoard. The name does not need to be unique. It may be ignored by some software (e.g. the SNMP agent).
data type and data size	**data type** — integer, float, string, or array thereof. **data size** — nominal size of the data field. Data type indicates the storage type for the parameter value. See also "Data Types" on page 3–34.
access	Read/write access indicator. This field indicates whether the parameter can be modified. This enables the control software to display an appropriate control for read-only values, or to disallow edits. In OGP, the supported values are as follows: Access: ACCESS_READWRITE Value: 0x01 Description: Parameter may be modified by the control client. Access: ACCESS_READONLY Value: 0x00 Description: Parameter is read-only, and may not be set by the client. ***Note**: JSON messaging defines the access via the readonly property in the param object.*
precision	Precision to be displayed for printed numbers. **When used with numbers** — this field defines the number of digits following the decimal point displayed for printed numbers. It applies mainly to floating point numbers. **When used with string arrays** — in binary OGP, this field specifies the maximum number of characters permitted in a single element in the array. If it is 0, then there is no per-element limit, and the maximum number of characters in a parameter value is shared arbitrarily amongst all elements in the array. Precision is ignored for string arrays in JSON notation.
widget	Graphical display hint for this parameter The widget hint specifies the type of graphical control that should be used to display this parameter. To ensure backward compatibility with DashBoard 1.0, widget hints are ignored if the version field is less than 2. See also "Customizing Menus Using Display Hints" on page 3–8.
constraint	An object specifying the set of permitted values for the parameter. Constraints allow data to be limited to a certain range or certain values.

Constraints

Constraints are an important part of the parameter descriptor. They specify the range of valid values for each parameter. Certain constraint types also impact how the parameter is displayed within DashBoard. Certain widgets

require specific constraints, while others may behave in different manners depending upon the constraint applied to the parameter. For array parameters, the same constraint applies to each element of the array.

***Note**: The constraint should be considered a HINT to DashBoard. Even if the device specifies a constraint, DashBoard may send values that are not within it. It is up to the device to reject invalid values by sending back the correct value to a parameter set request.*

In OGP, constraints are specified through a numeric identifier called `ctype`. In JSON, the constraint type is specified by its Constraint Name. The supported constraint types are as follows:

Constraint Name	ctype	Param Types	Description
NULL_CONSTRAINT	0	All	Parameter is unconstrained.
RANGE_CONSTRAINT	1	INT16 INT32 INT16_ARRAY INT32_ARRAY FLOAT32 FLOAT_ARRAY32	Parameter is bounded by a min-max range. Display min-max range may be different from the value range.
CHOICE_CONSTRAINT	2	INT16 INT16_ARRAY	Parameter must be selected from a set (enumeration) of name-value pairs (up to 255 choices)
EXTENDED_CHOICE	3	INT16 INT16_ARRAY	Parameter must be selected from a set (enumeration) of name-value pairs (up to 255 choices)
STRING_CHOICE	4	STRING STRING_ARRAY	Provides a set of available choices. Parameter may be selected from this set, but arbitrary values are also permitted.
RANGE_STEP_CONSTRAINT	5	INT16 INT32 INT16_ARRAY INT32_ARRAY FLOAT32 FLOAT_ARRAY32	Parameter is bounded by a min-max range. Step size indicates the amount to increment/decrement the value each time it is changed.
ALARM_TABLE	10	INT16 INT32 INT16_ARRAY	Each bit in the parameter is a status flag, so param can display 16 or 32 concurrent named error conditions.
EXTERNAL_CONSTRAINT	11	All	Indicates that the constraint is encoded in an external object, rather than encoded within the descriptor.

Constraints are normally embedded within the parameter descriptor however; they may also be encoded separately as external objects (which allow longer choice lists, etc.).

A detailed definition of each constraint type, and rules for encoding each constraint, are provided below.

Unconstrained

To leave a parameter unconstrained, use the NULL_CONTRAINT constraint. Any parameter which does not have any other constraint applied must specify the NULL_CONSTRAINT.

Range Constraints

To constrain a numerical parameter to a specific range of values, the RANGE_CONSTRAINT or RANGE_STEP_CONSTRAINT must be specified. Both constraint types allow a minimum and maximum parameter value (minValue, maxValue). Additionally, an optional display minimum and maximum value (minDisp, maxDisp) may also be specified. This allows the display range to map to normalized parameter range. The value to be displayed is determined by the following linear mapping:

$$displayed\ value = minDisp + \frac{(value - minValue) \times (maxDisp - minDisp)}{(maxValue - minValue)}$$

Note that minDisp and maxDisp must be the same data type as the parameter. For example, to display the value of a 12-bit register (0-4095) as a percentage, set the following values:

- (minValue, maxValue) = (0, 4095)
- (minDisp, maxDisp) = (0, 100)

The difference between RANGE_CONSTRAINT and RANGE_STEP_CONSTRAINT is the that latter also allows a step size to be specified. The step is specified in the same data type as the parameter and is the minimum change increment on the parameter value (not necessarily the display value).

***Note**: It is strongly recommended that the range (maxValue – minValue) be evenly divisible by the provided step size. Otherwise, when starting from the minimum, the parameter will use values of minValue + n * stepSize and when starting from the maximum, the parameter will use values of maxValue – n * stepSize.*

Range constraints applied to an array parameter apply to all members of the array.

Choice Constraints

Choice constraints allow a parameter to provide a list of choices. CHOICE_CONSTRAINT and EXTENDED_CHOICE constraints provide a mechanism to create a set of enumerated values for an INT16 or INT32 parameter. This allows integer types to be limited to a specific set of valid values, as well as providing a mechanism to provide text choices in the DashBoard UI for these parameters.

STRING_CHOICE constraint provides a set of default values which may be populated in a STRING, however unlike CHOICE_CONSTRAINT and EXTENDED_CHOICE, it does not limit the user to only these values; any arbitrary value may still be filled into the string.

***Note**: In JSON messaging, all choice constraints are handled as EXTENDED_CHOICE constraints.*

External Constraints

An EXTERNAL_CONSTRAINT is used to indicate that the constraint for this parameter is provided in an external object, rather than embedded within the parameter descriptor.

This constraint simply provides a reference to the external object.

Alarms

Assigning an ALARM_TABLE constraint to an integer parameter tells DashBoard to treat the integer as an array of alarms. When alarms are set, they will impact the overall status reporting of the device.

Parameter Structure Objects

DashBoard allows user defined structures to be defined within parameters, called structs. Structs are defined by encoding a `struct` within the value object of a parameter. This is done by inserting an array of sub-OID descriptors (see "param Object (descriptor)" on page 4–59) into the value field of a parameter. Structs must have their type set to `STRUCT` or `STRUCT_ARRAY`.

A parameter may inherit the struct descriptor from another parameter through use of a struct constraint (see "constraint Object (Struct Constraints)" on page 4–54), which specifies a templateoid. The templateoid specifies the OID of a parameter whose descriptor will be inherited, thus eliminating the need to define identical struct descriptor for each instance of a struct parameter.

***Note**: structs are not supported on the binary OGP protocol.*

Parameter References

Sub-params within a structure may also be defined as references to other parameters. These behave much like C++ or Java variable references. A parameter reference points to the referenced parameter's type, attributes and constraints.

In JSON messaging, parameter references are identified by the naming convention `_r_oid`.

Required Parameters

Each device may create whatever parameters it requires to properly control that device, however there are certain parameters that each device must specify. The required parameters are as follows:

Parameter	OID	Function
PRODUCT_NAME	0x0105	The name of the Product.

Menu-Groups

How a device is displayed in DashBoard is determined by the menu data provided by the device. DashBoard provides two methods for a device to specify menu layout and structure:

- Default openGear layout
- openGear Layout Markup Language (OGLML)

Default Menu Layout

The default menu layout is designed to make it very simple for devices to display a menu structure. Each menu consists of a name and a list of object identifiers specifying the parameters to be displayed in the menu. Menus are organized into groups, where each group consists of a name and an array of menus.

Menus are divided into menu groups. The default layout displays only two groups:

- Group 0: Status (read-only)
- Group 1: Configuration

Below is an example of the default layout:

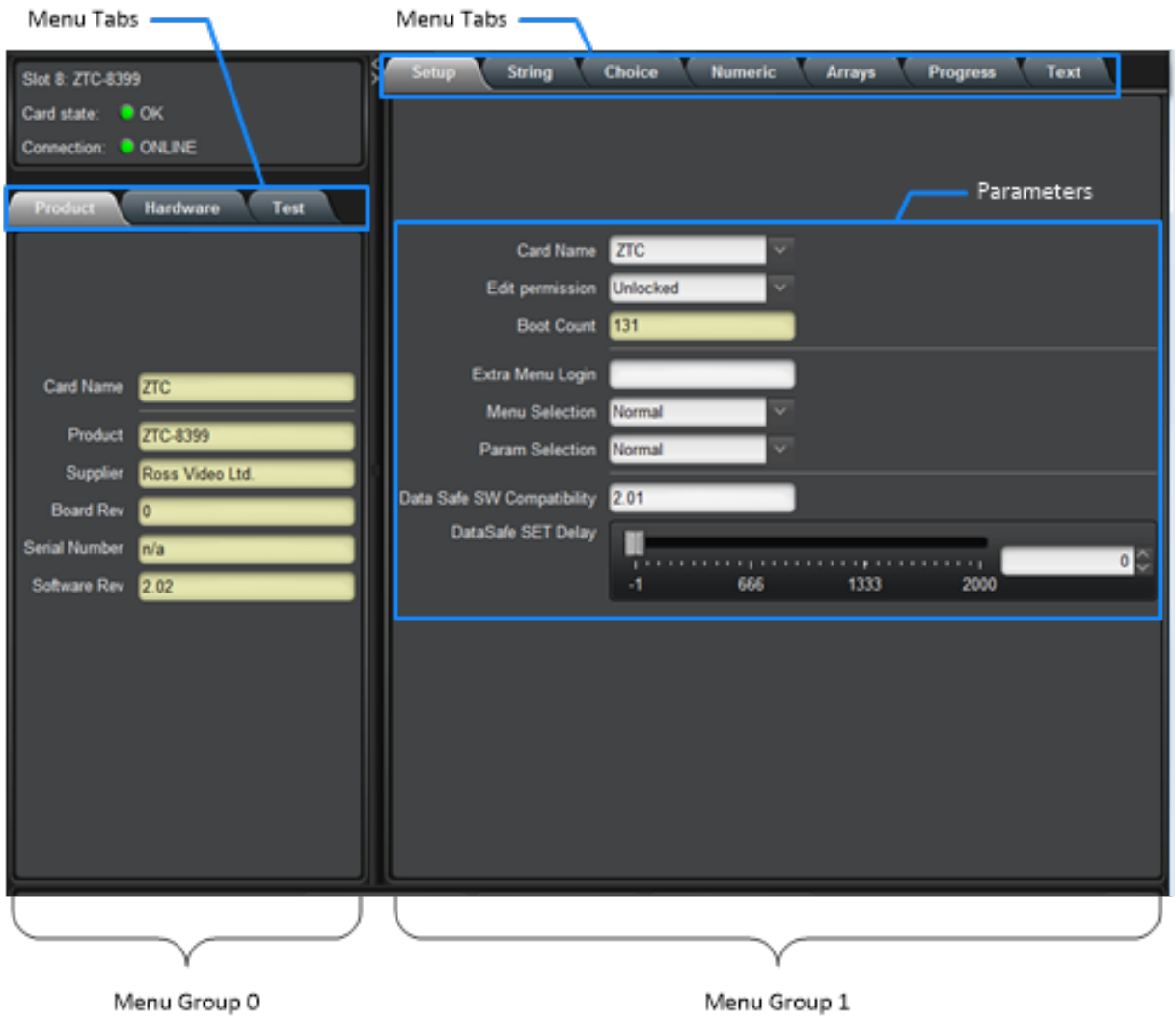


Figure 3.2 *- Default Menu Layout*

Each product may define any number of menus and groups, but the DashBoard control system recognizes only two groups in the default UI layout: group 0 = status parameters (read only), and group 1 = configuration parameters. Other menu groups are not displayed in the default UI layout presented by DashBoard, but may be used in OGLML UI layouts or for integration into DashBoard User Rights Management.

OGLML Menu Layout

Advanced menu layouts are available with openGear Layout Markup Language (OGLML). OGLML documents can replace an individual menu or the entire device configuration in DashBoard. For more information about OGLML, see the ***DashBoard CustomPanel Development Guide (8351DR-007-xx)***.

Customizing Menus Using Display Hints

The descriptor for each parameter includes a widget hint to allow the device designer to specify the type of control to be used to display the parameter. The hints available depend on the parameter type, the constraint type, and the values in the constraint for each parameter. This allows the designer to customize the menu for each device.

DashBoard 1.0 ignored widget hints and provided a default control based on parameter and constraint type. For backwards compatibility, DashBoard 2.0 (and later) ignores widget hints for parameters with the version field set to 0 or 1, providing the same default behavior as DashBoard 1.0. To use widget hints, it is necessary to set the version field within the parameter to 2.

When a read-only parameter provides a widget hint, a read-only version of the parameter's preferred widget is used. The exceptions are WIDGET_DEFAULT (displays like DashBoard 1.0) and Alarm Tables (display the alarm). Hints for status menu parameters are overridden for correct display in that space.

***Note**: OGP uses a numerical value to specify a widget hint. JSON messaging uses the JSON Name or Widget Name to specify the hint. Device XML and OGLML uses the Widget Name to specify the hint.*

Universal Hints

The following widget hints may be used for any parameter type:

Widget Name	JSON Name	Value	Description
WIDGET_DEFAULT	default	0	DashBoard chooses what it thinks is the best widget to use for the parameter type and constraint (makes the parameter work like it does with DashBoard 1.0).
WIDGET_TEXT_DISPLAY	text-display	1	shows a read-only version of the parameter value (uses same widget that is shown when WIDGET_DEFAULT parameter is set to read-only).
WIDGET_HIDDEN	hidden	2	still uses space on the menu page and shows the label for the parameter but show a blank area on the menu page where the widget would be.
WIDGET_LABEL	label	100	Displays the value of the parameter as a read-only label.

Separators, Titles and Layout Hints

The following hints are used with string parameters to provide separators, titles, and extended layout options for menus. Parameters using these widget hints are treated as read only and constant; they do not update live on the screen.

***Note**: For more comprehensive layouts, OGLML is recommended, as it provides more flexibility than the hints described in this section.*

The following table describes the hints. Examples follow the table.

Widget Name	JSON Name	Value	Description
WIDGET_TITLE_LINE	title-line	5	Displays the value of the String parameter as a label with all other parameter labels and a line across the content area of the menu page.
WIDGET_LINE_ONLY	line	6	Displays a line across the content area of the menu page with no label on the left.
WIDGET_TITLE_ONLY	title	7	Displays the value of the String parameter as a label with empty space in the content area of the menu page.
WIDGET_PAGE_TAB	tab	8	Creates a third-level tab within the menu page. The value of the parameter is used as the tab label.
WIDGET_TITLE_HEADER	header	10	Displays a title over the content area of the menu with the value of the parameter used as the header text.

WIDGET_TITLE_LINE (5)

This displays the value of the String parameter as a label aligned with all other parameter labels, and a line across the content area of the menu page. The name of the parameter is ignored.

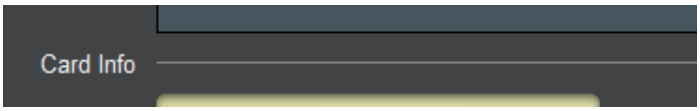


***Figure 3.3** - WIDGET_TITLE_LINE Hint*

WIDGET_LINE_ONLY (6)

This displays a line across the content area of the menu page with no label on the left. The name and value of the parameter are ignored.

***Figure 3.4** - WIDGET_LINE_ONLY Hint*

WIDGET_TITLE_ONLY (7)

This displays the value of the String parameter as a label with empty space in the content area of the menu page. The name of the parameter is ignored.

***Figure 3.5** - WIDGET_TITLE_ONLY Hint*

WIDGET_PAGE_TAB (8)

Whenever a new String parameter with a WIDGET_PAGE_TAB hint is found on a menu page, a new third-level tab is created inside of that menu page. The label on that tab shows the value of the String parameter. All parameters listed after each WIDGET_PAGE_TAB String parameters (until the next such parameter) are placed on a menu page inside of that third-level tab.

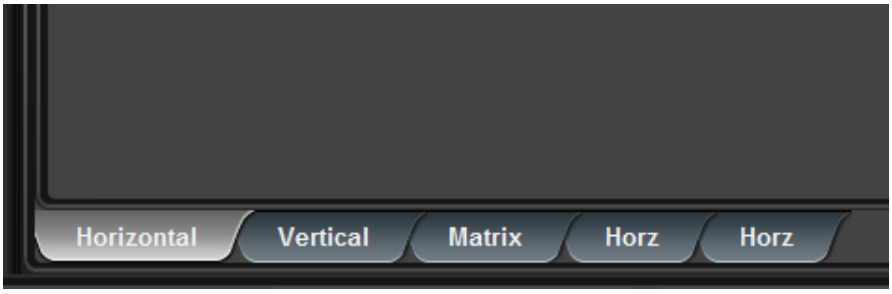


***Figure 3.6** - A Menu with WIDGET_PAGE_TAB Hints*

***Note**: Whenever WIDGET_PAGE_TAB hints are used on a menu, the first OID in the menu should be for a String parameter with a widget hint defining the first tab's label.*

WIDGET_TITLE_HEADER (10)

Displays a title over the content area of the menu with the value of the parameter used as the header text. No label is shown on the left and the name of the parameter is ignored.

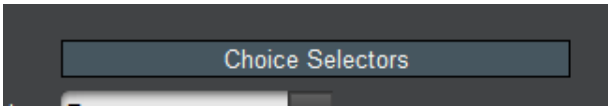


***Figure 3.7** - WIDGET_TITLE_HEADER Hint*

Array Layout Hints

***Note**: For more comprehensive layouts, OGLML is recommended, as it provides more flexibility than the hints described in this section.*

By default, all array parameters are displayed horizontally across a menu page. Adjacent OIDs of the same size will format in DashBoard in a tabular format. For example, if there are three array parameters with four elements each, the layout would appear as follows:

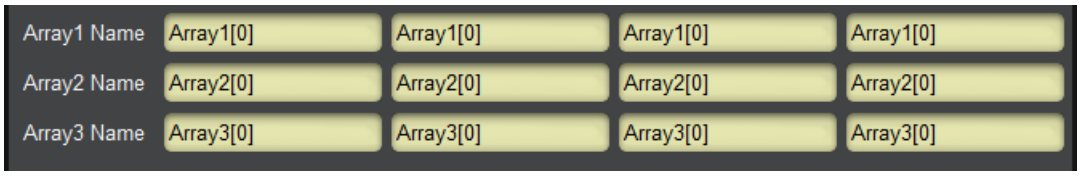


Figure 3.8 *- Default Array Layout (three parameters with four elements each)*

Column headers can be added by adding a read-only INT16_ARRAY parameter to the menu immediately before the other arrays (widget hint WIDGET_ARRAY_HEADER_HORIZONTAL). The parameter is expected to have a choice constraint. The string values of the elements of this parameter provide the column headers. The resulting display is as follows:

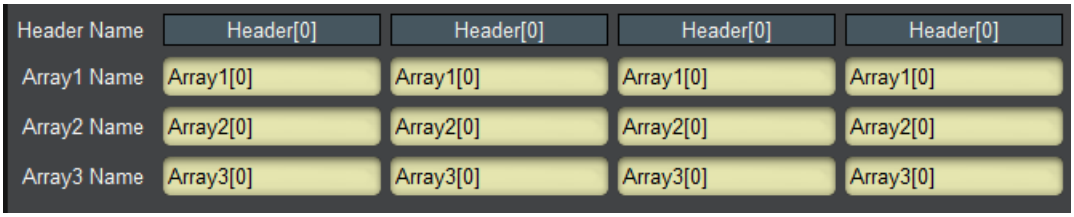


Figure 3.9 *- WIDGET_ARRAY_HEADER_HORIZONTAL Hint*

Array elements can also be given a vertical layout. Changing the widget hint for the header array to WIDGET_ARRAY_HEADER_VERTICAL provides the following layout:

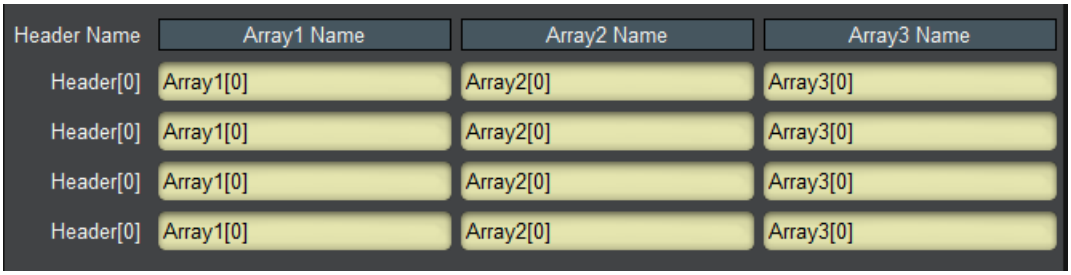


Figure 3.10 *- WIDGET_ARRAY_HEADER_VERTICAL Hint*

Array layout can be specified by including a read-only INT16_ARRAY parameter as a header, with one of the following widget hints:

Widget Name	JSON Name	Value	Description
WIDGET_ARRAY_HEADER_VERTICAL	array-horizontal	15	Indicates that the associated array parameter and all subsequent parameters should be displayed in a vertical layout.
WIDGET_ARRAY_HEADER_HORIZONTAL	array-vertical	16	Indicates that the associated array parameter and all subsequent parameters should be displayed in a horizontal layout.

Normally sequential array OIDs will be formatted as a single table. If it is desired to break a block of sequential array OIDs into multiple tables, it is necessary to insert a non-array OID, or switch from a horizontal layout hint to a vertical layout hint (or vice versa). If multiple arrays of different size are encoded with different sizes, the layout may be unpredictable.

WIDGET_ARRAY_HEADER_VERTICAL (15)

This hint indicates that the associated array parameter and subsequent parameters should be displayed in a vertical layout. The elements of the parameter will be used as row labels for display. The names of the following arrays are used as column labels. The header should be a read-only INT16_ARRAY parameter with a choice constraint to allow meaningful text labels. The elements of each array are displayed as specified by the widget hint for that array.

The vertical array layout will be applied until another WIDGET_ARRAY_HEADER_VERTICAL starts a new set of vertical columns, a WIDGET_ARRAY_HEADER_HORIZONTAL declares that subsequent arrays should be laid out horizontally, a non-array element in found on the page, or the end of the menu page is reached.

Figure 3.11 shows an INT16_ARRAY parameter named "Channel", provides a vertical layout and row labels for 7 array parameters named “Channel Update”, “Source”, “Vertical Channel”, “Delay Array (ms)”, “Gain (dB)”, “Invert”, and “Destination”.

Figure 3.11 *- INT_16_ARRAY Vertical Layout Example*

WIDGET_ARRAY_HEADER_HORIZONTAL (16)

The WIDGET_ARRAY_HEADER_HORIZONTAL is used to create a header over a horizontal array. It will also will end a block of vertical array elements. Each element in the header parameter will be displayed as a column header.

Figure 3.12 shows an INT16_ARRAY parameter named "Channel" providing a horizontal layout and column labels for 7 array parameters named “Horizontal Channel”, “Source”, “Delay Array (ms)”, “Gain (dB)”, “Invert”, “Destination” and “Transition”.

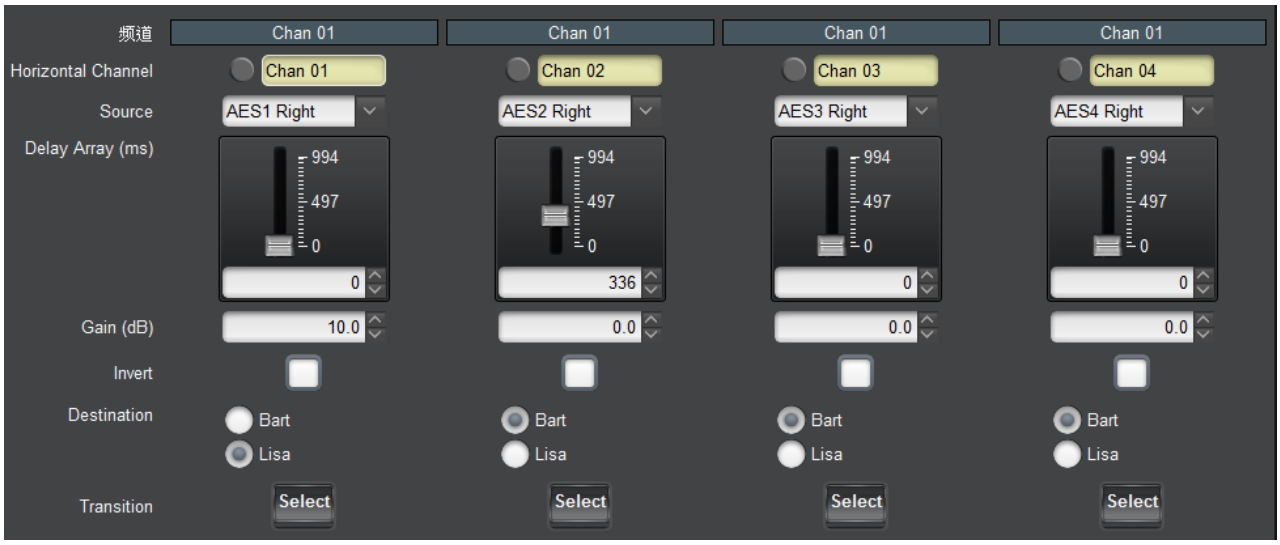


Figure 3.12 *- INT16_ARRAY Horizontal Layout Example*

INT16 / INT32 Parameters with Choice Constraints

The following hints apply to INT16, INT16_ARRAY, INT32, and INT32_ARRAY Parameters provided that they use a constraint of type CHOICE or EXTENDED_CHOICE. There are some restrictions for certain hints (checkboxes and toggle buttons are only valid for two-choice constraints, buttons with and without prompts are only valid for single-choice and two-choice constraints). If a widget hint is used incorrectly, the combo box will be substituted in place of the chosen widget.

The following table describes the hints. Examples follow the table.

Widget Name	JSON Name	Value	Description
WIDGET_COMBO_BOX	combo	7	Displays a dropdown list of selectable options. This is the default widget used for any choice parameter with more than one choice provided. If other widget hints are used incorrectly, the combo box is shown instead.
WIDGET_CHECKBOX	checkbox	8	Displays a checkbox. Checkboxes only apply to parameters with exactly two choices. The first choice is considered false or unchecked; the second choice is considered true or checked.
WIDGET_RADIO_HORIZONTAL	radio	9	Displays a radio button for each integer value option. The radio buttons are placed beside each other horizontally on the page.
WIDGET_RADIO_VERTICAL	radio-vertical	10	Displays a radio button for each integer value option. The radio buttons are placed in a vertical column.
WIDGET_BUTTON_PROMPT	button-prompt	11	Provides a button with confirmation prompt. Whenever the button is pressed and confirmed, the parameter value is send to the device.

Widget Name	JSON Name	Value	Description
WIDGET_BUTTON_NO_PROMPT	button	12	Provides a button without confirmation prompt. Whenever the button is pressed, the parameter value is send to the device.
WIDGET_BUTTON_TOGGLE	toggle	13	Displays a toggle buttons. This hint applies only to parameters with exactly two choices. The first choice is shown when the button is up (not pressed); the second choice is shown when the button is down (pressed).
WIDGET_FILE_DOWNLOAD	file-download	18	Display a file download widget. This hint requires an external object with an OID matching the value of the parameter.
WIDGET_MENU_POPUP	menu-popup	20	Each value in the parameter must refer to the menu ID of an OGP Menu. The choice corresponding to the parameter value has its name used as the value displayed on a button. When the button is pressed, the menu with an OID corresponding to the parameter value is displayed in a popup menu.
WIDGET_RADIO_TOGGLE_BUTTONS	radio-toggle	22	Displays a toggle button for each integer value option. The toggle buttons are placed beside each other horizontally on the page.
WIDGET_TREE	tree	31	Displays a tree control. Tree elements are defined by the elements of the choice constraint. The tree hierarchy is defined by "-" characters at the beginning of the choice. See detailed description below for more information.
WIDGET_TREE_POPUP	tree-popup	32	Displays the tree (same definition as WIDGET_TREE) in a combo box control. See detailed description below for more information.

WIDGET_COMBO_BOX (7)

Displays a dropdown list of selectable options. This is the default widget used for any choice parameter with more than one choice provided.

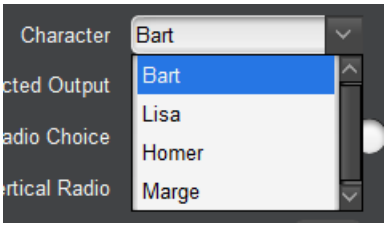


***Figure 3.13** - WIDGET_COMBO_BOX Hint*

WIDGET_CHECKBOX (8)

Displays a checkbox. Checkboxes apply only to integer choice constraints with exactly two choices. The first choice is considered false or unchecked; the second choice is considered true or checked.

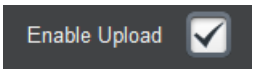


***Figure 3.14** - WIDGET_CHECKBOX Hint*

WIDGET_RADIO_HORIZONTAL (9)

Displays a radio button for each element in the choice constraint. The radio buttons are placed beside each other horizontally on the page.

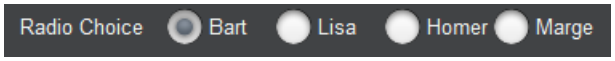


***Figure 3.15** - WIDGET_RADIO_HORIZONTAL Hint*

WIDGET_RADIO_VERTICAL (10)

Displays a radio button for element in the choice constraint. The radio buttons are placed in a column vertically on the page.

***Figure 3.16** - WIDGET_RADIO_VERTICAL Hint*

WIDGET_BUTTON_NO_PROMPT (12)

This hint can be used only for a parameter having a choice constraint with one or two choices. It displays a button with the name of the first choice as the button label. When the button is pressed, a parameter set request is sent to the device immediately (without user confirmation). If the parameter has only one choice, the value of that choice is sent to the device. If the parameter has two choices, the value of the second choice is sent. The device should normally reset the parameter value to the first choice when it acknowledges the set request.

Figure 3.17 shows a single-choice parameter named "Factory Defaults" with a hint of WIDGET_BUTTON_NO_PROMPT and a value of "Reset". There will be no confirmation dialog.

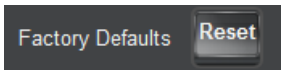


***Figure 3.17** - WIDGET_BUTTON_NO_PROMPT Hint*

WIDGET_BUTTON_PROMPT (11)

This hint can only be used for a parameter having a choice constraint with one or two choices. It is the default widget used when only one choice is available. It displays a button with the name of the first choice as the button label. When the button is pressed, a confirmation dialog is displayed before sending anything to the device. The dialog uses the format: “[Button Label] [Parameter Name]?” So a choice called “Reset” with a parameter named “Parameter Values” would display “Reset Parameter Values?” as the prompt. When the button is pressed and confirmed, a parameter set request is sent to the device. If the parameter has only one choice, the value of that choice is sent to the device. If the parameter has two choices, the value of the second choice is sent. The device should normally reset the parameter value to the first choice when it acknowledges the set request. If a two-state button is desired, see “**WIDGET_BUTTON_TOGGLE (13)**” on page 3–16.

Figure 3.18 shows single-choice parameter named "Factory Defaults" with a hint of WIDGET_BUTTON_PROMPT and a value of "Reset".

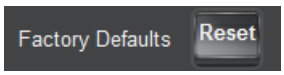


Figure 3.18 *- WIDGET_BUTTON_PROMPT Hint*

Note*: Two choices are necessary for using WIDGET_BUTTON_PROMPT and WIDGET_BUTTON_NO_PROMPT with array parameters.*

WIDGET_BUTTON_TOGGLE (13)

Toggle buttons work exactly the same was a checkbox. The toggle button applies only to integer constraints with exactly two choices. The name of the first choice is shown when the button is up (not pressed) and the name of the second choice is shown when the button is down (pressed).

Figure 3.19 shows a two-choice integer parameter named "Bold Toggler" with choice 1 set to "First Value" and choice 2 set to "Second value" The figure shows the button's display for both before and after a button toggle.

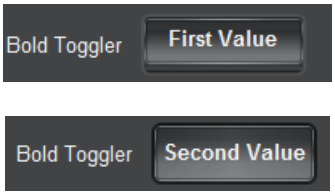


Figure 3.19 *- WIDGET_BUTTON_TOGGLE Hint*

WIDGET_FILE_DOWNLOAD (18)

This hint requires that an external object with an OID matching the value of the parameter be available. For each choice in the parameter's choice constraint, the choice value represents an external object's OID and the value represents the filename to display. When the 'save' button is pressed, DashBoard requests the external object with the given OID and save the external object's bytes to the filename/location defined by the user (default filename is defined in the choice constraint).

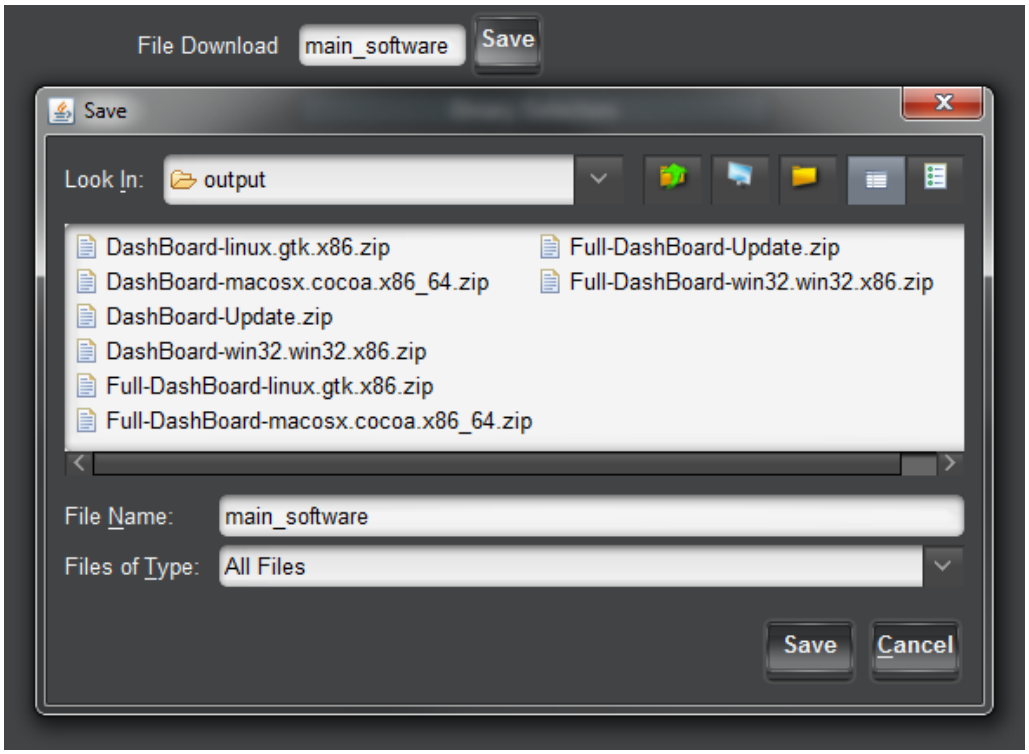


Figure 3.20 *- WIDGET_FILE_DOWNLOAD Hint*

Note*: Information about requesting/returning external objects is defined in section 3-19. Information about uploading data to a device is defined in section 5.*

WIDGET_MENU_POPUP (20)

This hint requires that an OID Menu with a menu ID matching the value of the parameter be available. For each choice in the parameter's choice constraint, the choice value represents a menu's ID and the choice name represents the label to display on the button. When the button is pressed, DashBoard displays the menu with the given ID as a popup menu.

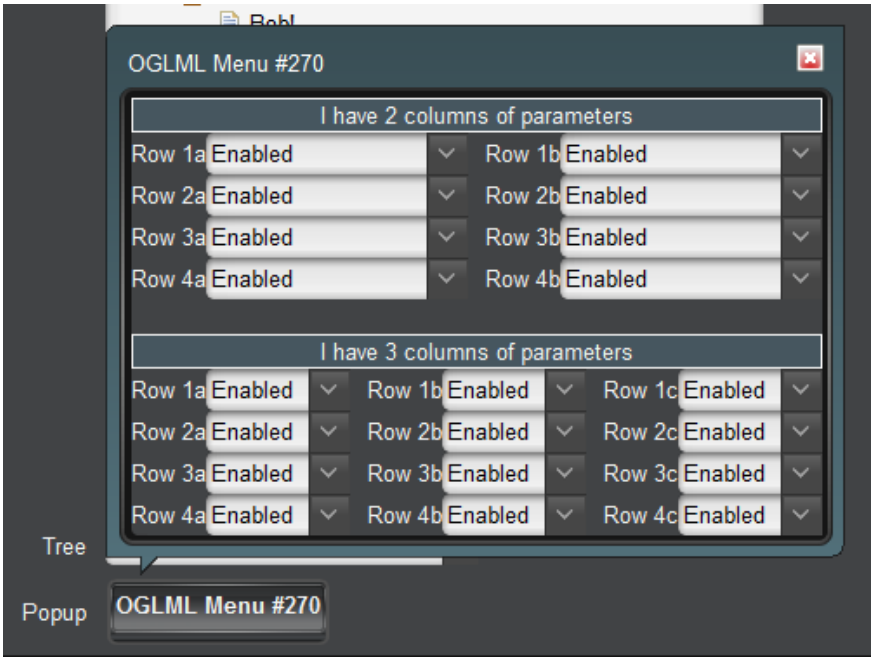


Figure 3.21 *- WIDGET_MENU_POPUP Hint*

WIDGET_RADIO_TOGGLE_BUTTONS (22)

Displays a radio toggle button for each integer value option. The radio toggle buttons are placed beside each other horizontally on the page.

Figure 3.22 *- WIDGET_RADIO_TOGGLE_BUTTONS Hint*

WIDGET_TREE (31)

Displays a tree control. Tree elements are defined by the elements of the choice constraint. The tree hierarchy is defined by "-" characters at the beginning of the choice. When an element in the tree is selected, the parameter value is set to the value of the selected choice. All other expand/collapse changes are local only to the DashBoard on which the change occurred.

"+" indicates that an element should be expanded by default.

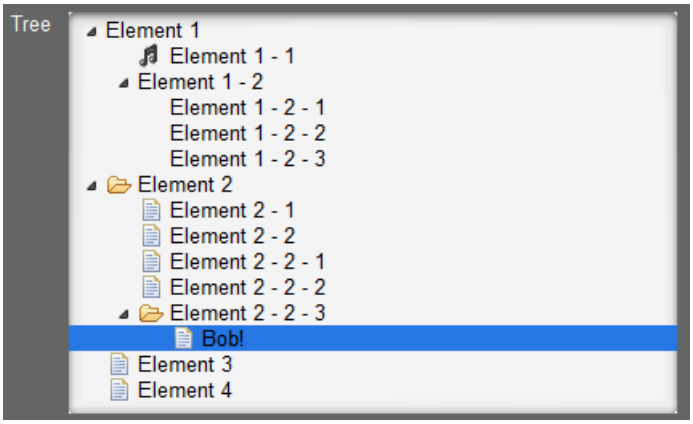


Figure 3.23 *- WIDGET_TREE Hint*

The tree shown in **Figure 3.23** is defined by the following list of choices:

1. Element 1<i:>
2. +Element 1 - 1<i-u:http://127.0.0.1/icons/small/sound2.png>
3. +Element 1 - 2<i:>
4. +-Element 1 - 2 - 1<i:>
5. +-Element 1 - 2 - 2<i:>
6. +-Element 1 - 2 - 3<i:>
7. Element 2
8. +-Element 2 - 1
9. +-Element 2 - 2
10. +-Element 2 - 2 - 1
11. +-Element 2 - 2 - 2
12. +-Element 2 - 2 - 3
13. +--Bob!
14. Element 3
15. Element 4

WIDGET_TREE_POPUP (32)

Displays the tree (same definition as WIDGET_TREE) in a combo box control. This functions the same as WIDGET_TREE, with the difference that only the currently selected item shows by default. When the user clicks on the value, a popup appears, allowing selection to be made.

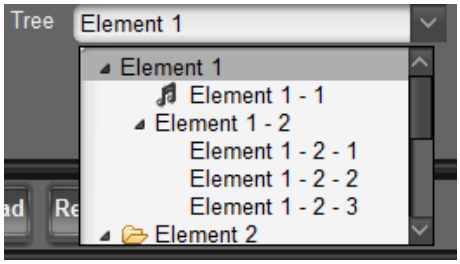


***Figure 3.24** - WIDGET_TREE_POPUP Hint*

Hints for Numeric Parameters with Other Constraints

The following hints are for INT16, INT16_ARRAY, INT32, INT32_ARRAY, FLOAT32, and FLOAT_ARRAY32 parameters and arrays with constraints other than choices. Most hints have specific restrictions.

The following table describes the hints. Examples follow the table.

Widget Name	JSON Name	Value	Description
WIDGET_SLIDER_HORIZONTAL	slider-horizontal	3	Displays a horizontal slider control. This is the default control for range-bounded integer and floating point parameters when they are not used in an array.
WIDGET_SLIDER_VERTICAL	slider-vertical	4	Displays a vertical slider control. This is the default control for range-bounded integer and floating point array parameters.
WIDGET_SPINNER	spinner	5	Displays a spinner (entry field plus up/down arrows). This is the default for unbounded INT16 parameters. This cannot be used for unbounded FLOAT32 or INT32 parameters.
WIDGET_TEXTBOX	text	6	Displays a numeric entry field. This is the default for unbounded FLOAT32 and INT32 parameters.
WIDGET_IP_ADDRESS	ip	14	Displays an IP Address entry field. Only works with unconstrained INT32 parameters.
WIDGET_PROGRESS_BAR	progress	17	Displays a read-only progress bar control.

Widget Name	JSON Name	Value	Description
WIDGET_AUDIO_METER	level-meter	19	Displays a read-only audio level meter control with green, yellow, and red markers.
WIDGET_TIMER	timer	21	Display a label that counts down from the parameter value to 0 when double-clicked.
WIDGET_COLOR_CHOOSER	color-picker	23	Put a color chooser as an element in the UI. Changes made to the color chooser are instantly sent to the device. Color values are INT32 values in ARGB format.
WIDGET_SLIDER_HORIZONTAL_NO_LABEL	slider-horizontal-nolabel	24	Display a horizontal slider control with no label.
WIDGET_SLIDER_VERTICAL_NO_LABEL	slider-vertical-nolabel	25	Display a vertical slider control with no label.
WIDGET_VERTICAL_FADER	fader	26	Display a vertical slider that looks like a fader bar.
WIDGET_TOUCH_WHEEL	wheel	27	Display a touch wheel control.
WIDGET_HEX_SPINNER	spinner-hex	28	Displays a spinner (entry field plus up/down arrows). Display the value in Base 16.
WIDGET_ABSOLUTE_POSITIONER	positioner	29	Provides a 2-axis absolute positioning element. When used as an INT16, the 8 LSBs represent the X coordinate and the 8 MSBs represent the Y coordinate. When used as an INT32, the 16 LSBs represent the X coordinate and the 16 MSBs represent the Y coordinate. A crosshair in a box can be dragged to the absolute position of the value in 2-D space.

Widget Name	JSON Name	Value	Description
WIDGET_ABSOLUTE_CROSSHAIR	Joystick-crosshair	30	Position a value in 2-D space. When used as an INT16, the 8 LSBs represent the X coordinate and the 8 MSBs represent the Y coordinate. When used as an INT32, the 16 LSBs represent the X coordinate and the 16 MSBs represent the Y coordinate. A crosshair that snaps to the center when released makes changes in +/- X, +/- Y relative to the offset from the center.
WIDGET_JOY_STICK	joystick	34	Position a value in 2-D space. When used as an INT16, the 8 LSBs represent the X coordinate and the 8 MSBs represent the Y coordinate. When used as an INT32, the 16 LSBs represent the X coordinate and the 16 MSBs represent the Y coordinate. Displays a joystick and modifies the X, Y values as the joystick is dragged north, south, east, and west of the center.
WIDGET_COLOR_CHOOSER_POPUP	color-picker-popup	33	Displays a combo box control showing the 'current' color. On click, show the color chooser. If "Live" is togged on, update the parameter value immediately. If "Live" is toggled off, update the parameter value when the popup is closed. Color values are INT32 values in ARGB format.
WIDGET_GRAPH	chart	256	Displays a plot graph of a parameter's value over time.

WIDGET_SLIDER_HORIZONTAL (3)

Horizontal sliders are the default widgets used for range-bounded integer and floating point numbers when they are not used in an array. Sliders are not available for unbounded (null constraint) parameters.

Figure 3.25 shows an integer parameter with a range constraint bounded by (0, 200) and a WIDGET_SLIDER_HORIZONTAL hint.

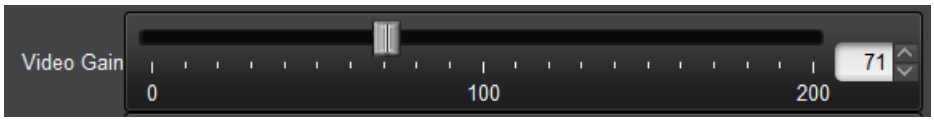


***Figure 3.25** - WIDGET_SLIDER_HORIZONTAL Hint*

WIDGET_SLIDER_VERTICAL (4)

Vertical sliders are the default widgets used for range-bounded integer and floating point numbers when they are used in an array. Sliders are not available for unbounded (null constraint) parameters.

Figure 3.26 shows an integer parameter with a range constraint bounded by (0, 994) and a WIDGET_SLIDER_VERTICAL hint.

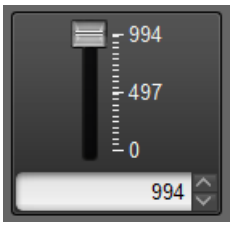


***Figure 3.26** - WIDGET_SLIDER_VERTICAL Hint*

WIDGET_SLIDER_HORIZONTAL_NO_LABEL (24)

***Figure 3.27** - WIDGET_SLIDER_HORIZONTAL_NO_LABEL Hint*

WIDGET_SLIDER_VERTICAL_NO_LABEL (25)

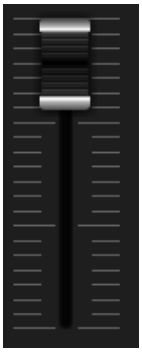


***Figure 3.28** - WIDGET_SLIDER_VERTICAL_NO_LABEL Hint*

WIDGET_VERTICAL_FADER (26)

This hint specifies that the number shall be displayed as a vertical fader bar. The user can adjust the level by dragging the handle of the fader up or down.

***Figure 3.29** - WIDGET_VERTICAL_FADER Hint*

WIDGET_TOUCH_WHEEL (27)

This hint specifies that the number shall be displayed as a touch wheel (or circular slider). The user grabs the dot on the circle and drags clockwise to increment the value and counter clockwise to decrement it. The touch wheel can be configured to take a specified number of revolutions to go from the minimum value to the maximum value and can also be configured to roll over to the minimum or maximum when the limits of the range are reached.

***Figure 3.30** - WIDGET_TOUCH_WHEEL Hint*

A parameter using a WIDGET_TOUCH_WHEEL widget may also specify additional configuration parameters in the config object of the parameter. (see "config Object" on page 4–46.)

Touch wheel properties are as follows:

Property	Type	Default	Description
w.loop	Boolean	False	Specifies if the touch wheel loops over values in the constraint.
w.thumbcolor	Integer	00000000	Color of the thumb when the control has focus.
w.zero	Integer	0	Zero point (in degrees) for the rotation.

WIDGET_PROGRESS_BAR (17)

This hint specifies that the number shall be displayed as a horizontal progress bar. For a range-bounded parameter, the progress bar displays the specified range (similar to a slider). For an unbounded parameter, the progress bar displays from 0 to 100%.

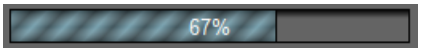


***Figure 3.31** = WIDGET_PROGRESS_BAR Hint*

WIDGET_SPINNER (5)

Spinner widgets provide a compact way to navigate a bounded integer or float parameter. Spinner widgets are the default widgets used for unbounded INT16 parameters. The spinner widget cannot be used with an unbounded floating point or INT32 parameter.

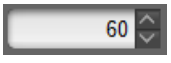


***Figure 3.32** - WIDGET_SPINNER Hint*

***Note**: The range of the parameter: abs(max – min) x precision cannot exceed the maximum size of a signed integer for sliders and spinners*

***Note**: To aid in touch screen environments, clicking and dragging a spinner up/down will increase/decrease its value.*

WIDGET_HEX_SPINNER (28)

Displays a spinner (entry field plus up/down arrows). Displays the value in Base 16.

***Figure 3.33** - WIDGET_HEX_SPINNER Hint: INT32 (left), and INT16 (right)*

Due to the lack of unsigned data types in OGP, hex spinners do not function properly in the following circumstances:

- An INT16 parameter with any values in the range of 0x8000 – 0xFFFF
- An INT32 parameter with any values in the range of 0x80000000 – 0xFFFFFFFF

To enable a spinner to function in the range from 0x80000 – 0xFFFF, use an INT32 parameter.

WIDGET_TEXTBOX (6)

This hint specifies that a simple text entry field should be used for a number. The information entered into the text field is forced to conform to the constraints provided by the parameter. This is the default widget used for unbounded floating point parameters.

***Figure 3.34** - WIDGET_TEXTBOX Hint*

WIDGET_IP_ADDRESS (14)

Displays an IPv4 Address format for a 32-bit integer. Only works with unbounded INT32 parameters.

***Figure 3.35** - WIDGET_IP_ADDRESS Hint*

WIDGET_AUDIO_METER (19)

This hint specifies that the number shall be displayed as a vertical audio meter. The number of red/yellow/green segments is fixed.

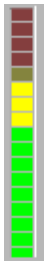


***Figure 3.36** - WIDGET_AUDIO_METER Hint*

WIDGET_TIMER (21)

This hint applies only to integer parameters with RANGE_STEP_CONSTRAINT constraints. The parameter is displayed as a label and counts down if minVal < 0 or up if minVal >= 0. Negative numbers are not displayed. The step size is used to specify the number of ticks-per-second to use and must be a number between 1 and 1000.

When the maximum or minimum values are reached, the timer will stop counting.

To initialize the counter to a specific value but not have it start counting:

- If the minimum value is negative and the parameter value is positive, the timer will display absolute(min) –value but will not count.
- If the minimum value is positive and the parameter value is negative, the timer will display absolute(value) but will not count.

The timer can be reset or synchronized by sending a REPORT_PARAM message with the new parameter value (typically “1”).

Examples:

- min=-600, max=0, step=1 (count from 10:00 to 0:00 showing each second).
- min=0, max=600, step=1 (count from 0:00 to 10:00 showing each second)
- min=0, max=1000,step=1000 (count from 0:00:000 to 0:01:000 showing each millisecond)

Figure 3.37 shows an INT_32 parameter with a WIDGET_TIMER hint, a precision of 1000, and a value of 13794088.

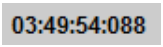


Figure 3.37 *- WIDGET_TIMER Hint*

WIDGET_ABSOLUTE_POSITIONER (29)

Positions a value in 2-D space.

- When used as an INT16, the 8 LSBs represent the X coordinate and the 8 MSBs represent the Y coordinate.
- When used as an INT32, the 16 LSBs represent the X coordinate and the 16 MSBs represent the Y coordinate.

A crosshair in a box is dragged to the absolute position of the value in 2-D space. The ratio of width to height is the ratio of xmax-xmin to ymax-ymin with the assumption that the screen pixels are square. Values are updated and sent to the device as the crosshair is dragged.

Figure 3.38 *- WIDGET_ABSOLUTE_POSITIONER Hint*

WIDGET_CROSSHAIR (30)

Positions a value in 2-D space.

- When used as an INT16, the 8 LSBs represent the X coordinate and the 8 MSBs represent the Y coordinate.
- When used as an INT32, the 16 LSBs represent the X coordinate and the 16 MSBs represent the Y coordinate.

A crosshair that snaps to the center when released makes changes in +/- X, +/- Y relative to the offset from the center. The ratio of width to height is the ratio of xmax-xmin to ymax-ymin. Values are updated and sent to the device as the crosshair is dragged.

Figure 3.39 *- WIDGET_CROSSHAIR Hint*

WIDGET_JOY_STICK(34)

Positions a value in 2-D space.

- When used as an INT16, the 8 LSBs represent the X coordinate and the 8 MSBs represent the Y coordinate.
- When used as an INT32, the 16 LSBs represent the X coordinate and the 16 MSBs represent the Y coordinate.

Displays a joystick and modifies the X,Y values as the joystick is dragged north, south, east, and west of the center.

Figure 3.40 *- WIDGET_JOY_STICK Hint*

WIDGET_COLOR_CHOOSER(23)

Display a color chooser as an element in the UI. Changes made to the color chooser are immediately sent to the device. Note that the color chooser provides control for Hue, Saturation, Lightness, but color values are INT32 values in ARGB format.

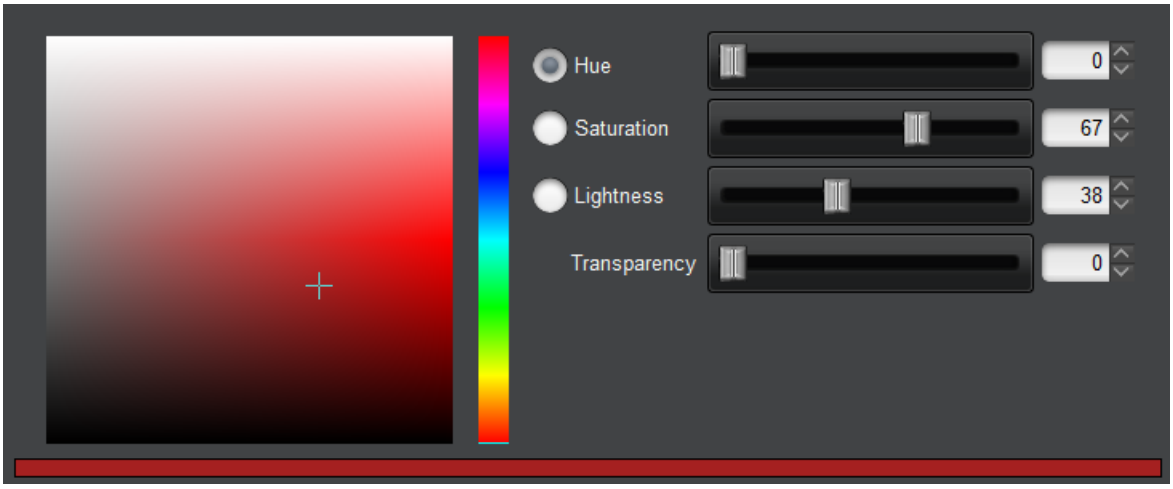


Figure 3.41 *- WIDGET_COLOR_CHOOSER Hint*

WIDGET_COLOR_CHOOSER_POPUP(33)

Display a combo box control showing the ‘current’ color. On click, show the color chooser.

- If “Live” is togged on, update the parameter value immediately.
- If “Live” is toggled off, update the parameter value when the popup is closed.

Color values are INT32 values in ARGB format.

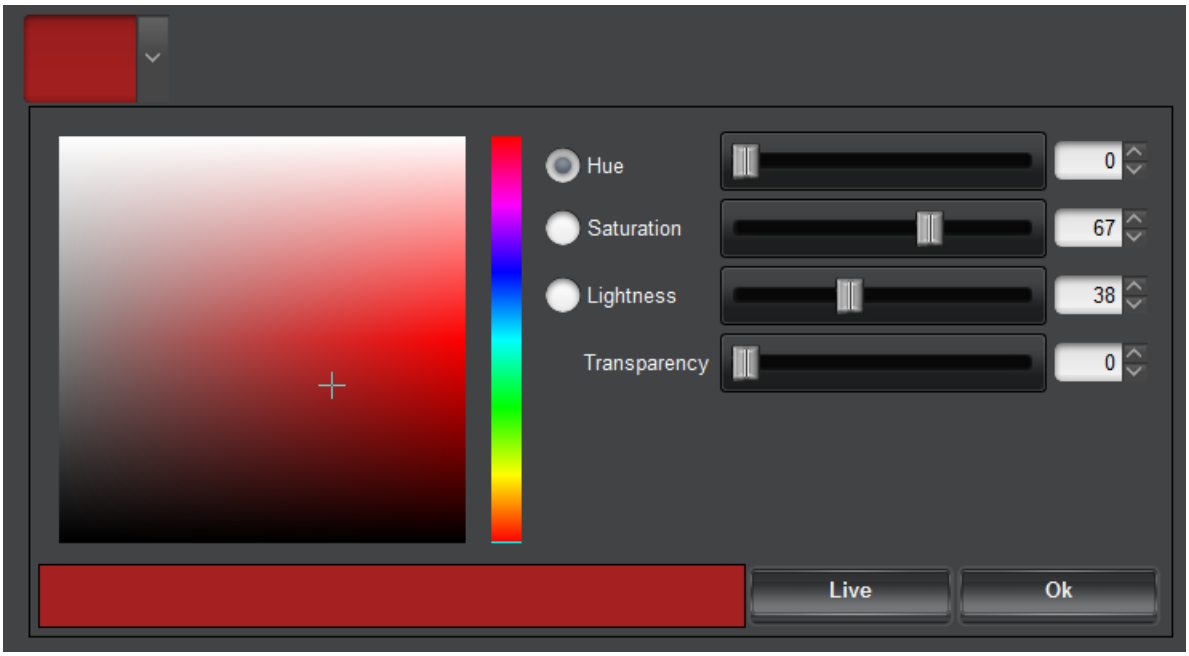


***Figure 3.42** - WIDGET_COLOR_CHOOSER_POPUP Hint*

WIDGET_GRAPH (256)

The graph widget provides a plot graph which tracks the value of a numeric parameter over time.

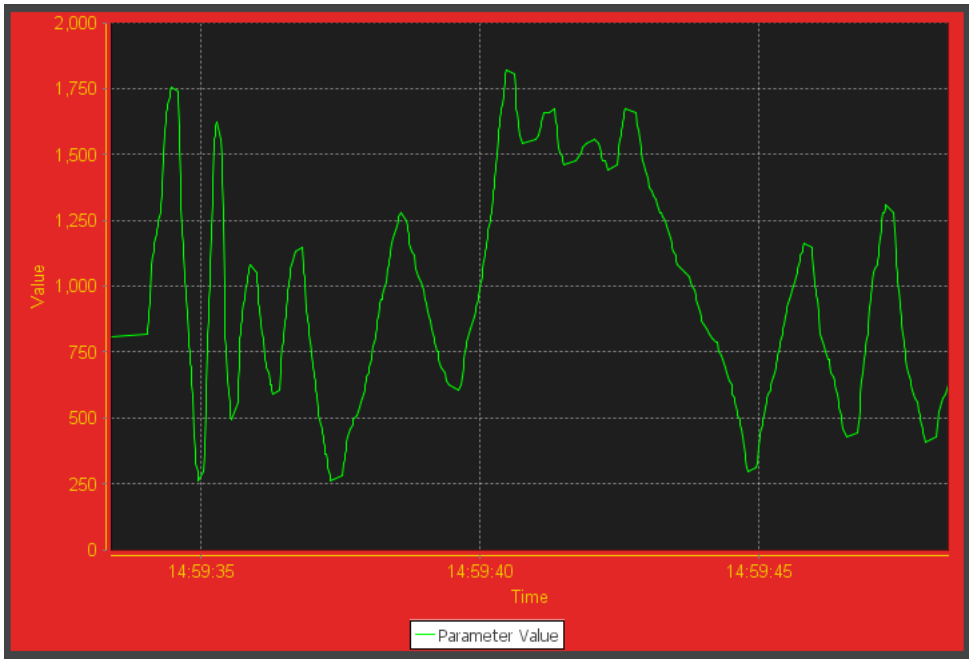


***Figure 3.43** - WIDGET_GRAPH Hint*

A parameter using a WIDGET_GRAPH widget may also specify additional configuration parameters in the config object of the parameter. (see “config Object” on page 4–46.)

Graph properties are as follows:

Property	Type	Default	Description
w.time	Integer		Sets the timescale of the plot. If set to 0, the timescale will adapt to display entire change history.
w.grid	String	black	Sets the color of the gridlines.
w.plotfg	String	red	Sets the color of the plot foreground.
w.plotbg	String		Sets the color of the plot background.
w.hidelegend	Boolean	false	• `true` – Legend is not shown • `false` – Legend is shown
w.hidex	Boolean	false	• `true` – X-axis scale is not shown • `false` – X-axis scale is shown
w.hidey	Boolean	false	• `true` – Y-axis scale is not shown • `false` – Y-axis scale is shown
w.autoadvance	Boolean	true	• `true` – graph will auto-update every 1 second • `false` – graph will only update upon parameter change.

Hints for String Parameters

The following widget hints may be used for String parameters (in addition to the separators and layout hints defined above). The last two hints apply only to a String parameter using the reserved objectID 0xFF01.

The following table describes the hints. Examples follow the table.

Widget Name	JSON Name	Value	Description
WIDGET_TEXT_ENTRY	text	3	Displays a normal text entry field. This is the default for editable String parameters.
WIDGET_PASSWORD	password	4	Displays an entry field for passwords (text entered in this field is obscured).
WIDGET_COMBO_ENTRY	combo	11	Displays an entry field together with a dropdown list of selectable items. This is applicable only with the STRING_CHOICE constraint
WIDGET_COLORED_DOT	dot	12	Displays a colored icon. The icon color is specified using a tag in the text string.
WIDGET_RICH_LABEL	html-text	13	Displays a read-only multi-line text field with HTML formatting.
WIDGET_MULTILINE_TEXT_ENTRY	multiline-text	14	Displays a multi-line text editor.
WIDGET_NAME_OVERRIDE_APPEND	n/a	0	Special hint only for objectID 0xFF01 – causes this string to be appended to the displayed product name.
WIDGET_NAME_OVERRIDE_REPLACE	n/a	1	Special hint only for objectID 0xFF01 – causes this string to replace the product name to be displayed.

Widget Name	JSON Name	Value	Description
WIDGET_COLOR_CHOOSER	color-picker	23	Put a color chooser as an element in the UI. Changes made to the color chooser are instantly sent to the device. Color values are hex string in ARGB format: AARRGGBB.
WIDGET_COLOR_CHOOSER_POPUP	color-picker-popup	33	Display a combo box control showing the 'current' color. On click, show the color chooser. If "Live" is togged on, update the parameter value immediately. If "Live" is toggled off, update the parameter value when the popup is closed. Color values are hex string in ARGB format: AARRGGBB.

WIDGET_TEXT_ENTRY (3)

This is a text entry field used to enter String values. This is the default widget used with editable String parameters. It is very important to correctly set the length of the String with this widget as the length affects the width of the text field. In DashBoard the value is sent to the device when the user hits 'Enter' or changes focus to a different control on the screen.

***Figure 3.44** WIDGET_TEXT_ENTRY Hint*

WIDGET_PASSWORD (4)

This is a text entry field used to enter passwords. When the device receives a set message for a parameter using this hint, a device could send an empty string back to the device to clear the password field. Text in the password field is sent when it has changed from the value reported from the device and the user hits "Enter" or moves focus to another control.

***Figure 3.45** - WIDGET_PASSWORD Hint*

WIDGET_COMBO_ENTRY (11)

This displays a text entry field along with a dropdown list. This option is available only for String parameters having a STRING_CHOICE constraint. The user may select an option from the dropdown list, or can type any value in the entry field. The text is sent to the device when a dropdown item is selected, when the user presses "Enter" or moves the focus after typing a value.

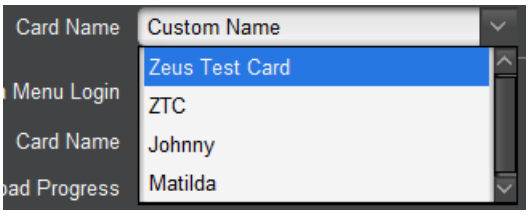


***Figure 3.46** - WIDGET_COMBO_ENTRY Hint*

WIDGET_COLORED_DOT (12)

This displays a colored icon. This should not be confused with Alarm parameters which have a similar appearance. The tag specifies the 24-bit RGB color index of the icon in hex, in the format <#RRGGBB>. If the string does not contain a valid color tag, the icon is drawn but not filled (i.e. background shows through).

***Figure 3.47** - WIDGET_COLORED_DOT Hint*

WIDGET_RICH_LABEL (13)

This displays a read-only multi-line text field, and formats the text according to the HTML formatting tags embedded in the text. Total string length including tags is limited to 250 bytes. The display uses html support within the Java display object, so the exact appearance of the label may vary depending on operating system and Java version.

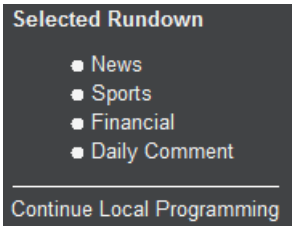


***Figure 3.48** - WIDGET_RICH_LABEL Hint*

WIDGET_MULTILINE_TEXT_ENTRY (14)

This displays a multi-line text entry field. The amount of data a user can input into the field is limited by the maximum length specified by the parameter. The size of the field is the same regardless of the maximum number of bytes the user is allowed to enter. If the parameter's value spans more lines than the number of rows represented by the text field, a vertical scrollbar is shown to allow the user to scroll. Text will be wrapped to avoid horizontal scrollbars.

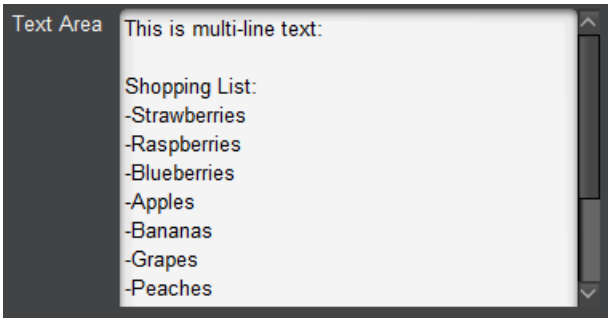


***Figure 3.49** - WIDGET_MULTILINE_TEXT_ENTRY Hint*

WIDGET_NAME_OVERRIDE_APPEND (0)

This is a special hint ONLY FOR OID 255.1 (0xFF01). This causes the value of the String parameter with OID 255.1 to be appended to the end of the device name in DashBoard.

Figure 3.50 shows the result of setting parameter 255.1 to " (XPF)" with a WIDGET_NAME_OVERRIDE_APPEND hint.

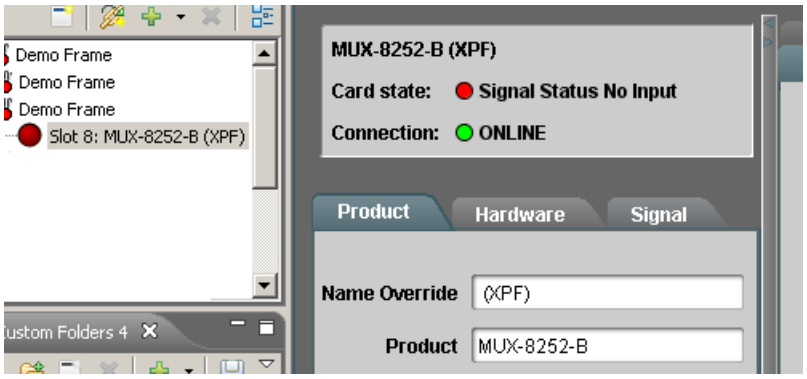


***Figure 3.50** - WIDGET_NAME_OVERRIDE_APPEND Hint*

WIDGET_NAME_OVERRIDE_REPLACE (1)

This is a special hint ONLY FOR OID 255.1 (0xFF01). This causes the value of the String parameter with OID 255.1 to be displayed as the device name instead of the product name (OID 0x0105) in DashBoard. This is the only supported method for changing a product name dynamically. Devices should never modify their base product name (OID 0x0105); DashBoard, DataSafe, and User Rights all depend on the base product name remaining fixed. Change of the product name is assumed to mean that the user has physically removed a card, and has replaced it with a different type of card.

Figure 3.51 shows the result of setting parameter 255.1 to "My Device Name" with a WIDGET_NAME_OVERRIDE_REPLACE hint.

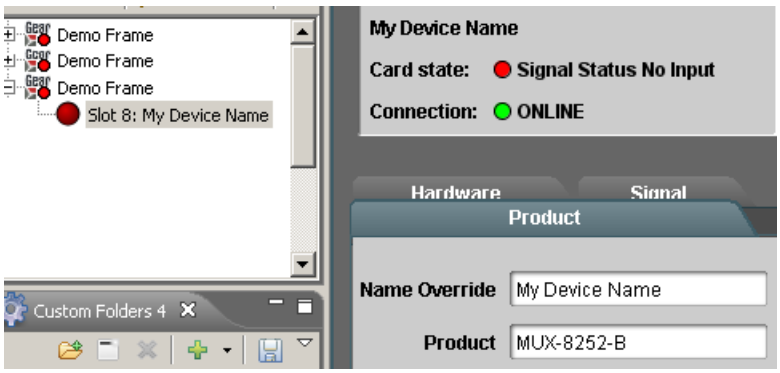


***Figure 3.51** - WIDGET_NAME_OVERRIDE_REPLACE*

Hints for STRUCT Types

Struct parameters may use the following widget types:

WIDGET_TABLE (36)

The table widget displays a line for each element in a STRUCT_ARRAY. Column headings are specified by the name property of each struct element. Each element of the struct is given a column in the table.

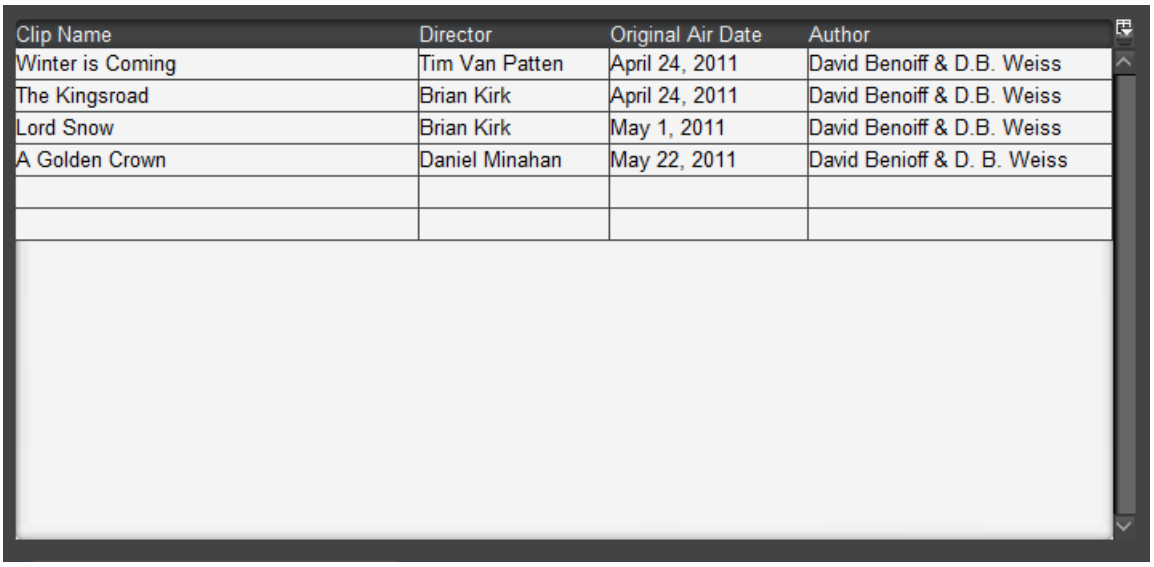


***Figure 3.52** - WIDGET_TABLE Hint*

A parameter using a WIDGET_TABLE widget may also specify additional configuration parameters in the config Widget table properties are as follows:

Property	Type	Default	Description
w.localselection	Boolean	false	• `true` – edits in the table row do not update backing parameter; changes in the backing parameter do not update the selected row(s). • `false` – backing parameter and table row changes track with each other.
w.scrollselection	Boolean	true	• `true` – auto scroll to the selected row • `false` – do not scroll to the selected row
w.reorder	Boolean	false	• `true` – allow drag to reorder values • `false` – do not allow drag to reorder values
w.rowstyleparam	String	none	OID of string array parameter providing style information (background, foreground, font, font size, etc.) for each row.
w.selectionparam	String	none	OID of integer parameter that will be populated with the index of a selected row.
w.rowaccessparam	String	none	OID of integer array parameter which determines read-only access for each row. (0 = read-only, 1 = read-write). If not specified, all rows are read-write.
w.rowheight	Number	automatic	Sets the row height. Specified in pixels.
w.colwidth.`n`	Number	automatic	Sets the width of the nth column. First column index is 0.
w.colminwidth.`n`	Number	automatic	Sets the minimum width of the nth column. First column index is 0.
w.hscroll	Boolean	false	• `true` – show horizontal scrollbar • `false` – do not show horizontal scrollbar
w.alwaysscroll	Boolean	false	• `true` – vertical scrollbar always shown • `false` – vertical scrollbar only shown only when required

Property	Type	Default	Description
w.hgrid	Boolean	true	• `true` – display horizontal grid lines • `false` – do not display horizontal grid lines
w.vgrid	Boolean	true	• `true` – display vertical grid lines • `false` – do not display vertical grid lines

Device Commands

`"cmds.resetclipcount"`. You can see that in this case the value argument updates two string parameters for the device and the clip value. `"_d_cmds.resetclipcount":`

```
{
   "oid": "cmds.resetclipcount",
   "name": "reset clip count",
   "readonly": false,
   "type": "STRUCT",
   "widget": "default",
   "value":
   [
       {
           "_d_deviceName":
           {
               "name": "device name",
               "readonly": false,
               "type": "STRING",
               "widget": "default",
               "maxlength": 0,
               "totallength": 0,
               "value": ""
           },
           "_d_clip":
           {
               "name": "clip",
               "readonly": false,
               "type": "STRING",
               "widget": "default",
               "maxlength": 0,
               "totallength": 0,
               "value": ""
           }
       }
   ],
   "constraint": {}
},
```

. . .

Data Types

OGP supports a number of parameter data types the supported types summarized in the table below. For OGP messaging, the OGP Type value is a numerical index to indicate the parameter's data type. For JSON messaging, the Data Type Name is used to indicate the parameter type.

Data Type Name	OGP type	data_size	Description
INT16	2	2	16-bit signed integer (INT16)
INT32	4	4	32-bit signed integer (INT32)
FLOAT32	6	4	32-bit IEEE single-precision floating point number
STRING	7	variable	null-terminated UTF-8 string data_size = maximum number of character data bytes
INT16_ARRAY	12	2 * len	array of 16-bit integers data_size = 2 * number of elements (total length of the array in bytes)
INT32_ARRAY	14	4 * len	array of 32-bit integers data_size = 4 * number of elements (total length of the array in bytes)
FLOAT_ARRAY32	16	4 * len	array of 32-bit floats data_size = 4 * number of elements (total length of the array in bytes)
STRING_ARRAY	17	variable	null-terminated UTF-8 strings precision = maximum string length for any element in the array data_size = maximum number of character data bytes
STRUCT	n/a	variable	User-defined data structure. (Dashboard 7.0+)
STRUCT_ARRAY	n/a	variable	Array of User-defined data structures. (DashBoard 7.0+)
BINARY_PARAM	18	variable	Array of binary data of type unknown to DashBoard.

Endianness

All numeric data encoding within OGP is in Big Endian format. Therefore, highest order bytes of multi-byte numeric values are transmitted first.

Number Encoding

Signed integer data types are binary encoded 2's complement numbers. Valid ranges for integer types are:

Response	Min	Max
UINT8	0	255
UINT16	0	65535
INT16	-32,768	32,767
INT32	-2,147,483,648	2,147,483,647

Floating point data types are encoded as 32-bit IEEE (single-precision) floating point numbers. This encoding is broken down as:

- Sign: 1 bit
- Exponent: 8 bits; Range -126 to +127
- Base: 23 bits
- Data size is the number of bytes occupied by the value.

String Encoding

All string data encoding within OGP is in UTF-8 format. Strings are preceded by a length count byte, and are followed by a null terminating byte. The length count is inclusive of the null terminator byte. Thus the maximum length for a string is 254 characters. This would be encoded as a length of 255 due to the null terminator byte.

Note carefully for binary OGP:

- The length count immediately preceding a string is inclusive of the null-termination byte. This applies to all strings transmitted as parameter values (OGP_PARAM_GET, OGP_PARAM_SET, and REPORT_PARAM), in menu names, and in choice/string/alarm constraints.
- There is one notable exception: the size and precision fields in the OGP_GET_DESCRIPTOR response. These values specify the number of characters, and DO NOT include the null-termination byte.
- The discrepancy between these two is unfortunate, but cannot be corrected, as it would affect the operation of existing openGear products.

An example to illustrate:

- Assume a string parameter declared with length = 20 in its descriptor
- Dashboard will permit up to 20 characters to be entered into the field by the user

When the value is transmitted to the card, up to 21 bytes may be sent: the 20 bytes from the user, followed by a null terminator byte. The length field in the OGP_PARAM_SET command can therefore be as high as 21.

External Data Objects

To support more complex interaction with the device than is possible with parameters, DashBoard includes a set of general data objects called External Objects. Each object is identified by a 2-byte objectID (like parameters), and contains a type identifier and object-specific data. External object OIDs can overlap with parameter OIDs. The range of OIDs from 0xFE00 to 0xFFFF is reserved for future use.

External objects include an object type to indicate the type of data they encapsulate. The supported object types are:

objtype	Description
1	Constraint
2	File
3	Image
4	OGLML or Index XML document

Constraint

Parameter constraint information can be taken outside of the parameter descriptor and moved into an external object. This is useful, for example if there is a choice constraint with a large number of options, or a common constraint is to be applied to multiple parameters. The constraint field in the parameter descriptor simply refers to an external object ID.

See also “Constraints” on page 3–4.

Any constraint type can be externalized except the external constraint type itself. An external constraint object can be shared by multiple parameters (the external object will be requested only once for all parameters which share the constraint). The object type of the external constraint must be 1, and the object data must be encoded in the same format as used for an embedded constraint.

An external object that is not object type 0x0001 will be treated as a NULL constraint (unconstrained). Just like constraints declared in the parameter descriptor, external constraints must have the same data type as the referring parameter.

Arbitrary File

Arbitrary binary data can be sent from the device to DashBoard as a file download. These files are requested by supplying an integer parameter with a WIDGET_FILE_DOWNLOAD widget hint and a choice constraint. The numeric value of the parameter must match the OID of an external object containing the file data to download. The string value of the choice constraint is used to supply a file name for the download. To upload the file data back to the device, the data must use the standard openGear file header information, as defined in the section "File Format for OGP Upload" on page 7–3.

Image

Images may be encapsulated within an External Object to be displayed in the device editor (via OGLML) or to be used to override its icon in DashBoard. The icon may include a status indicator or DashBoard can overlay a status indicator over the provided icon. Icons can be provided either by a URL or embedded directly in the external object.

Images must be formatted as JPEG, GIF, or PNG. Icons must be 16x16.

OGLML Descriptor or Index XML

DashBoard includes powerful feature for defining the on-screen layout of a device's configuration page in DashBoard. These configurations are defined in an OGLML Document (a variant of XML). These documents can be retrieved from a web server or sent to DashBoard in an external object Refer to the openGear Layouts document for details on OGLML.

JSON Reference

In This Chapter

This chapter describes the JSON Protocol. JSON messaging between devices and DashBoard is supported in DashBoard 7.0 and later. Subscriptions is supported in DashBoard 9.4 and later.

The following subjects are discussed:

JSON Messaging Overview

Device communication may be performed using JSON instead of the binary OGP protocol. This provides the advantage of readability of the messaging and data, as well as the ability to leverage standard software libraries for JSON parsing and serialization.

***Note**: JSON messaging is only available on TCP/IP connections; Binary OGP must be used for CANBus connections.*

Unlike OGP, JSON communication with DashBoard is an asynchronous protocol. It does not require the strict request/response paradigm of OGP. However, when DashBoard sends a parameter change to the device, the device is still expected to reply with the updated value so that DashBoard knows that the parameter change was successful or not. Also, the device must still fulfill the heartbeat requirements of OGP, otherwise DashBoard may close the connection.

Furthermore, when sending data via JSON, only changed data need be sent. Any unsent fields will retain their previous value, or default (if they have never been sent).

Netstrings Container

JSON messages are all wrapped in a modified netstring container. This provides a simple method for parsing and buffer management over the connection. For more information about netstrings, see http://en.wikipedia.org/wiki/Netstring.

Text-only messages are encoded in a standard netstrings container. DashBoard also supports a modified netstrings container to allow binary data to be appended to the message.

***Note**: Currently only the **eo message** (see "eo Message" on page 4–76) uses the mixed text/binary format. All other messages implement the standard text-only netstrings container.*

Syntax

For text-only messages:

```
text-length:text-data,
```

For mixed text and binary messages:

```
text-length,binary-length:text-data binary-data,
```

Fields

Field	Description
text-length	The number of 8-bit bytes in the text data field. text-length is encoded as an ASCII string.
binary-length	The number of 8-bit bytes in the binary data field encoded as an ASCII string. binary-length is encoded as an ASCII string.
:	Hex 0x3a. Delimiter between the length and data fields.
text-data	Text Message payload. UTF-8 encoded.
binary-data	Binary data payload, presented as raw binary data.
,	Hex 0x2c. Delimiter between text-length and binary-length. End of Message delimiter.

Examples

The text `hello world!` is encoded as:

```
12:hello world!,
```

The byte sequence is:

```
<31 32 3a 68 65 6c 6c 6f 20 77 6f 72 6c 64 21 2c>
```

An empty message is encoded as:

```
0:,
```

The byte sequence is:

```
<30 3a 2c>
```

A message containing the text `hello world!` followed by the binary data `c0cac01a` is encoded as follows:

```
  1  2  ,  4  :  h  e  l  l  o     w  o  r  l  d  !  □  □  □  □  ,
<31 32 2c 34 3a 68 65 6c 6c 6f 20 77 6f 72 6c 64 21 c0 ca c0 1a 2c>
```

JSON Message Format

The message payload is described later in this chapter, and depends upon the message type field.

Syntax

```
{
   "type": messageType,
   "slot": slotID,
   "payload" : {
      MessagePayload
   }
}
```

Fields

Attribute	Description	Equivalent OGP Field	Description
Type	String	msgType	Message Type
Slot	Number	dest	Destination Address
payload	Object	content	Message Payload

JSON Connection

JSON connects via a TCP connection, normally on port 5254. The connection port may be changed to any other value, but must be published as part of the device discovery. The device must publish its equipmentype as `opengear-json` during discovery.

Initiating Connection to JSON Devices

Connection Handshake

When DashBoard opens a TCP/IP connection, it sends a handshake request message (see page 4–79) to determine if the device will accept the connection. The device shall respond with a handshake response message (see page 4–80) to allow or deny the connection.

Device Query

Upon successful handshake, DashBoard will send a device-request message (see page 4–75) to query all parameters and menus associated with the device. The device shall respond with a device message (see page 4–74). This response includes a complete device object (see page 4–56), publishing all parameters, constraints and menus for the device. If your device supports OGP minimal mode, the response would only include parameters from the minimal mode set that you select. For more information about OGP minimal mode (see page 4–4).

OGP Minimal Mode Support

You may wish to support OGP Minimal Mode on your device if you want to reduce the amount of parameter updates that DashBoard receives from your device(s). This can be particularly useful if your connected devices send a substantial amount of parameter updates (such as a Graphite with Rave audio mixers) or if your DashBoard instance has multiple device panels that display upon startup.

DashBoard version 8.7.1 and later supports any devices that implement OGP Minimal Mode to only send DashBoard updates from a minimal set of parameters, until the devices' user interfaces open. The device determines which parameters are included for the minimal set of parameters. This could include a subset of alarms, device reserved OIDs (such as display names), or any other necessary parameter updates. The device must always provide Keep Alive, or heartbeat, messages.When a device user interface opens or if it is already open in DashBoard, then DashBoard requests full updates from the device.

DashBoard device panels that have been offloaded by the memory manager are treated the same as a closed device user interface.

Additionally, the DashBoard Proxy server enables OGP Minimal Mode support by default for any devices it shares.

Note: DashBoard Proxy Server Limitations

When a device is shared through the DashBoard proxy server, direct communication with the device from the DashBoard client hosting the proxy server will always be set to full updates regardless of the state of interfaces of the device.

For more information on minimal mode see:

- "Overview of Minimal Mode Workflow" on page 4–4
- "What happens if the device does not support minimal mode?" on page 4–5
- "How does this impact DashBoard Version 8.7.0 and earlier?" on page 4–5

Overview of Minimal Mode Workflow

To implement OGP Minimal Mode both DashBoard and the device must indicate support for minimal mode during the handshake process. Support for OGP minimal mode is currently implemented through the DashBoard Proxy Server.

First, DashBoard initiates a handshake message to the device, and indicates support for minimal mode as part of its payload. If the device also supports minimal mode, in the device reply it can indicate support for minimal mode with the device descriptor. Once the device has indicated its support for minimal mode, DashBoard expects to only receive device updates from the minimal set. If a device UI is opened in DashBoard or if a device UI is already open, then DashBoard sends a device-request for updates from the full parameter set. When all device UIs are closed or offloaded, then after 10 seconds DashBoard sends device-request message for updates from the minimal parameter set.

The OGP minimal mode workflow diagram is shown in **Figure 4.1**.

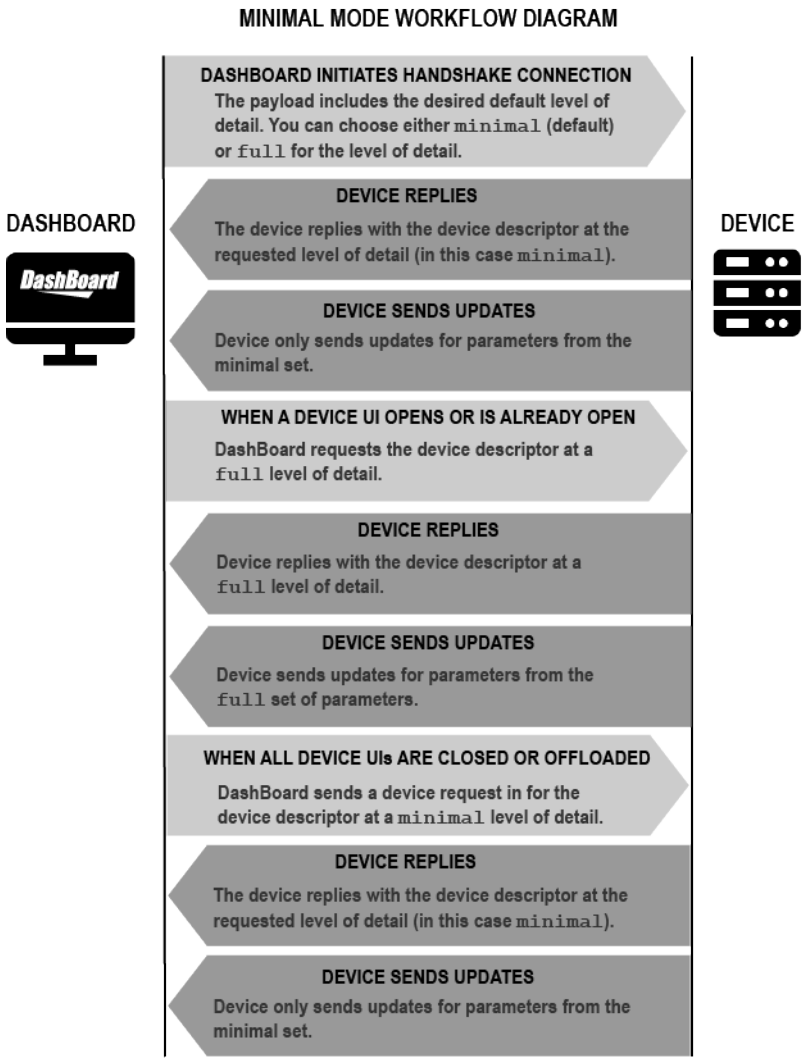


Figure 4.1 *Device with Support for Minimal Mode*

What happens if the device does not support minimal mode?

If your device does not support minimal mode, your device will continue to send full updates to DashBoard.

How does this impact DashBoard Version 8.7.0 and earlier?

If you're using an earlier version of DashBoard (before 8.7.1), your devices will **not** be impacted by this change, and your devices will continue to send full updates.

How to Support Minimal Mode

In this section you will learn how to implement support for minimal mode in the DashBoard Client and OGP device.

1. Verify that the DashBoard Client Supports Minimal Mode

To verify that the DashBoard client supports minimal mode, you'll need to verify that the level of `"detail"` is present in the device `handshake`. The level of `"detail"` should be set to `"minimal"` for minimal mode.

✶ **Note**: If the level of "detail" is set to "full" then devices will continue to send full updates, rather than only the parameters that are part of the defined minimal set.

"handshake" Syntax

```
{
   "type": "handshake",
   "slot": slotID,
   "payload" : {
      "trusted" : trusted flag,
      "password" : "device password",
      "force" : force connection flag,
      "build" : "major.minor.micro YYYY-MM-DD HH:MM",
      "detail" : "minimal",
      }
}
```

2. Collect subscription list from "device-request"

The DashBoard client collects the OIDs that are part of the minimal set from all open panels. The "device-request" message. Make sure to include the level of "detail" and set it to "minimal" to provide the minimal set of OIDs, as shown below:

"device-request" Message Syntax

```
{
   "payload":{
      "detail":"minimal",
      },
   "slot":0,
   "type":"device-request"
}
```

Explanation:

In response to the "device-request" message, the device sends a "device" message response. This response is much smaller than before, because it only includes a subset of the device's parameters (or the defined minimal set), not the full list.

3. Populate the "device" Message

In response to the DashBoard Client's "device-request" message, the device sends a "device" message response.

a) Add the detail field to the device message

Update the **"device"** message, to indicate the **"minimal"** level of **"detail"** in the **"payload"** object.

"device" Message Syntax

```
{
   "type": "device",
   "slot": slotID,
   "payload" : {
       "slot": slot id,
       "detail": "minimal" | "full",
       "menu-groups": { menu-groups object },
       "params": { params object },
       "multi-set-enabled": multi-set-flag
   }
}
```

b) Add the full descriptor of items from the minimal set and subscriptions set

In the params section of the payload of the `"device"` message, only include the descriptors of items from the minimal set.

c) Flag every item in minimal set with "minimalset":true

Add the new **`"minimalset"`** flag to the param Object in device's minimal set of parameters. It must be set to **`"true"`**.

param Object Syntax

```
"_d_OID" : {
   "oid": "oid",
   "parent": "parent oid and element index",
   "name": "parameter name",
   "type": "parameter type",
   "readonly": read-only flag,
   "widget": "parameter widget",
   "precision": parameter precision,
   "maxlength": maximum string length,
   "totallength": maximum total length of string array,
   "constraint": { constraint object },
   "config": { config object },
   "value": parameter value,
   "minimalset": Boolean
}
```

Explanation:

Every parameter that is flagged as part of the **`"minimalset"`**, will always remain active and updates to it will always be required.

Subscriptions Support for OGP JSON devices

The DashBoard platform provides support for a new subscription protocol that device developers who are part of Ross Video's openGear partner program can implement on OGP JSON devices to get the most out of the DashBoard Connect™ ecosystem.

Once a device developer adds support for the subscription protocol to an OGP device, then any DashBoard device panel can specify which parameter(s) it would like to subscribe to from that OGP device. DashBoard device panels that only subscribe to the required parameters will benefit from the reduced message size and minimal communication between the DashBoard Client and the OGP device.

Using subscriptions is recommended to:

- Optimize memory usage and communication
- Increase panel efficiency
- Load device panels faster

✶ **Important**: Subscriptions is a protocol extension to OGP JSON, and does not support OGP binary devices.

Software Version

- The subscription protocol is only available in DashBoard v9.4 and later.
- The prerequisite Minimal Mode protocol must also be supported.

Assumptions

For the purposes of this document, a device panel refers to any DashBoard CustomPanel with a device context that is running in the DashBoard client, or a device user interface (UI) page that is served up by a device directly.

For More Information on...

- About openGear Partners, see the opengear.tv website
- More important terms, see the "**Subscriptions Glossary**"
- Accessing the DashBoard built-in help, see the DashBoard top menu and navigate to **Help > Help Contents**.

Table of Contents

How Subscriptions Work

Subscriptions is a protocol built on top of minimal mode, and as such requires that support for minimal mode has already been implemented before implementing support for subscriptions.

But wait, what is Minimal Mode?

Minimal Mode allows you to specify which device parameters updates each device panel subscribes to. When you provide support for Minimal Mode in your device communication, it greatly decreases the response times for panels that require a large amount of parameter updates. Instead of the DashBoard panel receiving all of a device's parameter updates in `full`, the panel will only receive the `minimal` set of parameter updates that you have defined.

You will find more details on how to support Minimal Mode in the "**Before You Begin**" section.

Once I support Minimal Mode, how can I support Subscriptions?

To take advantage of subscriptions, all device panels need to provide their list of subscriptions using one of the ways in which subscriptions are declared in the panel's OGLML document structure.

To add subscriptions support in the OGLML Document Structure

1. You must add `subscriptions="true"` in the layout container or device context to support subscriptions.
2. You must add `<subscription oids=""/>` to support subscriptions.

✶ **Important**: You can use the device context in the topmost container, as shown in this example, or any other device context in the OGLML document structure. The **<subscription oids=""/>** tag does not need to be nested within **<meta>** tags, but it is recommended.

You can see an example of the correct OGLML document syntax below:

Example 1: **OGLML Document Syntax for a Layout Container**
Where the syntax shows an absolute container <abs/> with an opengear **contexttype** attribute. Note that any layout or container type attribute could be used.

```
<abs contexttype="opengear" id="_top" keepalive="false" objectid="MyUltritouch..."
subscriptions="true">
   <meta>
      <subscription oids="oid1, oid2, oid3*"/>
   </meta>
</abs>
```

Example 2: OGLML Document Syntax for a Device Context

Where the syntax shows a Subscriptions Panel with a device <context/> Tag. Using this format makes it possible to organize a panel that has multiple device contexts (or device sources).

```
<context contexttype="opengear" objectid="DeviceID..." subscriptions="true">
   <meta>
      <subscription oids="oid1, oid2, oid3*"/>
   </meta>
</context>
```

For More Information on...

- OGLML document structure, see: ***DashBoard CustomPanel Development Guide (8351DR-007)***
- Device contexts, see: "**Subscriptions Glossary**"

Ultritouch Panel Example with Subscriptions Support

```
<abs contexttype="opengear" id="_top" keepalive="false" objectid="My_Ultritouch"
objecttype="MyUltritouchDevice" subscriptions="true">
    <meta>
       <subscription oids="db.uiselectormapping, db.touch*, device.memoryusage"/>
    </meta>
    <table height="112" left="103" oid="0xFF01" top="102" width="295">
       <tr>
<label anchor="east" fill="none" insets="0,0,0,5" name="Display Name"
weightx="0.0"/>
<param anchor="west" expand="true" fill="both" mid="-1" oid="0xFF01"
showlabel="false" weightx="1.0" weighty="1.0"/>
      </tr>
    </table>
    <table height="68" left="102" oid="device.memoryusage" top="247" width="298">
       <tr>
                  <label anchor="east" fill="none" insets="0,0,0,5" name="Device
```

```
Memory Usage" weightx="0.0"/>
                  <param anchor="west" expand="true" fill="both" mid="-1"
oid="device.memoryusage" showlabel="false" weightx="1.0" weighty="1.0"/>
            </tr>
    </table>
    <table height="86" left="103" oid="db.touch.type" top="380" width="301">
            <tr>
                  <label anchor="east" fill="none" insets="0,0,0,5" name="Type"
weightx="0.0"/>
                  <param anchor="west" expand="true" fill="both" mid="-1"
oid="db.touch.type" showlabel="false" weightx="1.0" weighty="1.0"/>
        </tr>
    </table>
    <table height="79" left="105" oid="db.touch.serial" top="523" width="303">
      <tr>
                  <label anchor="east" fill="none" insets="0,0,0,5" name="Serial"
weightx="0.0"/>
                  <param anchor="west" expand="true" fill="both" mid="-1"
oid="db.touch.serial" showlabel="false" weightx="1.0" weighty="1.0"/>
      </tr>
    </table>
</abs>
```

Explanation

The example above shows the OGLML document structure of an Ultritouch 4RU panel with subscriptions support. If you look at the structure, the subscriptions tag and subscription OIDs tag are highlighted in red in the device context.

As you can see the subscription tag lists the OIDs of some of the parameters used in this OGLML document:

```
<meta>
       <subscription oids=" db.uiselectormapping, db.touch*, device.memoryusage"/>
<meta/>
```

To make sure all the device parameters referenced in this OGLML document are updated and have their full description available, you need to add them to the subscription list(s) in the OGLML document. In the example above we are using the device's UI mapping, type, and memory usage.

The subscription tag above shows that the OIDs of these parameters are in subscription list:

```
<subscription oids="db.uiselectormapping, db.touch*, device.memoryusage"/>
```

Even though device's display name is used in the panel, explicit subscription to it is not required. This is because device's display name is part of reserved OIDs, which is considered a parameter in device's minimal set. Subscription to items in device's minimal set is done by default. If a parameter that is not part of a device's minimal set is used in the OGLML document file while its oid is not in any subscription tag, then the value for that parameter may not be accessible.

Each subscription tag will trigger a request to the device. We recommend combining all subscription OIDs into a single tag for a device context in an OGLML document, as shown above. The **<meta/>** tags are not required, but the **<subscription oids=""/>** tag must be in the device context.

In this case a wildcard is not required, but is used for **"db.touch*"** to ensure that any other OIDs that begin with the prefix **db.touch** will automatically be included. For example, **"db.touchserial"** is included.

To support subscriptions, in addition to updating the device panel's OGLML document structure, as shown above, the device itself must also support subscriptions.

Overview of Subscriptions

The steps to fulfill the requirements for subscriptions support are summarized below:

1. Each DashBoard device panel must indicate its own subscription list.
2. The DashBoard client collects the subscription list of each active panel and submits the entire subscription list to the device.
3. On the device side, the device must provide a list of all the parameters in a subscription list in addition to parameters in its minimal set.
4. Once implemented, the device should only send updates to the values of the items that the device panel has subscribed to in its subscription list.
5. When a panel is closed in DashBoard, the DashBoard client will send the list of items that are no longer in use to the device to unsubscribe from the updates.
6. The device uses the updated list to filter out which updates are required by the DashBoard client.

✶ **Note:** If the device sends updates to parameters that a DashBoard client has not subscribed to, the message will be ignored by the DashBoard client.

FAQ

How does message handling differ with subscriptions?

• If a DashBoard client supports subscriptions, it only requires change updates from the specified parameters that DashBoard subscribes to. This increases the performance significantly, particularly for devices with many parameters. Previously, devices without subscriptions were required to broadcast all parameter changes that occur via other clients via an asynchronous parameter reporting message to all connected DashBoard clients.

• With support for subscriptions, the device message is not required to provide the descriptors for all parameters. The only parameter descriptors that are required are those from the minimal set and the list of subscription parameters.

What if a subscriptions tag is not included?

Even if a device supports subscriptions, if a subscriptions tag with the attribute set to **`subscriptions=true`** is *not* included in the DashBoard panel's device context, or if the attribute is set to **`subscriptions=false`**, then it is assumed to be a legacy panel that receives *all* parameter updates. In this case it does not support subscriptions.

Essentially, if subscriptions is not supported, or the attribute is undefined or set to false, then subscriptions are not enabled for the OGLML document and the full device descriptor is requested.

What if a list of the required OIDs is not provided?

✶ **Important**: If you add **`subscriptions=true`** but fail to provide a subscriptions list of the parameters you wish to subscribe to, using the **`<subscription oids=""/>`** list, then you will only be subscribed to the minimal set.

How can I update my legacy panels (created before DashBoard v9.4)?

Legacy panels do not include the **`subscriptions="true"`** tag, and will be assumed to be false (**`subscriptions="false"`**). You must add a **`subscriptions="true"`** tag to indicate support for subscriptions. Updating your legacy panel will only produce the desired result when working with a device that supports minimal mode and subscriptions protocol.

For More Information on...

- OGLML document structure, see: ***DashBoard CustomPanel Development Guide (8351DR-007)***
- Subscriptions, see: ***openGear Software Development Guide*** **(8200DR-006)**
- Wildcard usage, see "**About Wildcards**" section in this guide.

Before You Begin

In this section, you can verify that you meet the software and device requirements. If you wish to use the DashBoard Proxy Server, additional requirements are also listed below. You will then learn about Minimal Mode requirements, and how to enable subscriptions features in the DashBoard Application.

Software and Device Requirements

Ensure that your software and devices meet the requirements listed below.

Software

- DashBoard Version 9.4 and later supports subscriptions

✶ **Important**: Minimal Mode support must be implemented before adding subscriptions support. For more information see, the ***Minimal Mode Application Note***.

Devices

- Devices must support OGP JSON protocol or devices must be shared through the DashBoard Proxy Server. Refer to the section "**DashBoard Proxy Server Requirements**" below.

✶ **Important**: Devices that use OGP binary must switch to use OGP JSON before implementing subscriptions.

DashBoard Proxy Server Requirements

When a DashBoard Client that supports subscription connects to a device through the DashBoard proxy server, the communication between DashBoard client and the proxy server can use subscriptions if the proxy server also supports subscriptions. Both the DashBoard client and DashBoard Proxy Server support subscriptions in DashBoard v9.4 and later. It is important to note that even if the device supports subscriptions, the proxy server connected to the device will always subscribe to all of device parameters. The proxy server can communicate with any device that supports either OGP JSON or OGP binary. The Proxy Server automatically converts any devices to OGP-JSON with subscription support regardless of which version of the protocol the device uses natively for its communication.

Implementing Minimal Mode

Minimal Mode is supported and enabled by default in DashBoard Version 8.7.1 and later.

✶ The protocol changes do not impact CustomPanels at all, and do not require any changes on the CustomPanel side.

When you provide support for minimal mode in your device communication, it takes the first step to make your device communication more efficient. With minimal mode, any connected devices will send only the basic required communication and only parameter updates from a defined minimal set of OIDs. Implementing minimal mode greatly reduces the memory consumption in DashBoard when a device is not actively in use. The benefits will be widely felt for any users with multiple connected devices that are on standby, and who wish to increase performance. Instead of DashBoard receiving all of a device's parameter updates in `full`, the panel will only receive the `minimal` set of parameter updates that you have defined.

You must follow the requirements listed here to enable it and follow the ***openGear Software Development Guide*** instructions to update your device communication to support minimal mode (details can be found under **JSON Reference > OGP Minimal Mode Support**):

- Minimal Mode must be enabled (default behavior)
- Device and Client messages must indicate support for the level of `detail` as `minimal` or `full`.

How to Implement Subscriptions

In this section you will learn how to implement subscriptions support in the DashBoard Client and OGP device.

✶ **Note**: This document assumes that support for minimal mode has already been implemented and that you are using a version of DashBoard that supports subscriptions (DashBoard version 9.4 or later).

1. Verify that the DashBoard Client Supports Subscriptions

To verify that the DashBoard client supports subscriptions, you'll need to verify that the `"subscriptions"` flag and level of `"detail"` are present in the device `"payload"`.

Verify that the subscriptions flag is present in the device `"handshake"` message and that the `"subscriptions"` flag is set to `"true"`. The level of `"detail"` should be set to `"minimal"` if minimal mode has been implemented.

`"handshake"` Syntax

```
{
   "type": "handshake",
   "slot": slotID,
   "payload" : {
      "trusted" : trusted flag,
      "password" : "device password",
      "force" : force connection flag,
      "build" : "major.minor.micro YYYY-MM-DD HH:MM",
      "detail" : "minimal" | "full",
      "subscriptions" : true | false
      }
}
```

`"handshake"` Example

```
{
   "type": "handshake",
   "slot": slotID,
   "payload" : {
      "trusted" : trusted flag,
      "password" : "1232khfsd6I9f3",
      "force" : force connection flag,
      "build" : "major.minor.micro 2022-01-07 07:10",
      "detail" : "minimal",
      "subscriptions" : true
      }
}
```

Field	Type	Required	Description
detail	String	*Yes - required for subscriptions.	This can be set to **minimal** | **full**, where: • **minimal** - Receives the minimal set. • **full** - Receives all. (This is the default for legacy panels)
subscriptions	Boolean	Yes	This can be set to **true** or **false**, where: • **true**- Indicates that this connection has support for subscriptions. • **false** - Indicates that this connection does NOT support subscriptions.

2. Collect subscription list from "device-request"

The DashBoard client collects the subscription list from all open panels and adds them to the subscription list in the `"device-request"` message. The `"device-request"` message includes the level of `"detail"` set to `"subscription"` and the `"subscription"` field provides a list of parameter OIDs that you wish to subscribe to:

"device-request" Message Syntax

```
{
   "payload":{
      "detail":"subscription",
      "subscription":[
         "oid_id1",
         "oid_id2",
      ]
   },
   "slot":0,
   "type":"device-request"
}
```

Field	Type	Required	Description
detail	String	For subscriptions, • **Yes** - it is required that you set it to **subscription.** If you are not supporting subscription, • **No** - The default is set to **full**.	This can be set to **subscription** | **minimal** | **full**, where: • **subscription** - Receives the minimal set and subscriptions list. *This is required for the subscriptions list! • **minimal** - Receives the minimal set. • **full** - Receives all. (This is the default for legacy panels)
subscription	Array of String	For subscriptions, • **Yes** - it is required that you include a list of parameters. If you are not supporting subscription, • **No** - The default is set to **full**.	List of parameter OIDs for subscription. • **Note**: You must set the **"detail"** field to **"subscription"** if you are including a subscription list.

"device-request" Message Example

```
{
   "payload":{
      "detail":"subscription",
      "subscription":[
         "db.touch.type",
         "db.serial*"
      ]
   },
   "slot":0,
   "type":"device-request"
}
```

Explanation:

In response to the `"device-request"` message, the device sends a `"device"` message response. This response is much smaller than before, because it only includes a subset of the device's parameters in the full description (the client subscription list and the device minimal set). When subscriptions is not supported, the message size is significantly larger, because it includes a full description of all the device's parameters.

3. Populate the device Message

In response to the DashBoard Client's `"device-request"` message, the device sends a `"device"` message response. Follow the procedures below to populate the `"device"` message. This includes adding the subscription flag to the device message, adding the descriptor of items in the minimal and subscription set, and flagging items in the minimal set.

a) Add the subscriptions flag to the device message

Update the **"device"** message, to include the **"subscriptions"** flag to `true` in the **"payload"** object, and set the level of **"detail"**. You must also include the list of **"subscription"** OIDs that were returned.

"device" Message Syntax

```
{
   "type": "device",
   "slot": slotID,
   "payload" : {
       "slot": slot id,
       "subscriptions": true,
       "detail": "minimal" | "full" | "subscription",
       "subscription": [
          "OID_1″,
          "OID_2″,
          "OID_3″
       ],
       "menu-groups": { menu-groups object },
       "params": { params object },
       "multi-set-enabled": multi-set-flag
   }
}
```

Explanation:

You must add the `"subscriptions": true` flag to indicate that your device supports subscriptions. Alternatively, if the flag is not present in the payload, this indicates that the device does not support subscriptions (and ensures that DashBoard panels that were created before subscriptions was released in DashBoard v9.4 and earlier will continue to function as previously).

b) Add the full descriptor of items from the minimal set and subscriptions set

In the params section of the payload of the `"device"` message, only include the descriptors of items from the minimal set and subscription list.

c) Flag every item in minimal set with "minimalset":true

Add the new `"minimalset"` flag to the param Object in device's minimal set of parameters.

param Object Syntax

```
"_d_OID" : {
   "oid": "oid",
   "parent": "parent oid and element index",
   "name": "parameter name",
   "type": "parameter type",
   "readonly": read-only flag,
   "widget": "parameter widget",
   "precision": parameter precision,
```

```
   "maxlength": maximum string length,
   "totallength": maximum total length of string array,
   "constraint": { constraint object },
   "config": { config object },
   "value": parameter value,
   "minimalset": Boolean
}
```

Field	Type	Required	Description
minimalset	Boolean	No	A flag to indicate a parameter is part of a device's minimal set. If present and **true**, the parameter is in the device's minimal set. Default is **false**.
parent	string	No. This attribute is required if the device is responding to a panel's subscription request for an element inside of a Struct, such as a **STRUCT_ARRAY**.	This attribute tells the DashBoard Client which oid this parameter is a child of. The string value is the oid of the parent oid. **Note**: For Struct types, such as a **STRUCT_ARRAY**, the string value must also include the **index** of the parent element in the struct.

Explanation:

Every parameter that is flagged as part of the **`"minimalset"`**, will always remain active and updates to it will always be required.

4. Provide support for the "update-subscriptions" message

a) Support DashBoard Client "update-subscriptions" message

After the device-request message is sent, every time a new device panel opens or closes, the DashBoard client collects the subscriptions and sends an `"update-subscriptions"` message to the device. This message allows the client to `"subscribe"` to or `"unsubscribe"` from the subscriptions list (of parameter OIDs).

Syntax

```
"type": "update-subscriptions",
"slot": slotID
"exe-id": "unique ID string",
"payload":
{
   "subscribe": [
```

```
        "ID_1",
        "ID_2*",
        …
    ],
    "unsubscribe": [
        "id3",
        "id4",
        …
    ]
}
```

Field	Type	Required	Description
type	String	Yes	Set to `update-subscription`.
slot	Number	Yes	Destination slot for this message.
exe-id	String	Optional* **Note:** It is required if the DashBoard client is expecting a response.	The unique execution ID that DashBoard client adds to the header of the update-subscription message when it requests new subscriptions. When this ID is present in the request, the device adds the ID to the header of the params message response. This allows DashBoard to verify that it has received the response to a specific request.
payload	object	Yes	Provides the lists of parameter OIDs to subscribe or unsubscribe to.
subscribe	Array of String	Yes	Subscribe to the provided list of parameter OIDs.
unsubscribe	Array of String	Yes	Unsubscribe to the provided list of parameter OIDs.

b) Update the Device "params" Message

When an update subscription message has a `"subscribe"` section with content, the device is required to reply with a `"params"` message that includes the descriptors of the items that the DashBoard client is subscribed. It should also contain the same `"exe-id"` if the unique ID is present in the update subscription request.

Syntax

```
{
        "type": "params",
```

```
        "slot": slotID,
        "exe-id": "execution ID",
        "payload" : {
           "_d_OID1": { param object 1 },
           "_d_OID2": { param object 2 },
           "_d_OID3": { param object 3 }
        }
},
```

Field	Type	Required	Description
type	String	Yes	Set to params.
slot	Number	Yes	Destination slot for this message.
exe-id	String	Yes - Only required if an `exe-id` exists in the update subscription message request.	This field is only required if the `exe-id` exists in the update subscription message request. Then the device response should include the `exe-id` in the header of the params message. **Note**: The ID is an arbitrary string provided by the requesting client. It is not guaranteed to be globally unique.
payload	Object	Yes	The payload will include the parameter descriptor information for every requested parameter.
_d_OID	Param Object (descriptor)	No	Parameter descriptor for each parameter object.

Explanation:

When a device UI panel is opened in DashBoard, the DashBoard client will send the list of items in the panel's subscription list to the device and `"subscribe"` to the updates. When a panel is closed in DashBoard, the DashBoard client will send the list of items that are no longer in use to the device and `"unsubscribe"` from the updates. The device should use this information to update its local subscription list for this client and also to determine which updates are no longer required by the DashBoard client.

5. Populate the DashBoard Client's Basic Param Info Request

Follow the procedures below to populate the "**basic-param-info-request**" message. The basic param info request, or BPI request, is a message that is sent from the DashBoard client side.

a) Update the "basic-param-info-request"

The **basic-param-info-request** defines a request to get the name, type, oid, template, and length for the children of a particular OID. This is used to build parameter trees, anywhere you can see a list of all the devices and their parameters in the DashBoard CustomPanel. Whenever you open up this type of menu, a BPI request will be sent.

In the payload the **recursive** flag (**true** or **false**) indicates whether you want the response to include the entire sub-tree or just the immediate children of the specified parameter (**oid**). Where **oid** could be set to an asterisk (*****) for all or a specific device parameter OID.

If a DashBoard user clicks through the parameter tree, DashBoard will send out the OIDs requested, and fill in the tree. Initially, DashBoard sends out **oid:*** and **recursive=false**, this indicates that it wants only the top level of OIDs returned. This implements a form of lazy-loading, where only the required OIDs are loaded as you need them. The BPI only includes the minimal amount of information required to fill in the tree and it only gets the required info when it's needed.

If a user attempts to use the **Search** to find a particular parameter or OID, DashBoard must send a request with **recursive=true** to get all of the information necessary to perform the search. When **recursive** is set to **true**, it ensures that you will get all the children and the children's children recursively.

Note: It is required that you support **recursive=true**, and recommended that you support **recursive=false**. If you choose not to support **recursive=false**, then you must still respond with a recursive list of BPI.

An example of a tree views in DashBoard includes the Visual Logic Editor for the Device & Parameters area (which appears only when you are editing ogScript code). For more details about the Visual Logic Editor, see the DashBoard User Guide.

The parameter list for **Devices & Parameters** is shown below:

Figure 5 *Visual Logic Editor - Devices & Parameters Tree View (Close-up)*

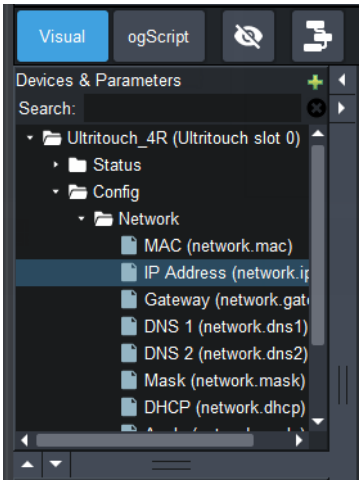
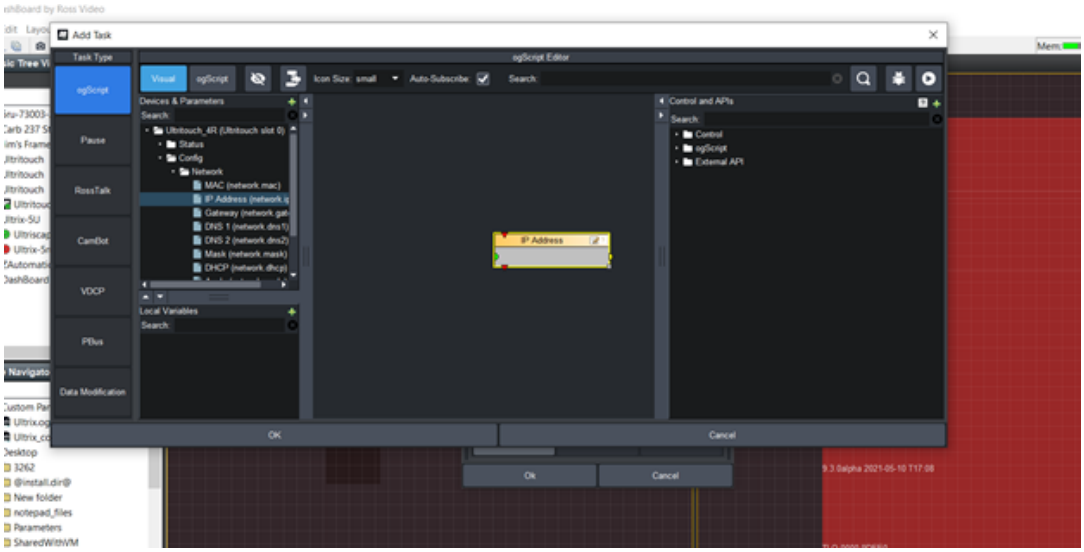


***Figure 6** Visual Logic Editor - Tree Views (Full View)*



"*basic-param-info-request*" *Syntax*

```
{
   "type":"basic-param-info-request",
   "slot": slot ID,
   "exe-id":"execution id",
   "payload":{
      "oid": oid*,
      "recursive": boolean
   }
}
```

Field	Type	Required	Description
type	String	Yes	Set to `basic-param-info-request`.
slot	Number	Yes	Destination slot for this message.
exe-id	String	Optional *If you are not expecting a response.	The unique execution ID that DashBoard client adds to the header of the basic-param-info-request message when it requests new basic param info. This allows DashBoard to verify that it has received the response to a specific request.

Field	Type	Required	Description
payload	Object	Yes	The "payload" provides the oid and recursive field.
oid	String	Yes	The "oid" is the parent parameter you want the children for. This is a single oid (not a list). If the "oid" is "*", it means you want the top-level parameters.
recursive	Boolean	Yes - • It is **required** to support "recursive": true • It is recommended to also support "recursive": false	In the payload the recursive flag (true or false) indicates whether you want the response to include the entire sub-tree or just the immediate children of the specified parameter.

Example Retrieving OIDs and Children of OIDs Recursively

To get an OID and the children of the specified OID, you must add the OID. You have the option to set "recursive" to "true" or to "false",

```
{
   "type":"basic-param-info-request",
   "slot": 25,
   "exe-id":"1",
   "payload":{
      "oid":"auto.detect",
      "recursive":true
   }
}
```

Example Retrieving Only Top Level OIDs

To get only the top level of OIDs (or parent OIDs), set "recursive" to "false", and set the "oid" to an asterisk, "*".

```
{
   "type": "basic-param-info-request",
   "slot": 25,
   "exe-id": "1",
   "payload":{
      "oid": "*",
      "recursive": false
   }
}
```

b) Update the "basic-param-info" response from a device

Follow the procedures below to populate the "basic-param-info" response from a device.

"basic-param-info"

Defines a response message from a device. The basic param info response provides support for specific OIDs and the option to recursively retrieve all of the children of the specified OID.

Note: There is an optional **metadata** field that allows you to provide additional information about your **basic-param-info** response and make changes to the requested data. For example, this could allow you to flag that you are responding recursive even if the request was for non-recursive. It could allow you to include an entire parameter's **basic-param-info** as opposed to a single element or to repeat the **basic-param-info** of the parent in the payload as opposed to only providing the **basic-param-info** for its children.

Syntax of Basic Param Info Response

```
{
    "metadata": {
      "index": value of Index,
      "parent-in-payload": boolean,
      "oid": "devices",
      "recursive": boolean
   },
    "exe-id": "executionID",
    "slot": slotID,
    "type": "basic-param-info",
    "payload": {
      "_d_OID1": "basic-param-info object 1",
      "_d_OID2": "basic-param-info object 2"
   }
}
```

Basic Param Info Response Attributes

Field	Type	Required	Description
metadata	String	No	This field can be used in scenarios where the device may want to respond to a basic-param-info" type request from the DashBoard Client with a more general (higher level) reply. **Note:** This field is optional and is only useful if you wish to change the level of the device reply from the one requested by the DashBoard Client.
exe-id	String	Yes	This is the unique execution ID that DashBoard client added to the header of the **basic-param-info** message when it requests new basic param info. This allows DashBoard to verify that it has received the response to a specific request.
slot	Number	Yes	Destination slot for this message.
type	String	Yes	Set to `basic-param-info`.
payload	Object	Yes	The payload will include information for every child parameter requested.
OID	Param Object (descriptor)	No	Parameter descriptor for each parameter object.

Child Attributes for Metadata field

Field	Type	Required	Description
oid	object	No	Allows the device to communicate to DashBoard which parameter it's reporting the basic-param-info for. If you plan to change which **oid** you're reporting the **basic-param-info** for, make sure to populate this field (and the **index** - if applicable).
index	object	No	See description for "oid" above. **Note**: If the device is reporting a higher level parameter, like an array type, then this field is required.
parent-in-payload	object	No	DashBoard usually expects only the children of the requested parameter in the **basic-param-info** message. If the device instead reports the parent in the message, then this attribute must be set to **"true"**.
recursive	object	No	This indicates to DashBoard whether your message contains the full (recursive) tree of children for the requested oid.

Example Retrieving OIDs and Children of OIDs Recursively

This example shows the device response when you have set `"recursive"` to `true` in the `"basic-param-info-request"`.

```
{
   "exe-id":"1630693459714:2",
   "slot":0,
   "type":"basic-param-info",
   "payload":{
      "0xFF10":{
         "name":"DEVICE_IP_ADDRESS",
         "type":"STRING"
      },
      "deviceoptions.calibration":{
         "name":"Run Calibration",
```

```
            "type":"INT16"
         },
         "devices":{
            "name":"Devices",
            "type":"STRUCT_ARRAY",
            "length":4,
            "children":[
               {
                  "name":{
                     "name":"Device Name",
                     "type":"STRING"
                  },
                  "status":{
                     "name":"",
                     "type":"INT16"
                  },
                  "connection_settings":{
                     "name":"Connection Settings",
                     "type":"STRUCT_ARRAY",
                     "length":10,
                     "children":[
                        {
                           "key":{
                              "name":"Key",
                              "type":"STRING"
                           },
                           "value":{
                              "name":"Value",
                              "type":"STRING"
                           }
                        }
                     ]
                  }
               }
            ]
         }
      }
   }
```

Example Retrieving Only Top Level OIDs

This example shows the device response when you have set `"recursive"` to `false` in the `"basic-param-info-request"`. Note that the `"devices"` OID, does not retrieve the child OIDs.

```
   {
      "exe-id":"1630692185591:0",
      "slot":0,
      "type":"basic-param-info",
      "payload":{
```

```
      "0xFF10":{
         "name":"DEVICE_IP_ADDRESS",
         "type":"STRING"
      },
      "deviceoptions.calibration":{
         "name":"Run Calibration",
         "type":"INT16"
      },
      "devices":{
         "name":"Devices",
         "type":"STRUCT_ARRAY",
         "length":4
      }
   }
}
```

Examples showing when to use the Metadata field

- When the DashBoard Client requests BPI for devices.1.connection_settings.0:

```
{
   "payload":{
      "oid":"devices.1.connection_settings.0",
      "recursive":false
   },
   "exe-id":"1648571608632:32",
   "slot":1,
   "type":"basic-param-info-request"
}
```

- If a device decides that's too granular, it responds with the entirety of the "devices" parameter:

```
{
   "metadata":{
      "index":-1,
      "parent-in-payload":true,
      "oid":"devices",
      "recursive":true
   },
   "exe-id":"1648571608632:32",
   "slot":1,
   "type":"basic-param-info",
   "payload":{
      "name":"Devices",
      "type":"STRUCT_ARRAY",
      "length":2,
      "children":[
         {
```

```
      "name":{
         "name":"Device Name",
         "type":"STRING"
      },
      "status":{
         "name":"",
         "type":"INT16"
      },
      "connection_settings":{
         "name":"Connection Settings",
         "type":"STRUCT_ARRAY",
         "length":2,
         "children":[
            {
               "key":{
                  "name":"Key",
                  "type":"STRING"
               },
               "value":{
                  "name":"Value",
                  "type":"STRING"
               }
            },
            {
               "key":{
                  "name":"Key",
                  "type":"STRING"
               },
               "value":{
                  "name":"Value",
                  "type":"STRING"
               }
            }
         ]
      },
      "connection_state":{
         "name":"Current Connection State",
         "type":"INT16"
      }
   },
   {
      "name":{
         "name":"Device Name",
         "type":"STRING"
      },
      "status":{
         "name":"",
         "type":"INT16"
      },
```

```
            "connection_settings":{
               "name":"Connection Settings",
               "type":"STRUCT_ARRAY",
               "length":1,
               "children":[
                  {
                     "key":{
                        "name":"Key",
                        "type":"STRING"
                     },
                     "value":{
                        "name":"Value",
                        "type":"STRING"
                     }
                  }
               ]
            },
            "connection_state":{
               "name":"Current Connection State",
               "type":"INT16"
            }
         }
      ]
   }
}
```

Example to show the effect of the parent-in-payload flag:

```
{
   "payload":{
      "oid":"struct",
      "recursive":false
   },
   "exe-id":"1648575366928:3",
   "slot":0,
   "type":"basic-param-info-request"
}
```

Example with no flag (reporting only the children of the struct):

```
{
   "slot":0,
   "type":"basic-param-info",
   "exe-id":"1648575366928:3",
   "payload":{
      "name":{
         "name":"Name",
         "type":"STRING"
      },
      "integer":{
```

```
            "name":"Integer",
            "type":"INT16"
         },
         "substruct":{
            "name":"Child Struct",
            "type":"STRUCT"
         }
      }
   }
```

Example with flag (reporting parent and children)

```
{
   "metadata":{
      "parent-in-payload":true
   },
   "exe-id":"1648575366928:3",
   "slot":0,
   "type":"basic-param-info",
   "payload":{
      "name":"My Struct Param",
      "type":"STRUCT",
      "children":[
         {
            "name":{
               "name":"Name",
               "type":"STRING"
            },
            "integer":{
               "name":"Integer",
               "type":"INT16"
            },
            "substruct":{
               "name":"Child Struct",
               "type":"STRUCT"
            }
         }
      ]
   }
}
```

Related to

- basic-param-info-request — (see page 4–23)

"basic-param-info" object

See the entry for **basic-param-info-request** and **basic-param-info** for more information.

Note: It is required that you include the oid, name, type (and length of the parameter for arrays only) in the basic-param-info object.

Syntax

```
"OID":{
    "oid": "oid",
    "name": "parameter name",
    "type": "parameter type",
    "length": "integer"
}
```

Member of

- basic-param-info — (see page 4–26)

Members

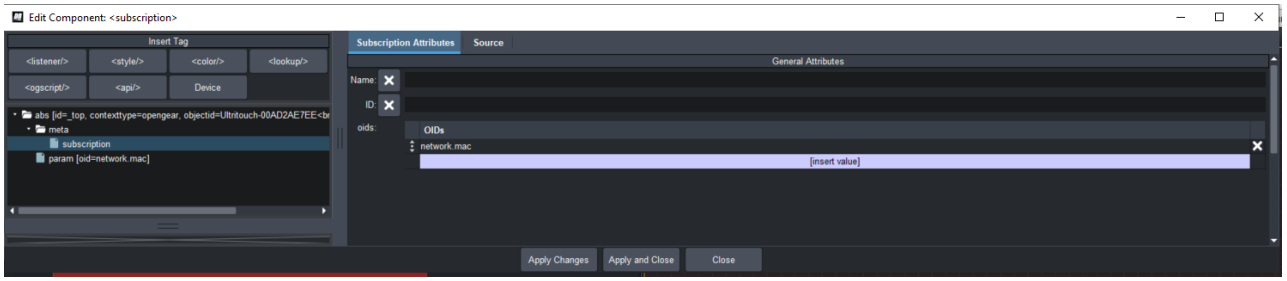
Member	Type	Required	Description
oid	String	Yes	The Object Identifier (OID) for this parameter. The OID must be globally unique within the scope of the device.
name	String	Yes	Parameter Name
type	String	Yes	The data type for the parameter. See "Data Types" on page 3–34 for a list of valid data types.
length	Integer	Yes - It is required for array data types.	The length of the parameter.

6. Add the subscription tag to the DashBoard CustomPanels

a. In DashBoard, to add the subscription flag to the custom panel, go to the **Component Editor > Source** tab and within the top level container of the `opengear` device context, include a list of all of the OIDs used in that custom panel page.

Tip: You can navigate to the editor by double-clicking on a blank area in your DashBoard CustomPanel while in PanelBuilder **Edit** Mode, and the **Component Editor** will open with the header selected in the side navigation. You can see an example with an Ultritouch panel below:

b. Add the `<subscription oids=""/>` tag. Once you have added the tag, in the tree view on the left side, select **subscription**. Select the **Subscription Attributes** tab, and under **OIDs**, you can add the list of device parameter OIDs you wish to include.



Syntax

```
<abs contexttype="opengear" id="_top" keepalive="false" objectid="My_Device"
objecttype="MyDevice" subscriptions="true">
   <meta>
      <subscription oids="list of comma-separated oids here"/>
   </meta>
</abs>
```

Ultrix Example

```
<abs contexttype="opengear" id="_top" keepalive="false" subscriptions="true"
objectid="MyUltrix" objecttype="Ultrix">
   <meta>
      <subscription
oids="devices.*,types.audiomixer*,params.framesrc.ports.0.port"/>
   </meta>
...
```

✶ **Note:** You can use wildcard asterisks to include multiple OIDs simultaneously that have the same starting prefix in the name. The wildcard should be added after this prefix. These wildcards are useful when you don't want to type out a whole list of similar OIDs manually. Instead you can add a subset of OIDs by including a wildcard. If wildcards are used, your list of subscriptions are optimized by DashBoard to use the wildcard that includes the most items. For more details see, "**About Wildcards**".

About Wildcards

Adding a wildcard asterisk to a list of parameter OIDs in a DashBoard device panel, will allow you to quickly add multiple sets of parameter OIDs that start with the same prefix. You can only add an asterisk to the end of an oid prefix name. The asterisk means that you will subscribe to all parameters that start with the prefix you entered.

For example, if you wanted to add three OIDs, **`types.audiomixer`**, **`types.audiomixerpartition`** and **`types.audiosound`**, you could use the following wildcards: **`ty*`**, **`types.audio*`**, or **`types.au*`**. If you use more than one wildcard that applies to the same parameters, DashBoard will choose the most efficient wildcard to optimize. In the example above, **`ty*`** would be used. You cannot add a wildcard before the prefix or have text after the wildcard. For example, **`*ypes.`** and **`ty*p`** are not valid.

7. Add the subscription list to the DashBoard device panels (optional)

You must add support to subscribe and/or unsubscribe from parameter updates in the device panel's OGLML structure. You can also use the template that is provided in the DashBoard PanelBuilder Script Palette.

For More Information on...

- Templates, see: "**Using the DashBoard Script Palette's Command Templates**"

"params.subscribe" Syntax

```
//SUBSCRIBE
var subList = new Array();
   subList.push("oid1");
   subList.push("oid2");
var subscriptionOwnerObject = params.subscribe(subList, callback);
```

"params.subscribe" Example

```
 <task tasktype="ogscript">
          var subList= new Array();
             subList.push("deviceoptions.speakerlevel");
             subList.push("db.touch.version.*");
          var subscriptionOwnerObject = params.subscribe(subList, callback);
   ogscript.putObject('my-subscription-owner-object', subscriptionOwnerObject);
</task>
```

Function	Parameters	Returns	Description
subscribe	[Array of strings, callback]	Returns `subscriptionOwnerObject` for later use to unsubscribe.	Subscribes to parameters with the provided OIDs.

Explanation

In this example, the **`ogscript.putObject`** is used to retain the result of the **`params.subscribe`** function, which is later used to unsubscribe.

"params.unsubscribe" Syntax

```
//UNSUBSCRIBE
params.unsubscribe(subscriptionOwnerObject);
```

Function	Parameters	Returns	Description
unsubscribe	[subscriptionOwnerObject]	N/A	Unsubscribes from the OIDs provided by the subscriptionOwnerObject.

"params.subscribe" Example

```
 <task tasktype="ogscript">
          var subscriptionOwnerObject =
ogscript.getObject('my-subscription-owner-object');
          params.unsubscribe(subscriptionOwnerObject);
</task>
```

Explanation

In the subscribe example above, the **ogscript.putObject** is used to retain the result of the **params.subscribe** function and **ogscript.getObject** fetches it when we want to unsubscribe (**params.unsubscribe)**. You can see that the subscribe response object is used to unsubscribe.

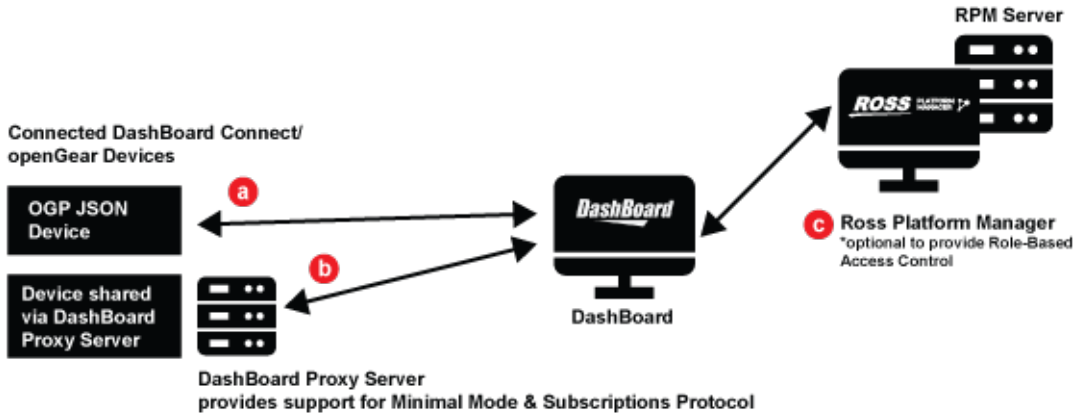
Now that you have successfully implemented subscriptions support, make sure that you leverage the built-in automations within DashBoard to support subscriptions. See, "**Using the DashBoard Script Palette's Command Templates**".

DashBoard PanelBuilder Features

A DashBoard panel builder may wish to connect to OGP devices with subscriptions support, or share multiple devices through the DashBoard Proxy Server (which provides support for Minimal Mode and Subscriptions protocol). This workflow greatly improves the user experience for any users who need DashBoard to run smoothly with numerous connected devices, but especially if those devices send the DashBoard Client a high volume of parameter updates. Subscriptions improves the DashBoard platform's performance by eliminating any unnecessary

communication with connected devices that are not in use, and that otherwise would send a high volume of parameters even while idle. You can see an example of a typical workflow below:

***Figure 7** DashBoard Panel Workflow with Devices that Support Subscriptions*



a. **OGP JSON Device** — Any OGP JSON device that supports subscription. This includes Ross devices in the DashBoard Connect ecosystem that support subscriptions, such as Ultritouch.

b. **Device Shared via DashBoard Proxy Server** — The DashBoard Proxy Server will provide support for Minimal Mode and Subscriptions Protocol.

c. **The Ross Platform Manager (RPM)** — This is an optional component that can be used to provide Role-Based Access Control for connected devices.

Overview

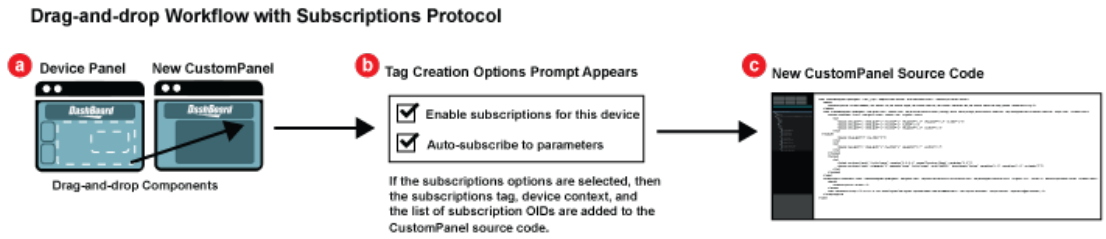
DashBoard panel builders can use the following PanelBuilder automations to create device panels for OGP devices/ or devices shared through the DashBoard Proxy Server:

- Drag-and-Drop Panel Components with Subscriptions Support - Reuse panel components from an existing device panel that supports subscription, and use the provided checkboxes to choose whether newly dragged in components will have subscriptions turned on or off, and whether to add the parameters to the subscription list for the device.
- Adding Subscriptions Support for a Device Context - When adding a device to DashBoard context, panel builders can choose to enable subscriptions for that device using the new "**Enable subscriptions for this device**" checkbox in the Device Context Dialog.
- Using the DashBoard Script Palette's Command Templates - Use the ogScript Editor's Script Palette to add provided code snippets for **`param.subscribe`** and **`param.unsubscribe`**.

Drag-and-Drop Panel Components with Subscriptions Support

DashBoard provides the ability to drag-and-drop components from one device panel for reuse in a secondary CustomPanel, and the any data-backed components will receive any updates from the original source panel.

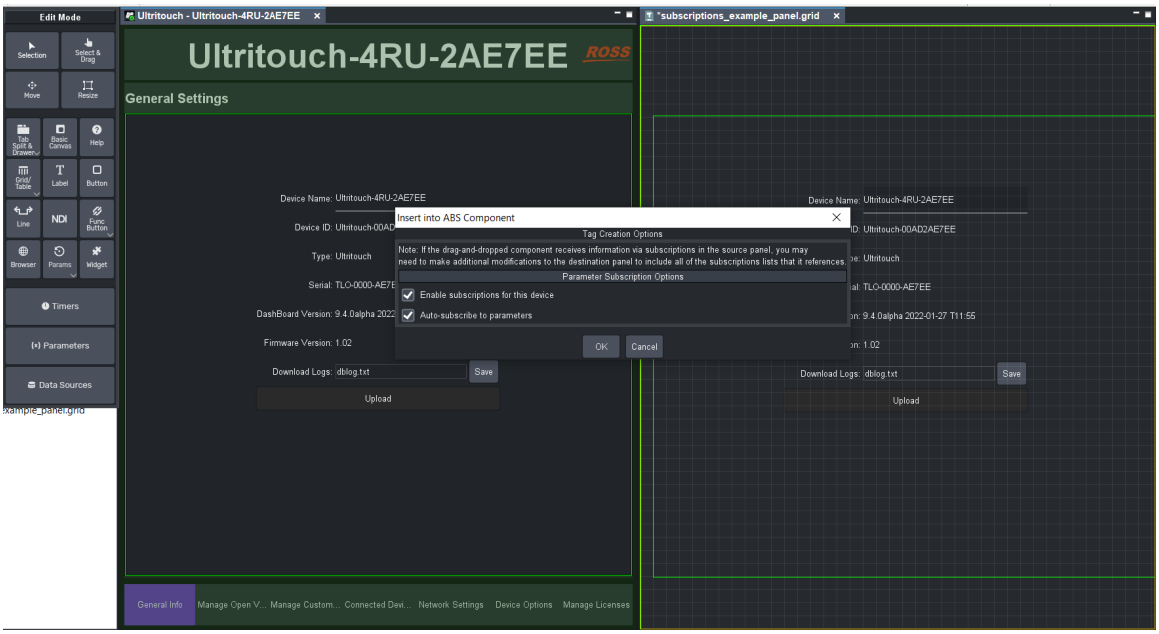
DashBoard provides automatic support for subscriptions using the following workflow below:



a. When you want to bring components from an existing openGear device frame that supports subscriptions into a new CustomPanel, you can drag and drop parameters from the existing device panel into your new panel.

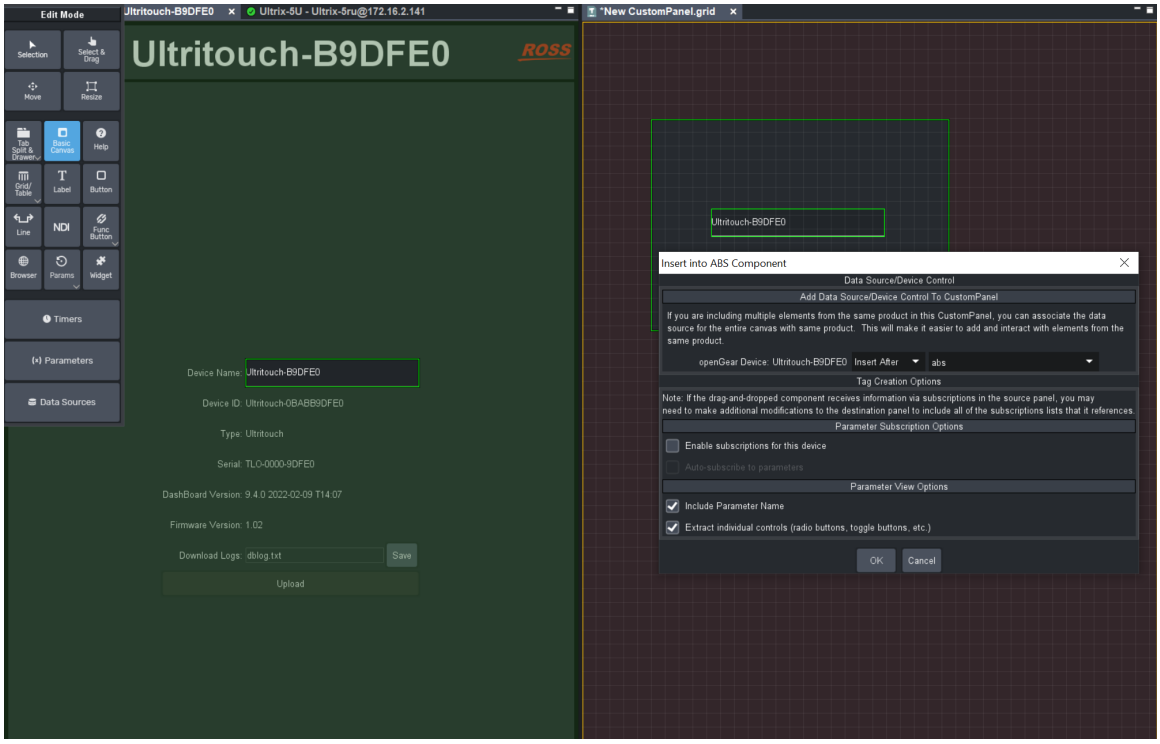
b. When you drag a parameter in, a Insert into ABS Component dialog appears. This dialog now has two new options: "**Enable Subscriptions for this device**" and "**Auto-subscribe to parameters**", as shown in the screen-shot below. Checking both of these boxes will automatically add the parameter to the panel's subscription list.

***Figure 8** An Ultritouch Device that supports Subscriptions (left) and a New Panel (right)*



✶ **Tip**: You can also drag-and-drop a component into an existing area, such as a basic canvas, and the dialog prompt will have additional options that allow you to choose where you want the data source to be inserted.

***Figure 9** An Ultritouch Device component that has been dropped into an existing basic canvas*

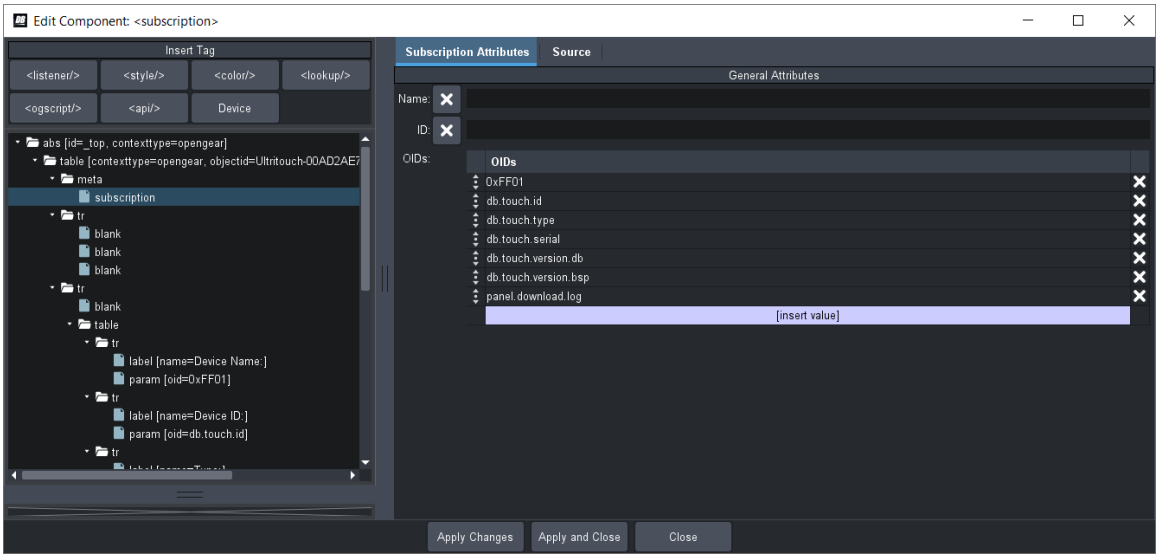


c. You can verify that it's been added, by double-clicking on the displayed parameter to open the **Component Editor** dialog and navigating to the **Source** code tab to view the updated subscription list. It will include the parameter's oid in the subscription list:

```
<meta>
   <subscription oids="0x907"/>
<meta>
```

You can also view the OIDs, in the **Component Editor** dialog's tree view by selecting **Subscriptions** in the tree view. The Subscriptions Attributes tab on the right includes the full list of OIDs, as shown below:

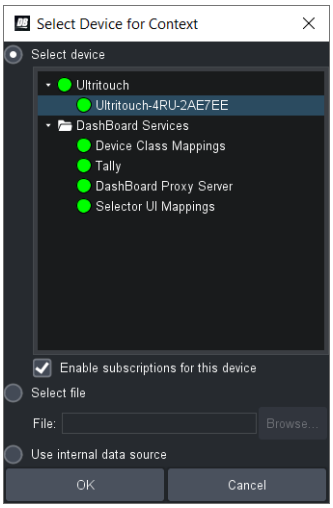
***Figure 10** Subscriptions OIDs displayed in the Subscriptions Attributes tab*



Adding Subscriptions Support for a Device Context

When adding a device to DashBoard panel's context, you can choose to enable subscriptions for that device using the new "**Enable subscriptions for this device**" checkbox in the Device Context dialog, as shown below:

***Figure 11** Enabling Subscriptions Support in the Device Context Dialog*



✶ **Note:** The **Enable subscriptions for this device** checkbox only appears if the selected device supports subscriptions, if not it will be disabled.

Selecting the **Enable subscriptions for this device** checkbox, and clicking **OK**, will automatically add a **`subscriptions="true"`** tag to the device context, as shown in the example below:

```
<abs contexttype="opengear" id="_top" keepalive="false" objectid="MyUltritouch"
objecttype="Ultritouch Device" subscriptions="true">
```

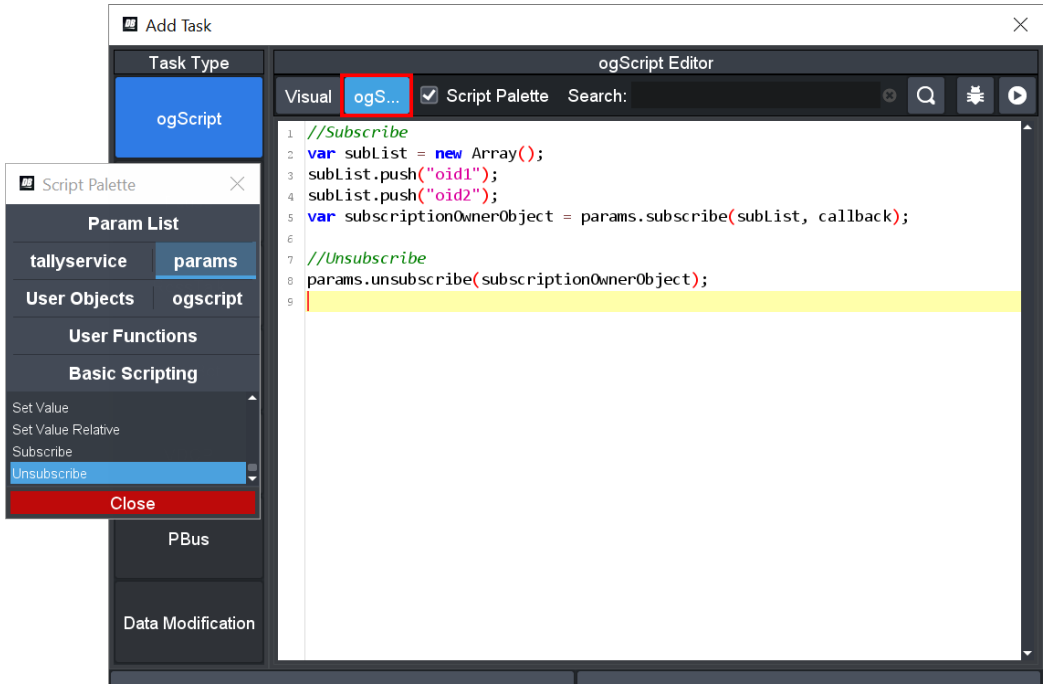
If the checkbox is not selected, then DashBoard automatically adds a **`subscriptions="false"`** tag to the device context. For more details on how to add a device context, see the procedure below.

Adding a device context

1. In PanelBuilder **Edit Mode**, double-click anywhere on the blank canvas to open the Component Editor, and scroll down to the **Data Source/ Device Control** area.
2. For the openGear or XPression Datalinq field, select the checkbox and click **Configure**. The **Select Device for Context** dialog appears, with the option to select a device, and enable subscriptions (only if that device supports subscriptions).

Using the DashBoard Script Palette's Command Templates

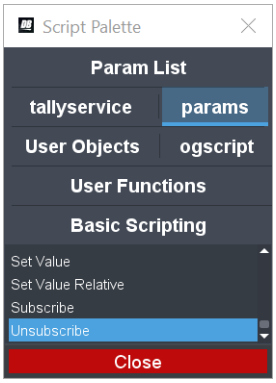
You can use the **Subscribe** and **Unsubscribe** command templates that are built into the **Script Palette**. Follow the steps in the procedure below to use the templates, as shown below:



To add the template for the Subscribe or Unsubscribe command as code snippets

1. Ensure that you have a button with a task added.
2. In the manual **ogScript Editor,** select the **Script Palette** checkbox, and click the **params** tab.
   Note: The Script Palette is currently only available in the manual **ogScript** text editor, not the **Visual** editor.

3. Move the cursor to the area that you would like to add the code snippet, and then double-click on the "**Subscribe**" or "**Unsubscribe**" templates to add the code snippet there.



4. Close the Script Palette when you are finished, and save your changes.

Subscriptions Glossary

This section provides a list of terms used in this document, and defines them for clarity.

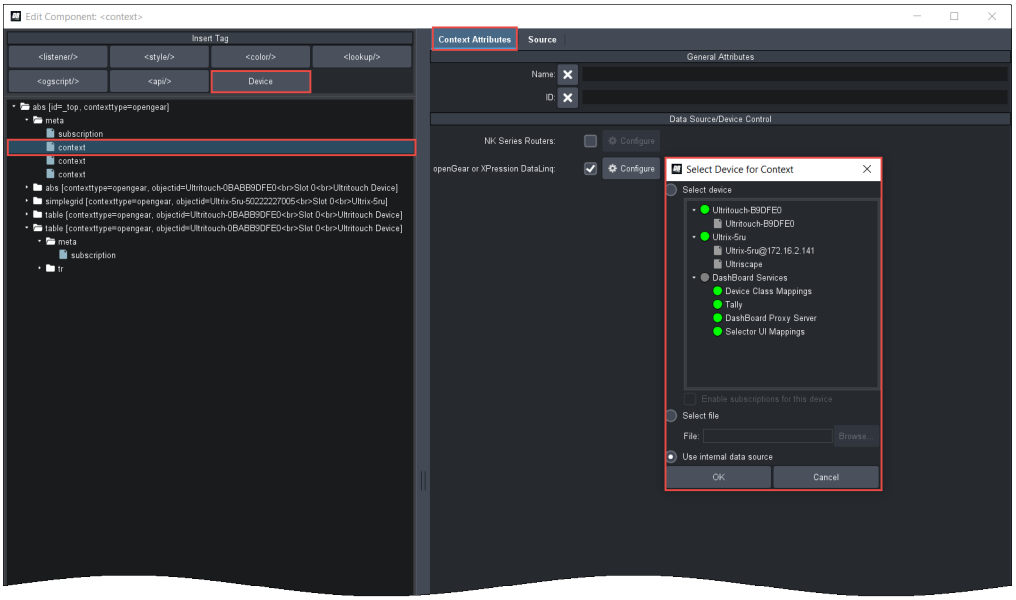
DashBoard Application/ DashBoard platform — Refers to Ross Video's DashBoard platform, which is a free application designed for remote control and monitoring of the openGear architecture and openGear™ ecosystem.

DashBoard Client side — Refers to the messages sent from the DashBoard Client side (the OGP JSON Client).

DashBoard Connect™ Ecosystem — The ecosystem of openGear devices manufactured by our openGear partners that can be controlled using the DashBoard Facility Control System.

Device context — Subscriptions will be implemented within the device's context, which defines the scope within the OGLML document. Each default panel is created with a default context named **`"opengear"`**, but if multiple devices are linked to the OGLML document, each will have its own separate context. A device context can be a **`<context/>`** tag or it can be defined in any container tag by setting the **`contexttype`** attribute to **`"opengear"`** and specifying the device's ID with the **`objectid`** attribute.

The diagram below shows a **Device** button that allows you to insert a **`<context/>`** tag, which appears highlighted in the tree view on the left. When the context is still highlighted, on the right side under the **Context Attributes** tab, you can configure a device for that context.



Device side — Refers to the device messages sent from the device side (the OGP JSON Server).

openGear Protocol (OGP) — OGP is the protocol for parameter reporting, control and alarm reporting.

JSON Objects

Object Hierarchy

This section documents the JSON representation of data objects. The object hierarchy is as follows:

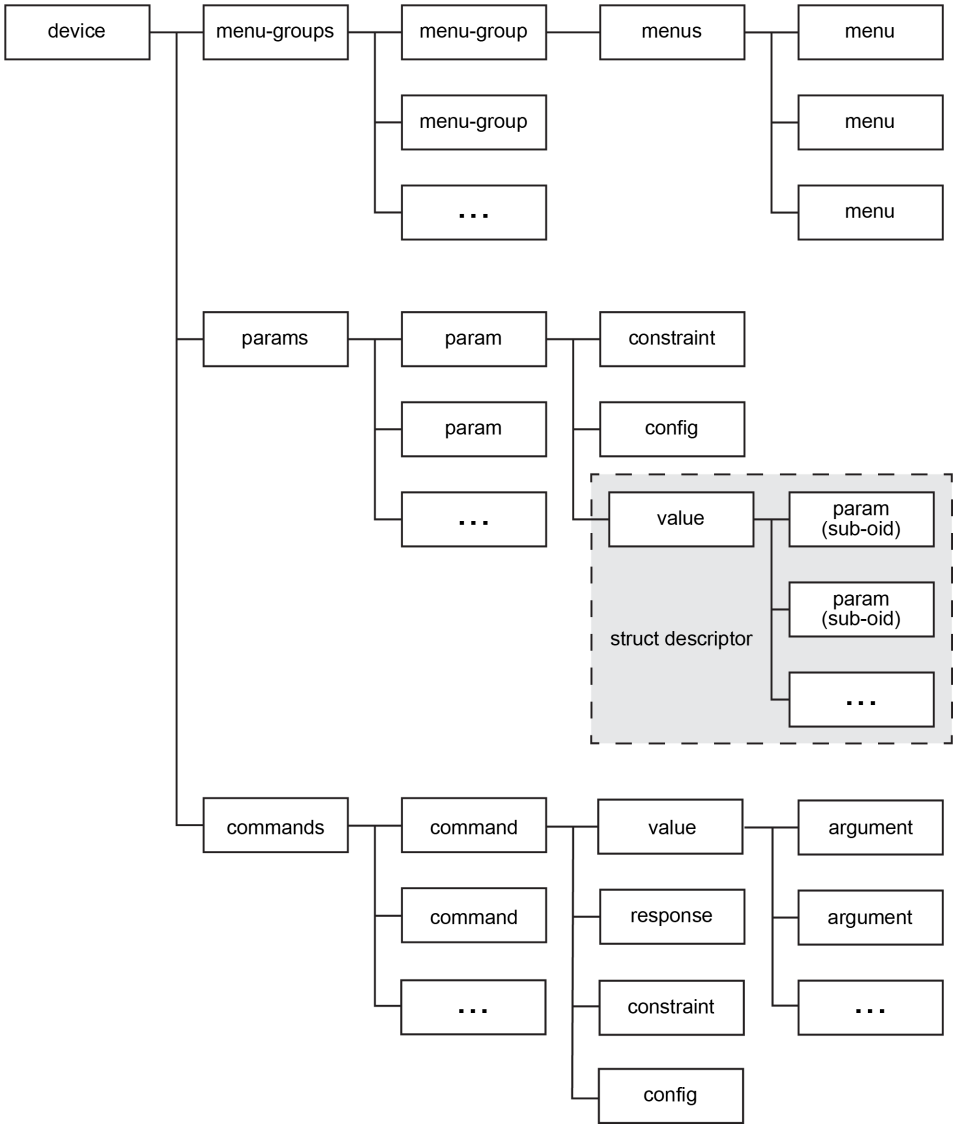


Figure 4.1 *- JSON Object Hierarchy*

The following objects are defined:

Object	Description
config (see page 4–46)	Container for extra configuration data for a parameter
constraint (see page 4–47)	Defines the constraint on a single parameter.
device (see page 4–56)	Defines all parameters and menus of a device.
menu (see page 4–56)	Defines a single menu tab.

Object	Description
menus (see page 4–57)	Defines a single menu group.
menu-group (see page 4–58)	Defines a set of menu groups for a device; a collection of menu-group objects.
menu-groups (see page 4–58)	Defines a set of menu groups for a device; a collection of menu-group objects.
param (see page 4–59)	Defines a single parameter.
params (see page 4–62)	Defines a set of parameters for a device; a collection of param objects.
value (struct descriptor) (see page 4–64)	Defines a struct descriptor.
command (see page 4–68)	Defines a single command.
commands (see page 4–68)	Defines a command group.
value (argument) (see page 4–64)	Defines a value (argument).

config Object

Provides a container for extended configuration key-value pairs for elements related to a parameter. Contents are dependent on other constraints, parameter types or widgets.

Syntax

```
"config" : {
                "key": value,
                "key": value,
                . . .
            }
```

Member of

- param Object (descriptor) — (see page 4–59)

Members

Varies

Examples

The following config object sets attributes of a table widget:

```
"_d_params.levels": {
      "oid": "params.levels",
      "name": "Level List",
      "type": "STRUCT_ARRAY",
      "widget": "table",
```

```
                "constraint": {
                    "type": "STRUCT",
                    "templateoid": "types.level",
                },
        "config" : {
    "w.alwaysscroll": true,
    "w.hgrid": true,
    "w.vgrid": true
                }
    }
```

constraint Object

Defines the constraint of a parameter. The structure of this object depends upon the type of constraint.

Constraint	Constraint Type	Param Type
Unconstrained (see page 4–48)	INT_NULL	INT16 INT16_ARRAY INT32 INT32_ARRAY
	FLOAT_NULL	FLOAT32 FLOAT32_ARRAY
Range Constraint (see page 4–48)	INT_RANGE INT_STEP_RANGE	INT16 INT16_ARRAY INT32 INT32_ARRAY
	FLOAT_RANGE FLOAT_STEP_RANGE	FLOAT32 FLOAT32_ARRAY
Integer Choice Constraint (see page 4–49)	INT_CHOICE	INT16 INT16_ARRAY INT32 INT32_ARRAY
String Choice Constraint (see page 4–50)	STRING_CHOICE	STRING STRING_ARRAY
External Constraints (see page 4–51)	EXTERNAL	STRING
Alarm Table (see page 4–52)	ALARM_TABLE	INT16 INT32
Structure (see page 4–54)	STRUCT	STRUCT_PARAM STRUCT_ARRAY

***Note**: If no constraint is specified for a parameter, it will be unconstrained by default.*

Refer to the appropriate section below for definition of the constraint object for each constraint type.

constraint Object (Unconstrained)

Specifies that a parameter is unconstrained. All parameters are considered unconstrained by default if no constraint is applied.

Syntax

```
"constraint" : {
                "type": "constraint type",
            }
```

Member of

- param Object (descriptor) — (see page 4–59)

Members

Member	Type	Required	Description
type	String	Yes	See type table below

The type field must be set according to the type of the parameter:

Param Type	Constraint Type
INT16 INT16_ARRAY INT32 INT32_ARRAY	INT_NULL
FLOAT32 FLOAT32_ARRAY	FLOAT_NULL

Examples

The following constraint specifies an integer to be unconstrained:

```
"constraint": {
                    "type": "INT_NULL"
                }
```

constraint Object (Range Constraints)

Constrains a numeric parameter type to a specific range. Minimum and maximum values effect the parameter's valid range. Display minimum and maximum values scale the parameter value to a different range for display purposes. Finally, a step value can be set to constrain the minimum step size a value may be changed by a range constraint (see "Range Constraints" on page 3–5 for details).

Syntax

```
"constraint" : {
                "type": "constraint_type",
                "min": minimum parameter value,
                "max": maximum parameter value,
                "display-min": minimum display value,
                "display-max": maximum display value,
                "step": increment step size
            }
```

Member of

- param Object (descriptor) — (see page 4–59)

Members

Member	Type	Required	Description
type	String	Yes	See type table below.
min	Number	Yes	The minimum value a parameter may be set to.
max	Number	Yes	The maximum value a parameter may be set to.
display-min	Number	No	The displayed value of the parameter when the parameter has a value of `min`. The default value is `min`.
display-max	Number	No	The displayed value of the parameter when the parameter has a value of `max`. The default value is `max`.
step	Number	No	Smallest increment a value may be changed by. Only valid for INT_RANGE_STEP and FLOAT_RANGE_STEP constraints. Spinner widgets will increment a parameter by the step value. Note that the step increment is applied to the parameter value, not the display value.

The type field must be set according to the type of the parameter:

Param Type	Constraint Type	Step Constraint
INT16 INT16_ARRAY INT32 INT32_ARRAY	INT_RANGE	no
	INT_STEP_RANGE	yes
FLOAT32 FLOAT32_ARRAY	FLOAT_RANGE	no
	FLOAT_STEP_RANGE	yes

Examples

The following constraint constrains an integer to [0, 255] mapping it to a display range of [0, 100], and the value increments by steps of 2.

```
"constraint": {
                    "type": "INT_STEP_RANGE",
                    "min": 0,
                    "max": 255,
                    "display-min": 0,
                    "display-max": 100,
                    "step": 2
                }
```

constraint Object (Integer Choice Constraints)

Choice constraints provide a list of possible values for a parameter, based upon a text selection. For integer parameters, the parameter may only be assigned a value specified in the constraint.

Syntax

```
"constraint" : {
```

```
            "type": "INT_CHOICE",
            "choices": [
{
"name" : "choice 1 name",
"value" : "choice 1 value",
},
{
"name"  : "choice 2 name",
"value" : "choice 2 value",
},
. . .
]
      }
```

Member of

- param Object (descriptor) — (see page 4–59)

Members

Member	Type	Required	Description
type	String	Yes	Set to `INT_CHOICE`
choices	Array of Choice Objects	Yes	The choice objects are JSON objects containing the name and value fields (as shown in the example)
name	String	Yes	Text name for the current enumerated choice
value	Number	Yes	Numeric assignment of current enumerated choice.

Examples

The following constraint provides two options on an integer parameter.

```
"constraint": {
      "type": "INT_CHOICE",
      "choices": [
          {
              "name": "Enabled",
              "value": 1
          },
          {
              "name": "Disabled",
              "value": 0
          }
      ]
  }
```

constraint Object (String Choice Constraints)

Choice constraints provide a list of possible values for a parameter, based upon a text selection. For String parameters, the constraint provides a set of defaults, but the user may arbitrarily enter any other value for the parameter.

Syntax

```
"constraint" : {
                "type": "STRING_CHOICE",
                "choices": [
       "choice 1 name",
       "choice 2 name",
       . . .
       ]
            }
```

Member of

- param Object (descriptor) — (see page 4–59)

Members

Member	Type	Required	Description
type	String	Yes	Set to `STRING_CHOICE`
choices	Array of String	Yes	Available strings for drop-down widget

Examples

The following constraint provides four string options for a String parameter.

```
"constraint": {
                "type": "STRING_CHOICE",
                "choices": [
                    "CH 001",
                    "CH 002",
                    "CH 003",
                    "CH 004"
                 ]
              }
```

constraint Object (External Constraints)

An EXTERNAL_CONSTRAINT is used to indicate that the constraint for this parameter is provided in an external object (“External Data Objects” on page 3–35 and “External Data Objects” on page 6–37), rather than embedded within the parameter descriptor. This constraint simply provides a reference to the external object.

Syntax

```
"constraint" : {
                "type": "EXTERNAL",
                "eo": "0x05"
            },
```

Member of

- param Object (descriptor) — (see page 4–59)

Members

Member	Type	Required	Description
type	String	Yes	Set to `EXTERNAL`
eo	String	Yes	The external object ID that points to the file that contains the external constraints.

Examples

The following constraint provides a link to the external object ID 0x05.

```
"constraint" : {
                "type": "EXTERNAL",
                "eo": "0x05"
            },
```

The contents of the file that is referred to as `0x05` would contain the constraint as defined in the other sections above. For example, External Object `0x05` could contain a float range constraint, as shown below:

```
{
 "type": "FLOAT_STEP_RANGE",
 "min": "5.0",
 "display-max": "35.0",
 "max" "1",
 "precision": "1",
 "display-min": "5.0",
 "step": "0.5"
}
```

constraint Object (Alarm Constraints)

Alarm constraints map a set of alarms as bitfields into an INT16 or INT32. Each bit represents an independent alarm which may have a message and severity assigned to it. Alarm parameters contribute to the device's overall alarm status in DashBoard; the most severe alarm set will determine the device's overall reported alarm status.

Syntax

```
"constraint" : {
                "type": "ALARM_TABLE",
                "alarms": [
       {
       "bitno": bit index,
       "message": "alarm message",
       "severity": alarm severity,

       },
       {
       "bitno": bit index,
       "message": "alarm message",
       "severity": alarm severity,

       },
       . . .
       ]
            }
```

Member of

- param Object (descriptor) — (see page 4–59)

Members

Member	Type	Required	Description
type	String	Yes	Set to `ALARM_TABLE`
alarms	Array of alarm Objects	Yes	See fields below
bitno	Number	Yes	The bit position for the alarm (0 is LSB). Valid range is: • INT16: 0..15 • INT32: 0..31
severity	Number	Yes	The severity of the alarm: • 0 = OK • 1 = WARN • 2 = ERROR

Examples

The following constraint has three alarms.

```
"constraint": {
    "type": "ALARM_TABLE"
    "alarms": [
        {
            "bitno": 0,
            "message": "HW A-OK",
            "severity": 0,
            "value": 1
        },
        {
            "bitno": 1,
            "message": "HW warning",
            "severity": 1,
            "value": 2
        },
        {
            "bitno": 2,
            "message": "HW error 1",
            "severity": 2,
            "value": 4
        },
        {
            "bitno": 3,
            "message": "HW error 2",
            "severity": 2,
            "value": 8
        }
```

```
            ],
        }
```

constraint Object (Struct Constraints)

Struct Constraints allow a parameter to define a complex structure of multiple parameters. The Struct Constraint is applied to each parameter that is an instance of a Struct.

Syntax

```
"constraint" : {
       "type": "STRUCT",
       "templateoid" : "template OID",
       "description" : "structure description",
       "structname" : "structure name",
       "structtype" : "structure type"
           }
```

Member of

- param Object (descriptor) — (see page 4–59)

Members

Member	Type	Required	Description
type	String	Yes	Set to `"STRUCT"`
templateoid	String	No	Specifies a template OID to pre-populate the structure. All parameters, constraints and widgets for the sub-OIDs are copied from the specified template OID.
description	String	No	Text description of the Structure type
structname	String	No	Structure name
structtype	String	No	Defines the structure type. Used by custom widgets to reference backing parameters.

Examples

The following code is an example of a struct instance. The struct descriptor from the parameter `types.level` is inherited and defines the structure of the struct:

```
"_d_params.levels": {
      "oid": "params.levels",
      "name": "Level List",
      "type": "STRUCT_ARRAY",
      "widget": "table",
      "value": [
          {
              "LevelName": "3G",
              "Color": -9137676,
              "Description": "The 3G Level"
          },
          {
              "LevelName": "HD",
              "Color": -5046403,
```

```
                    "Description": "Not the 3G Level"
                },
                {
                    "LevelName": "SD",
                    "Color": -607985,
                    "Description": "Also not the 3G Level"
                }
            ],
            "constraint": {
                "type": "STRUCT",
                "templateoid": "types.level",
                "description": "Descriptor for types.level",
                "structtype": "types.level"
            }
        }
```

The definition of `types.level` looks like this:

```
    "_d_types.level": {
        "oid": "types.level",
        "name": "Level (Type)",
        "type": "STRUCT_ARRAY",
        "widget": "default",
        "value": [
            {
                "_d_LevelName": {
                    "name": "Level Name",
                    "type": "STRING",
                    "widget": "default",
                    "maxlength": 0,
                    "totallength": 0,
                    "value": "Placeholder"
                },
                "_d_Color": {
                    "name": "Color",
                    "type": "INT32",
                    "widget": "color-picker-popup",
                    "precision": 0,
                    "constraint": {
                        "min": -2147483648,
                        "max": 2147483647,
                        "type": "INT_STEP_RANGE",
                        "unconstrained": true
                    },
                    "value": 16711680
                },
                "_d_Description": {
                    "name": "Description",
                    "type": "STRING",
                    "widget": "default",
```

```
                    "maxlength": 0,
                    "totallength": 0,
                    "value": "Placeholder"
                }
            }
        ]
    }
```

device Object

Defines the device. Sent in response to a `device-request` message.

Syntax

```
{               "slot": slot id,
               "detail": { level of detail },
               "menu-groups": { menu-groups object },
               "params": { params object },
    "multi-set-enabled": multi-set-flag
        }
```

Member of

None

Members

Member	Type	Required	Description
slot	Number	Yes	The slot identifier for the device.
detail	String	No	The level of detail for parameter updates. You can set the level of detail to either `minimal` or `full`.
menu-groups	menu-groups Object (see page 4–58)	No	Definition of the menu groups for the device.
params	params Object (see page 4–62)	No	Definition of the parameters for the device.
multi-set-enabled	Boolean	No	• **true** (default) – DashBoard will use the multi-value Message to report multiple parameter changes. • **false** – DashBoard will use the value Message to report all parameter changes independently.

menu Object

Defines a menu. Each menu is displayed as an individual tab in DashBoard.

Syntax

```
"menu-identifier": {
      "name": "menu name",
                  "id": menu id,
                  "unique": menu unique identifier,
```

```
        "disabled": disabled flag,
        "hidden": hidden flag,
        "oids": [ "OID1", "OID2",  . . . "OIDn"  ]
    }
```

Member of

- menus Object — (see page 4–57)

Members

Member	Type	Required	Description
name	String	Yes	Name of the menu
id	Number	Yes	Numeric ID for the menu. Menu tabs within a menu group are displayed in numeric order, lowest first. This value may be changed to dynamically re-order menus.
unique	Number	Yes	Unique numeric identifier for this menu. This value must be only set once and not changed.
disabled	Boolean	No	• `true` – menu is read-only • `false` – menu is read/write Default: `false`
hidden	Boolean	No	• `true` – menu is not displayed • `false` – menu is displayed Default: `false`
oids	Array of String	No	List of parameters to be displayed in this menu. Parameters are displayed in the order listed in this array.

Examples

This example creates a menu tab called “Card Info”, displaying 4 parameters defined the OIDs 0x105, 0x107, 0x102 and 0x10B.

```
"0": {
      "name": "Card Info",
      "id": 0,
      "unique": 0,
      "disabled": false,
      "hidden": false,
      "oids": [
          "0x105",
          "0x107",
          "0x102",
          "0x10B"
      ]
  }
```

menus Object

Contains a collection of menus.

Syntax

```
"menus" : {
                "menu-identifier": { menu object },
                   "menu-identifier": { menu object },
                   "menu-identifier": { menu object },
        . . .
             }
```

Member of

- menu-group Object — (see page 4–58)

Members

Member	Type	Required	Description
menu-identifier	menu Object (see page 4–56)	Yes	Menu definition. Each menu is presented as a tab within the menu group in DashBoard.

menu-group Object

Defines a menu group.

Syntax

```
"menu-group-identifier" : {
                "name": "menu group name"
                "menuid":  menu-group-id-number,
                "menus": { menus object }
            }
```

Member of

- menu-groups Object — (see page 4–58)

Members

Member	Type	Required	Description
name	String	Yes	Name of the menu group.
menuid	Number	Yes	The unique menu group identifier. For default openGear layout, menuid=0 corresponds to the status menu (left-pane), and menuid=1 corresponds to the config menu (right-pane).
menus	menus Object (see page 4–57)	No	Collection of menus contained within this menu group. Each menu is presented as a tab within the menu group in DashBoard.

menu-groups Object

Defines the collection of menu groups for a device. `status` and `config` menus are used in default DashBoard layouts. Additional menu groups may be defined, but must use OGLML and/or ogScript to display in DashBoard.

Syntax

```
"menu-groups" : {
                "status": { menu-group object },
```

```
                "config": { menu-group object },
                "menu-group": { menu-group object },
        . . .
            }
```

Member of

- device Object — (see page 4–56)

Members

Member	Type	Required	Description
status	menu-group Object (see page 4–58)	No	Definition of the status menu group in DashBoard.
config	menu-group Object (see page 4–58)	No	Definition of the config menu group in DashBoard
menu-group	menu-group object (see page 4–58)	No	Definition of additional menu groups. The menu group is named by the value of the string identifier `menu-group`.

param Object (descriptor)

Defines the value and descriptor of a parameter.

Syntax

```
"_d_OID" : {
                "oid": "oid",
                "name": "parameter name",
                "type": "parameter type",
                "parent": "parent oid and element index",
                "readonly": read-only flag,
                "widget": "parameter widget",
                "precision": parameter precision,
                "maxlength": maximum string length,
                "totallength": maximum total length of string array,
                "constraint": { constraint object },
                "config": { config object },
                "value": parameter value
            }
```

***Note**: The identifier of the object must be prepended with _d_ to indicate this object as a parameter descriptor.*

Member of

- params Object (see page 4–62)
- value Object (struct descriptor) (see page 4–64)

Members

Member	Type	Required	Description
oid	String	Yes	The Object Identifier (OID) for this parameter. The OID must be globally unique within the scope of the device.
name	String	No	Parameter Name.
type	String	Yes	The data type for the parameter. See "Data Types" on page 3–34 for a list of valid data types.
parent	string	No. This attribute is required if the device is responding to a panel's subscription request for an element inside of a Struct, such as a **STRUCT_ARRAY**.	This attribute tells the DashBoard Client which oid this parameter is a child of. The string value is the oid of the parent oid. **Note**: For Struct types, such as a **STRUCT_ARRAY**, the string value must also include the **index** of the parent element in the struct. For more information, refer to the device's **basic-param-info** response, which now includes a new metadata attribute that includes a parent-in-payload child attribute.
readonly	Boolean	No	If set true, parameter is read-only. Default is false.
widget	String	No	The widget used to display the parameter in DashBoard. If not specified, the default widget for the given type will be used. Note that widget types are specified by name, not their OGP numerical index.
precision	Number	No	This field defines the number of digits following the decimal point displayed for printed numbers. It applies mainly to floating point numbers.
constraint	constraint Object (see page 4–47)	No	Parameter Constraint. See "constraint Object" on page 4–47 for description of the constraint object.
config	config Object (see page 4–46)	No	Extended parameter configuration.
maxlength	Number	No	For STRING type: the maximum length of a string. For STRING_ARRAY type: the maximum string length of a single array member.
totallength	Number	No	For STING_ARRAY type: the maximum total characters in ALL members of the array.
value	see below	No	Value of the parameter. See discussion below for syntax of value member.

Value Field

The contents of the value field depend upon the setting of the **type** field.

type	value format
INT16 INT32 FLOAT32	Number
STRING	String
INT16_ARRAY INT32_ARRAY FLOAT32_ARRAY	Array of Number
STRING_ARRAY	Array of String
STRUCT STRUCT_ARRAY	value Object

Examples

This example describes an integer parameter with OID `0x500`.

```
"_d_0x500": {
                "oid": "0x500",
                "name": "Calibration Left",
                "type": "INT16",
                "widget": "spinner",
                "precision": 0,
                "constraint": {
                    "min": -100.0,
                    "display-max": 100.0,
                    "max": 100.0,
                    "display-min": -100.0,
                    "step": 1,
                    "type": "INT_STEP_RANGE"
                },
                "value": 69
            }
```

This example describes an integer array with the string OID `gainParamArray`:

```
"_d_gainParamArray": {
                "oid": "gainParamArray",
                "name": "Gain",
                "type": "INT16_ARRAY",
                "widget": "slider",
                "precision": 0,
                "constraint": {
                    "min": 0,
                    "display-max": 100.0,
                    "max": 100.0,
                    "display-min": 0,
                    "step": 1,
```

```
                    "type": "INT_STEP_RANGE"
                },
                "value": [
50,
50,
96,
12
    ]
            }
```

param Object (reference)

Defines a parameter which is a reference to another parameter. The parameter is a hard link to the referenced parameter, and thus inherits the type and all attributes and constraints of the referenced parameter. If the referenced parameter is an array type, the referencing parameter will also be an array. It is not possible to create an array of references.

Syntax

```
"_r_OID" : "referenced OID"
```

***Note**: The identifier of the object must be prepended with _r_ to indicate this object as a parameter reference.*

Member of

- params Object — (see page 4–62)

Members

Member	Type	Required	Description
_r_OID	String	Yes	`OID` is the object identifier for this parameter. Must be prepended with `_r_` to specify this is a reference parameter.
referenced OID	String	Yes	The OID of the referenced parameter.

Examples

This example creates a parameter called Key1.Clip which references the parameter with OID `0x500`.

```
"_r_Key1.Clip" : "0x500"
```

params Object

Defines the collection of parameters for a device.

Syntax

```
"params" : {
                "_d_OID1": { param object 1 },
                "_d_OID2": { param object 2 },
                "_d_OID3": { param object 3 },
        . . .
            }
```

Member of

- device Object — (see page 4–56)

Members

Member	Type	Required	Description
_d_OID	param Object (descriptor) (see page 4–59)	No	Parameter descriptor. Note that the name of the param objects must be pre-pended with _d_ to designate them as parameter descriptors.

value Object (struct descriptor)

The struct descriptor is contained within the value object of a parameter. The value field consists of an array of params, one for each member of the struct. Each sub-param may have its own type and constraints. The struct definition parameter must have a type of `STRUCT` or `STRUCT_ARRAY`.

Note: Descriptors are required only for the first (0th) element. Elements 1, 2, 3, etc can simply use `{"suboid1"."Value", "suboid2"."Value", ...}`.

Syntax

```
"_d_OID" : {
                "oid": "oid",
                "name": "parameter name",
                "type": "parameter type",
        "value": [
                    {
                "_d_sub-param_name": { param object },
                "_d_sub-param_name": { param object },
        . . .
                    }
                ]
        }
```

Member of

- param Object (descriptor) — (see page 4–59)

Members

Member	Type	Required	Description
oid	String	Yes	The Object Identifier (OID) for this parameter. The OID must be globally unique within the scope of the device.
name	String	No	Parameter Name
type	String	Yes	Set to `STRUCT`
`_d_sub-param_name`	param Object (descriptor) (see page 4–59)	Yes	Parameter descriptor for each sub-parameter. Note that the sub-param does not include an `oid` field. The `name` `sub-param_name` (minus the `_d_` prefix) is the name that is used to reference the struct member in an instance of this struct type.

Examples

The following param object defines a struct descriptor, with three sub-parameters (`LevelName`, `Color`, `Description`):

```
            "_d_types.level": {
                "oid": "types.level",
                "name": "Level (Type)",
```

```
                    "type": "STRUCT",
                    "widget": "default",
                    "value": [
                        {
                            "_d_LevelName": {
                                "name": "Level Name",
                                "type": "STRING",
                                "widget": "default",
                                "maxlength": 0,
                                "totallength": 0,
                                "value": "Default Name"
                            },
                            "_d_Color": {
                                "name": "Color",
                                "type": "INT32",
                                "widget": "color-picker-popup",
                                "precision": 0,
                                "constraint": {
                                    "min": -2147483648,
                                    "max": 2147483647,
                                    "type": "INT_STEP_RANGE",
                                    "unconstrained": true
                                },
                                "value": 16711680
                            },
                            "_d_Description": {
                                "name": "Description",
                                "type": "STRING",
                                "widget": "default",
                                "maxlength": 0,
                                "totallength": 0,
                                "value": "Default Description"
                            }
                        }
                    ]
                }
```

The following param object defines a struct, with three sub-parameter references (`Clip`, `Gain`, `Transparency`) to OIDs `0x501`, `0x502`, `0x503`:

```
                "_d_Keyer": {
                    "oid": "Keyer",
                    "name": "Keyer Parameters",
                    "type": "STRUCT",
                    "widget": "default",
                    "value": [
{
                            "_r_Clip": "0x501"
},
{
```

```
                        "_r_Gain": "0x502"
    },
    {
                        "_r_Transparency": "0x503"
        },
                    }
                ]
            }
```

value Object (inherited descriptor)

A parameter may inherit its struct descriptor (see page 4–64) from another parameter through use of a STRUCT constraint (see page 4–54).with specified templateoid. In this case, the values of struct members can defined in a param object by specifying a set of key-value pairs within the value object of the struct or struct-array. The key is the sub-parameter OID, and the value's type must match type of the sub-parameter.

Syntax (STRUCT)

```
            "_d_OIDn": {
                "oid": "parameter identifier",
                "name": "parameter name",
                "type": "STRUCT",
                "value": {
                    "sub-param name": "sub-param value",
                    "sub-param name": "sub-param value",
     . . .
                },
            "constraint" : { constraint object }
                }
```

Syntax (STRUCT_ARRAY)

```
            "_d_OIDn": {
                "oid": "parameter identifier",
                "name": "parameter name",
                "type": "STRUCT_ARRAY",
                "value": [
                    {
                        "sub-param name": "sub-param value",
                        "sub-param name": "sub-param value",
            . . .
                    },
                    {
                        "sub-param name": "sub-param value",
                        "sub-param name": "sub-param value",
            . . .
                    },
            . . .
                ],
            "constraint" : { constraint object }
```

```
}
```

Member of

- param Object (descriptor) — (see page 4–59)

Members

Member	Type	Required	Description
sub-param name	String	Yes	The Object Identifier (OID) for the sub-parameter
sub-param value	varies	Yes	Value assigned to the sub-parameter. Type is defined in the struct descriptor.

Sub-parameters are defined by a struct descriptor (see page 4–64). The value object need only specify sub-parameters whose values are explicitly being set. Unspecified sub-parameters will be unchanged for their inherited default values.

Examples

The following param object represents an array of structs. See struct descriptor definition in struct descriptor (see page 4–64) example.

```
"_d_params.levels": {
      "oid": "params.levels",
      "name": "Level List",
      "type": "STRUCT_ARRAY",
      "widget": "table",
      "value": [
          {
              "LevelName": "3G",
              "Color": -9137676,
              "Description": "The 3G Level"
          },
          {
              "LevelName": "HD",
              "Color": -5046403,
              "Description": "Not the 3G Level"
          },
          {
              "LevelName": "SD",
              "Color": -607985,
              "Description": "Also not the 3G Level"
          }
      ],
      "constraint": {
          "type": "STRUCT",
          "templateoid": "types.level",
          "description": "Descriptor for types.level",
          "structtype": "types.level"
      }
  }
```

command

Defines an OGP command for a device. OGP commands provide a way to use the OGP connection to execute commands from other devices. For more information, (see page 4–68).

Syntax

```
"command1":{
    "oid": "command1",
    "name": "command 1",
    "type": "STRUCT",
    "readonly": false,
    "widget": "default",
    "value": ...
        }
```

Member of

- commands — (see page 4–68)

Members

Member	Type	Required	Description
oid	String	Yes	The Object Identifier (OID) for this parameter. The OID must be globally unique within the scope of the device.
name	String	No	Parameter Name
type	String	Yes	The data type for the parameter. See "Data Types" on page 3–34 for a list of valid data types.
widget	String	No	The widget used to display the parameter in DashBoard. If not specified, the default widget for the given type will be used. Note that widget types are specified by name, not their OGP numerical index.
constraint	constraint Object (see page 4–47)	No	Parameter Constraint. See "constraint Object" on page 4–47 for description of the constraint object.
config	config Object (see page 4–46)	No	Extended parameter configuration.
value	param Object (descriptor) (see page 4–59)	No	Value of the parameter. See discussion below for syntax of value member.

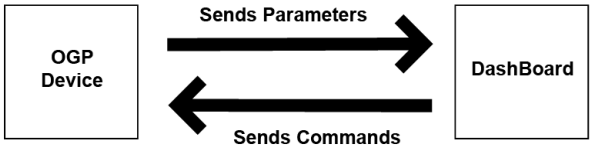
Examples

- See **commands** example—(see page 4–68)

commands

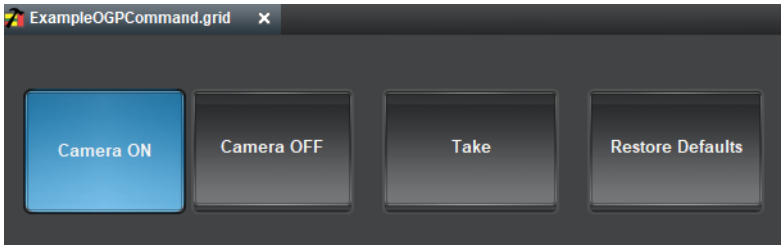
Defines the collection of OGP commands for a device. The OGP commands provide a way to use the OGP connection to execute commands from other devices. Many devices have useful commands, such as setting cross points, setting takes, and restoring defaults, that you can implement. These actions may include zero to many arguments (or parameters).

You can see an example diagram below:



The primary difference between using commands and parameters, is that the DashBoard OGP Client does not keep track of the state of the parameters in a command. The value of each parameter is specific to the execution request. This allows DashBoard to send multiple crosspoint command requests to the device and each one can have different values for the source/destination.

Once an OGP device has been added to DashBoard, you can use OGP commands to issue device commands directly from a CustomPanel. For example, the CustomPanel below shows a subset of a device commands that have been added to a CustomPanel. You can also create work flows using logic blocks in the Visual Logic Editor or editing the code directly in the ogScript Editor.



Syntax

```
"command1":{
                "oid": "command1",
                "name": "command 1",
                "type": "DATA_TYPE",
                "readonly": false,
                "widget": "default",
                "value": ...
              },
"command2":{
                "oid": "command2",
                "name": "command 2",
                "type": "DATA_TYPE",
                "readonly": false,
                "widget": "default",
                "value": ...
          }
```

Member of

- None

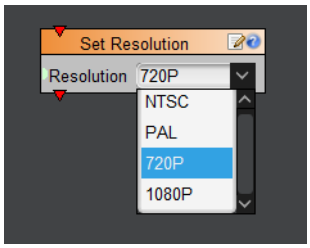
Members

Member	Type	Required	Description
oid	String	Yes	The Object Identifier (OID) for this parameter. The OID must be globally unique within the scope of the device.
name	String	No	Parameter name.
type	String	Yes	The data type for the parameter. See "Data Types" on page 3–34 for a list of valid data types.
widget	String	No	The widget used to display the parameter in DashBoard. If not specified, the default widget for the given type will be used. Note that widget types are specified by name, not their OGP numerical index.
constraint	constraint Object (see page 4–47)	No	Parameter Constraint. See "constraint Object" on page 4–47 for description of the constraint object.
config	config Object (see page 4–46)	No	Extended parameter configuration.
value	param Object (descriptor) (see page 4–59)	No	Value of the parameter. See discussion below the "param Object (descriptor)" on page 4–59 for the syntax of value member.

Examples

This example shows a command called "**SetResolution**" that has a "**Resolution**" argument that is constrained to the following choices: **NTSC**, **PAL**, **720P**, and **1080P**. The command is also shown in the **Visual Logic Editor** below.

Figure 4.2 *Visual Logic Representation of the Command*




```
"commands":{
    "SetResolution":{
        "oid":"SetResolution",
        "name":"Set Resolution",
        "readonly":false,
        "type":"STRUCT",
        "widget":"default",
        "value":[
            {
                "ResolutionOptions":{
                    "name":"Resolution",
                    "readonly":false,
                    "type":"STRING",
```

```
                    "widget":"text",
                    "maxlength":"0",
                    "totallength":"0",
                    "constraint":{
                        "value":"STRING_STRING_CHOICE",
                        "choices":[
                            {
                                "value":"NTSC",
                                "key":"NTSC"
                            },
                            {
                                "value":"PAL",
                                "key":"PAL"
                            },
                            {
                                "value":"720P",
                                "key":"720P"
                            },
                            {
                                "value":"1080P",
                                "key":"1080P"
                            }
                        ],
                        "strict":false
                    },
                    "value":"720P"
                }
            }
        ],
        "constraint":{

        },
        "response":true
    }
}
```

value (argument)

Defines an argument that can be passed to the command.

Syntax

```
"value":[
 {
     "argument1":{
         "name":"argument 1",
         "readonly":false,
         "type":"WIDGET_TYPE",
         "widget":"text",
```

```
            "maxlength":"0",
            "totallength":"0",
            "constraint":{
                "value":"PARAMETER_NAME",
                "choices":[
                    {
                        "value":"firstvalue",
                        "key":"firstvalue"
                    },
                    {
                        "value":"secondvalue",
                        "key":"secondvalue"
                    }
                ],
                "strict":false
            },
            "value":"firstvalue"
        }
    }
]
```

Member of

- commands — (see page 4–68)

Members

Member	Type	Required	Description
name	String	No	Parameter name.
type	String	Yes	The data type for the parameter. See "Data Types" on page 3–34 for a list of valid data types.
widget	String	No	The widget used to display the parameter in DashBoard. If not specified, the default widget for the given type will be used. Note that widget types are specified by name, not their OGP numerical index.
constraint	constraint Object (see page 4–47)	No	Parameter Constraint. See "constraint Object" on page 4–47 for description of the constraint object.
config	config Object (see page 4–46)	No	Extended parameter configuration.
value	param Object (descriptor)	No	Value of the parameter. See discussion below for syntax of value member.

Examples

This example shows a value argument that constrains a "`choices`" argument that is constrained to the following choices: `NTSC`, `PAL`, `730P`, and `1080P`.

```
"value":"STRING_STRING_CHOICE"
"choices":[
    {
        "value":"NTSC",
        "key":"NTSC"
    },
    {
        "value":"PAL",
        "key":"PAL"
    },
    {
        "value":"720P",
        "key":"720P"
    },
    {
        "value":"1080P",
        "key":"1080P"
    }
]
```

- For more information, see the full example for **commands**—(see page 4–68)

JSON Messages

Overview

JSON messages all follow a standard format. The basic structure of the JSON message is as follows:

```
{
       “type” : messageType,
       “slot” : slotID,
       “payload” : {
          MessagePayload
       }
}
```

Slot ID

Each message is addressed to a specific slot ID on the device. In openGear frames, the Slot ID refers to the physical card slot within the frame. Most DashBoard Connect devices need only implement a single slot (slot 0).

Message Type

The type field specifies the type of message. The following message types are supported:

Member	Type
device Message (see page 4–74)	Reports a device definition.
device-request Message (see page 4–75)	Requests a device definition.

Member	Type
eo Message (see page 4–76)	Returns an external object.
eo-request Message (see page 4–78)	Requests an external object.
handshake Message (request) (see page 4–79)	Initial connection handshake request.
handshake Message (response) (see page 4–80)	Initial connection handshake response.
hide Message (see page 4–81)	Requests DashBoard to hide a UI element.
multi-value Message (see page 4–82)	Reports multiple parameter values.
param Message (see page 4–84)	Reports a parameter descriptor.
param-request Message (see page 4–84)	Requests a parameter descriptor.
reload Message (see page 4–85)	Requests Dashboard to rebuild a UI element.
restart Message (see page 4–86)	Notification that a device is restarting.
restart-request Message (see page 4–86)	Requests a device to restart.
reveal Message (see page 4–87)	Requests DashBoard to reveal (un-hide) a UI element.
value Message (see page 4–88)	Reports a parameter value.
value-request Message (see page 4–89)	Requests a parameter value.

Payload

The payload field consists of an object whose content is dependent on the message type. See individual message descriptions below. This field may be omitted in some message types.

device Message

Reports all information for a device.

Syntax

```
{
        "type": "device",
        "slot": slotID,
        "payload" : {
                  "slot": slot id,
```

```
                "detail": { level of detail }
                "menu-groups": { menu-groups object },
                "params": { params object }
               "multi-set-enabled": multi-set-flag
        }
}
```

Message Fields

Field	Type	Required	Description
type	String	Yes	Set to `device`.
slot	Number	Yes	Destination slot for this message
payload	device Object (see page 4–56)	Yes	Device description. To see the fields for the payload, see (see page 4–56)
success	Boolean	No	Indicates if the message was accepted by the recipient of the device-request message.

Related to

- device-request Message — (see page 4–75)

device-request Message

Requests all information for a device. DashBoard sends a `device-request` message to a device upon initial successful connection to the device.

Syntax

```
{
        "type": "device-request",
        "slot": slotID
        “payload”: {
        “detail”: level of detail requested
}
```

Message Fields

Field	Type	Required	Description
type	String	Yes	Set to `device-request`
slot	Number	Yes	Slot for which device’s information is being requested.
payload	device Object	No	Device description. To see the fields for the payload, see (see page 4–4).
detail	String	No	Indicates the level of detail requested in the message. Set to `minimal` or `full`. The default setting is `full`. For more information on OGP minimal mode (see page 4–56).

Related to

- device Message — (see page 4–74)

eo Message

Returns an external object. DashBoard supports two encoding schemes for external object data; data may encoded as a base64 string within the JSON message or alternatively, the data may be appended as raw binary data utilizing the modified text/binary Netstrings Container (see page 4–1).

Syntax

Using Base64 encoding:

```
{
        "type": "eo",
        "slot": slotID,
        "payload" : {
        "id" : id,
        "type": object-type,
        "base64": "base64-data"
}
}
```

Using NetString text/binary encoding:

```
{
        "type": "eo",
        "slot": slotID,
        "payload" : {
        "id" : "id",
        "type": object-type,
}
}binary-data
```

Message Fields

Member	Type	Required	Description
type	String	Yes	Indicates the type of external object. 1 = Constraint 2 = File Download 3 = Image 4 = OGLML / Index XML document
slot	Number	Yes	Destination slot for this message.
id	Number	Yes	The identifier for the external object.
Type	Number	Yes	The external object type. See "External Data Objects" on page 3–35 for details about object types.
base64	Base64 String	Yes	BIN64 encoded external object data. for more information about base64 encoding, see http://en.wikipedia.org/wiki/Base64.
binary-data	Raw Binary Data	Yes	Raw binary data external object content.

Data Formatting

The External Object data (regardless of the transport encoding) must be formatted as follows, depending upon the type of external object.

Type 1 - Constraint:

Field	Length	Format	Description
Ctype	1	uint8	constraint type
Clen	1	uint8	length of constraint data in bytes (ignored). This is here for compatibility with embedded constraint. If constraint data is longer than 255 bytes, this field is ignored by DashBoard.
Cdata	varies	*	ctype-specific constraint data

Type 2 – File Upload:

Field	Length	Format	Description
hlen	2	uint16	header length (variable length – number of bytes between this field and the 'content' section including the hlen field itself)
ver	1	uint8	version of file encapsulation encoding used (0x01)
minver	1	uint8	minimum decoder version required to understand the encapsulation (0x01)
no-cache	1	uint8	• 0x00 - Cache this external object data (static data) • 0x01 - Do not cache external object data (dynamically generated data)
header type	1	uint8	• 0x00 – Do not generate a header (content will follow immediately) • 0x01 – Tail header (a file header will follow at the end of the content) • 0x02 – Generate header (DashBoard will generate an upload header based on the information provided in the following fields)
APPLIES ONLY TO HEADER TYPE 1 (Tail header)			
taillen	2	uint16	The number of bytes at the end of the content to move to the beginning of the file to use as a header.
APPLIES ONLY TO HEADER TYPE 2 (Generate header)			
load type	1	uint8	See ""File Format for OGP Upload" on page 7–3.
bit order	1	uint8	
version	2	uint16	
compat	4	uint16	
target	4	uint32	
pcount	2	uint16	
product[]	14*16	String	
APPLIES TO ALL			
content	dlen	byte* (byte array)	data to be uploaded (i.e. the original file)

Type 3- Image:

Field	Length	Format	Description
hlen	2	uint16	Header length (currently 2 but may be extended to include more header fields in the future)
Hide status overlay	1	uint8	If image is used as device icon, indicates if DashBoard may draw a Status Indicator over the Icon. • 0 = DashBoard will draw a status indicator over the icon. • 1 = DashBoard should not draw a status indicator over the icon (the icon provides its own status information)
Image type	1	uint8	• 0 = Image data is embedded in the content section • 1 = Content section contains a URL for the icon
content	*	varies depending on Image type: - if 0, binary data - if 1, ASCII data	• If image type = 0 — The binary image data encoded as a JPEG, GIF, or PNG. • If image type = 1 — length (uint16) followed by null-terminated UTF-8-Encoded URL for the image location.

Type 4 – OGLML or Index XML Document:

Field	Length	Format	Description
desctype	1	uint8	How to interpret the following content: • 0 = does not reference an OGLML document (end of descriptor) • 1 = contains the ID of another external object (followed by uint16) • 2 = contains the URL to fetch the external object via http or https (UTF-8 String) • 3 = contains the OGLML document itself as an XML file • 4 = contains the OGLML document as a zipped XML file (gzip and deflate supported)
content	*	varies depending on desctype value	Contains the content based on the above desctype field.

Related To

- eo-request Message — (see page 4–78)

eo-request Message

Requests an external object.

Syntax

```
{
        “type”: "eo-request",
        “slot”: slotID
        "payload" : {
```

```
        "oid" : "oid",
}
}
```

Message Fields:

Field	Type	Required	Description
type	String	Yes	Set to `value-request`
slot	Number	Yes	Slot for which device information is being requested.
oid	String	Yes	The Object Identifier (OID) for the requested external object.

Related To

- eo Message — (see page 4–76)

Examples

handshake Message (request)

The handshake request from DashBoard to a device. Upon initial connection to a device, DashBoard will send a handshake request. This provides opportunity for a device to reject a connection gracefully if security criteria are not met of connection count is exceeded. If DashBoard has an active connection to a URM server, it will indicate a trusted connection; otherwise it will provide a password if one has been specified. DashBoard may also request to force a connection to the device, in which case a device may disconnect other devices if it has exceeded its connection limit. DashBoard may set the desired level of detail for device updates to either minimal or full.

Syntax

```
{
        "type": "handshake",
        "slot": slotID,
        "payload" : {
        "trusted" : trusted flag,
        "password" : "device password",
        "force" : force connection flag,
        "detail" : “level of detail supported for device updates”,
        "build" : "major.minor.micro YYYY-MM-DD HH:MM"

}
}
```

Message Fields

Field	Type	Required	Description
type	String	Yes	Set to `handshake`
slot	Number	Yes	Destination slot for this message
trusted	Boolean	No	Indicates whether DashBoard has a connection to URM server • `true`: Trust the connection (DashBoard is able to contact its user rights server) • `false`: Authenticate the connection with the password specified in the **password** field, if specified.
password	String	No	If a password is required for connection, DashBoard will specify it here. Password is specified in cleartext. Not required if **trusted** is `true`.
force	Boolean	No	If `true`, the device must accept the connection from DashBoard, even if its maximum connection count has been reached. The device may disconnect other clients if necessary.
detail	String	No	The desired default level of detail for device updates. DashBoard sets the detail level to `minimal` by default. **Note:** The level of detail can also be set to `full` if minimal mode updates are disabled.
build	String	No	Dashboard build version and timestamp

Related to

- handshake Message (response) — (see page 4–80)

handshake Message (response)

Response by a device to a handshake request from DashBoard. This provides opportunity for a device to reject a connection gracefully if security criteria are not met of connection count is exceeded. The handshake response can indicate if the device supports minimal or full for the level of detail.

Syntax

```
{
        "type": "handshake",
        "slot": slotID,
        "payload" : {
        "disconnect" : disconnect flag,
        "reason" : "reason response"
        "detail": level of detail supported}
}
```

Message Fields

Field	Type	Required	Description
type	String	Yes	Set to `handshake`
slot	Number	Yes	Slot of the reporting device.
disconnect	Boolean	No	If `true`, the device refuses the DashBoard connection.
reason	String	No	If the connection is refused, this property will be specified with a reason string: • `bad-password`: the supplied password is incorrect. • `urm-required`: a trusted connection is required, but was not provided. • `refused`: connection is refused for other reason (e.g. connection limit reached)
detail	String	No	The desired default level of detail for device updates. DashBoard sets the detail level to `minimal` by default. **Note:** The level of detail can also be set to `full` if minimal mode updates are disabled.

Related to

- handshake Message (request) — (see page 4–76)

Examples

hide Message

Hides a UI container element in DashBoard. **hide** may only be applied to pop-up or abs elements. For pop-up elements, the pop-up will be closed. For abs elements, the visibility of the abs will be disabled. If the element is already hidden, this message has no impact.

Syntax

```
{
        "type": "hide",
        "slot": slotID,
        "payload" : {
        "id" : "element id",
        "id" : "element id",
        . . .
}
}
```

Message Fields

Field	Type	Required	Description
type	String	Yes	Set to `hide`
slot	Number	Yes	Destination slot for this message
id	String	Yes	ID of the element to hide. Multiple elements may be hidden in a single **hide** message.

Related to

- reveal Message — (see page 4–87)

multi-value Message

Reports multiple parameter values. Requires the attribute `multi-set-enabled` be set `true` in the device object; DashBoard will only send `multi-value` message through ogScript `MultiSet` methods. If `multi-set-enabled` is false, DashBoard will break up value changes to independent `value Messages` (see page 4–88).

Device may also send multi-value message to DashBoard.

Syntax

```
{
        "type" : "multi-value",
        "slot" : slotID,
        "payload" : [
                {
                "oid"  : "oid",
                "element" : is array element,
                "index" : array index,
                "value" : parameter value
                },
                {
                "oid"  : "oid",
                "element" : is array element,
                "index" : array index,
                "value" : parameter value
                },
            . . .
        ]
}
```

Message Fields

Field	Type	Required	Description
type	String	Yes	Set to `value`
slot	Number	Yes	Destination slot for this message
oid	String	Yes	The Object Identifier (OID) for this parameter.
element	Boolean	No	Indicates if this parameter is an element of an array. Default is `false`.
index	Number	No	The array index of the parameter. Only valid if `element` is `true`.
value	varies For more information, see "param Object (descriptor)" on page 4–59.	Yes	Value of the parameter.

Related To

- value-request Message — (see page 4–89)
- value Message — (see page 4–88)

Examples

The following message reports the value of a string parameter with the following OIDs: 0x534, 0x506, 0x535:

```
{
    "payload": [
        {
            "oid": "0x534",
            "value": 1.0,
            "element": true,
            "index": 0
        },
        {
            "oid": "0x506",
            "value": 1.0,
            "element": true,
            "index": 0
        },
        {
            "oid": "0x535",
            "value": 1.0,
            "element": true,
            "index": 0
        }
    ],
    "slot": 0,
    "type": "multi-value"
}
```

param Message

Reports a parameter descriptor.

Syntax

```
{
        “type” : "param",
        “slot” : slotID,
        “payload” : { param object }
}
```

Message Fields

Field	Type	Required	Description
type	String	Yes	Set to `param`
slot	Number	Yes	Destination slot for this message
payload	param Object (descriptor) — (see page 4–59)	Yes	Parameter descriptor object.

Related to

- param-request Message — (see page 4–84)

Examples

The following message reports the descriptor for OID 0x500.

```
{
   "type":"param",
   "slot":0,
   "payload":{
      "oid":"0x500",
      "name":"Calibration Left",
      "type":"INT16",
      "widget":"spinner",
      "precision":0,
      "constraint":{
         "min":-100.0,
         "display-max":100.0,
         "max":100.0,
         "display-min":-100.0,
         "step":1,
         "type":"INT_STEP_RANGE"
      },
      "value":69
   }
}
```

param-request Message

Requests a parameter descriptor. DashBoard will send a `param-request` message to a device to receive an updated a param descriptor.

Syntax

```
{
        "type" : "param-request",
        "slot" : slotID,
        "payload" : {
            "oid" : "oid"
            }
}
```

Message Fields

Field	Type	Required	Description
type	String	Yes	Set to `device-request`
slot	Number	Yes	Slot for which device information is being requested.
oid	String	Yes	The Object Identifier (OID) for the requested parameter.

Expected Response

- param Message — (see page 4–84)

Response To

None

Examples

reload Message

Requests DashBoard to rebuild the named UI element in the UI. This message applies to devices and/or individual menus defined with OGLML documents. Multiple elements can be listed in the same message. If certain parts of an OGLML document are dynamically generated or the OGLML document has changed, `reload` will allow those changes to be reflected in the UI.

Syntax

```
{
        "type" : "reload",
        "slot" : slotID,
        "payload" : {
            "id" : "element id"
            ,
            "id" : "element id"
            ,
            . . .
            }
}
```

Message Fields

Field	Type	Required	Description
type	String	Yes	Set to `reload`
slot	Number	Yes	Destination slot for this message
id	String	Yes	ID of the element to reload. Multiple elements may be revealed in a single **reload** message.

Related To

- hide Message — (see page 4–81)
- reveal Message — (see page 4–87)

Examples

restart Message

Notifies DashBoard that a device has restarted. This notification must be sent after the device has booted up. It may also be sent whenever the parameter or menu descriptors for the device have changed significantly. DashBoard will typically send a device-request Message (see page 4–75) upon receipt of a `restart` message from the device.

Syntax

```
{
        "type" : "restart",
        "slot" : slotID
}
```

Message Fields

Field	Type	Required	Description
type	String	Yes	Set to `restart`
slot	Number	Yes	Slot ID of the restarted device

Related To

- restart-request Message — (see page 4–86)

Examples

The following example shows a restart message for a device in slot 0.

```
{
        "type" : "restart",
        "slot" : 0
}
```

restart-request Message

Requests a device to reboot.

Syntax

```
{
        "type" : "restart-request",
        "slot" : slotID
```

```
}
```

Message Fields

Field	Type	Required	Description
type	String	Yes	Set to `restart-request`
slot	Number	Yes	Slot ID of the device to be rebooted.

Related To

- restart Message — (see page 4–86)

Examples

The following example shows a restart-request message for a device in slot 0.

```
{
        "type" : "restart-request",
        "slot" : 0
}
```

reveal Message

Reveals a UI container element in DashBoard. **reveal** may only be applied to pop-up, abs, or tab elements. For pop-up elements, the pop-up will be expanded. For abs elements, the visibility of the abs will be enabled. For tab elements, the tab will be brought forward. If the element is already revealed, this message has no impact.

*Note: If the specified element is contained within a hidden container (for example, a de-selected tab element), **reveal** will not reveal the parent container implicitly; it is necessary to also explicitly **reveal** the parent container(s).*

Syntax

```
{
        "type" : "reveal",
        "slot" : slotID,
        "payload" : {
            "id" : "element id"
            ,
            "id" : "element id"
            ,
            . . .
            }
}
```

Message Fields

Field	Type	Required	Description
type	String	Yes	Set to `reveal`
slot	Number	Yes	Destination slot for this message
id	String	Yes	ID of the element to reveal. Multiple elements may be revealed in a single **reveal** message.

Related To

- hide Message — (see page 4–81)

value Message

Reports a parameter value.

Syntax

```
{
        "type" : "value",
        "slot" : slotID,
        "payload" : {
            "oid"  : "oid",
            "element" : is array element,
            "index" : array index,
            "value" : parameter value
            }
}
```

Message Fields

Field	Type	Required	Description
type	String	Yes	Set to `value`
slot	Number	Yes	Destination slot for this message
oid	String	Yes	The Object Identifier (OID) for this parameter.
element	Boolean	No	Indicates if this parameter is an element of an array. Default is `false`.
index	Number	No	The array index of the parameter. Only valid if **element** is `true`.
value	varies For more information, see “param Object (descriptor)” on page 4–59.	Yes	Value of the parameter.

Related To

- value-request Message — (see page 4–89)

Examples

The following message reports the value of a string parameter with OID 0x500:

```
{
        "type" : "value",
        "slot" : 0,
        "payload" : {
            "oid"  : "0x500",
            "value" : "This is a new string"
            }
}
```

The following message reports the value of an element of string array parameter with OID 0x505:

```
{
        "type" : "value",
```

```
        "slot" : 0,
        "payload" : {
           "oid"  : "0x505",
           "element": true,
           "index": 5,
           "value" : "This is a new string for array index 5"
           }
}
```

The following message reports the value of all elements of an integer array with OID key.clip:

```
{
        "type" : "value",
        "slot" : 0,
        "payload" : {
           "oid"  : "key.clip",
           "value" : [ 40, 50, 40, 62 ]
           }
}
```

The following example reports the value of a struct parameter with OID `RouterLevel`:

```
{
        "type" : "value",
        "slot" : 0,
        "payload" : {
           "oid"  : "RouterLevel",
              "value" :  {
                      "LevelName": "3G",
                      "Color": -9137676,
                      "Description": "The 3G Level"
                  }
        }
}
```

value-request Message

Requests a parameter value.

Syntax

```
{
        "type" : "value-request",
        "slot" : slotID
        "payload" : {
           "oid"  : "oid",
           "element" : is array element,
           "index" : array index
           }
}
```

Message Fields

Field	Type	Required	Description
type	String	Yes	Set to `value-request`
slot	Number	Yes	Slot for which device information is being requested.

Field	Type	Required	Description
oid	String	Yes	The Object Identifier (OID) for the requested parameter.
element	Boolean	No	Indicates if this parameter is an element of an array. Default is `false`.
index	Number	No	The array index of the parameter. Only valid if **element** is `true`.

Related To

- restart Message — (see page 4–88)

OGP Messaging

In This Chapter

This chapter summarizes the general structure of the OGP messaging protocol.

The following subjects are discussed:

Initiating Connection to OGP Devices

Connection Handshake

When DashBoard opens a TCP/IP connection, it sends an OGP_PARAM_SET_REQUEST for a special OID: 0xFF03. This message will be addressed to address 0x10. DashBoard will continue processing messages from the device normally until it receives a CONNECTION_OGP_SET_PARAM Response.

Connection Handshake is only required for devices that communicate directly with DashBoard over TCP/IP. Individual cards within an openGear frame do not need to concern themselves with this, as the frame controller handles it.

***Note**: The response to this message must only be sent to the DashBoard that sent the request, rather than the normal convention of replying to all connected DashBoards.*

The values of the parameter and response are defined below.

Initial Parameter Query

Once connection has been established, DashBoard will query the device for its parameters.

The parameter-querying sequence is as follows:

1. DashBoard sends a single **OGP_GET_NUMPARAMS** request to the device to request the number of parameters. Device sends a valid **OGP_GET_NUMPARAMS** response with the number of parameters.
2. DashBoard will send one or multiple **OGP_GET_PARAM_OIDS** requests; each request may ask for up to 128 OIDs. Device replies with valid **OGP_GET_PARAM_OIDS responses.**
3. For each OID defined in the list:
   a. Fetch the descriptor using the **OGP_GET_DESCRIPTOR message.**
   b. In case of an external constraint, fetch the constraint.
   c. Fetch the current value of the parameter
   d. As certain special reserved OIDs are encountered, extra things may happen
4. While there are more menu groups:
   a. Request name of menu group
   b. Request number of menus in the group
   c. for each menu group (group 0, followed by group 1):
      - request menu group name (send OPG_GET_MENUSET_NAME request)
      - request number of menus (send OPG_GET_MENU_COUNT request)

- for each menu (0 <= menu < menu_count)
  - › request menu name (send OPG_GET_MENU_NAME request)
  - › request menu oids (send OPG_GET_MENU_OIDS request)
  - › request menu state (send OPG_GET_MENU_STATE request)

In order to make the initial parameter query more efficient, a device may begin broadcasting its list of parameters to DashBoard immediately after the **CONNECTION_OGP_SET_PARAM** is sent. In doing so, it will reduce the time to update DashBoard with the parameter and menu definitions by reducing the number of request/response messages between DashBoard and device. This broadcast should only be done once immediately after connection to DashBoard. For details about asynchronous notification messages, see "Asynchronous Messaging" on page 6–3.

***Note**: It is strongly recommended that devices broadcast all parameters to DashBoard upon initial connection, without waiting for DashBoard requests.*

Menu-Definition Protocol

The menu-definition protocol includes the following requests and their associated responses:

Command	Msg type	Description
OPG_GET_MENUSET_NAME	0x50	request the name of a menu group
OPG_GET_MENU_COUNT	0x51	request the number of menus in a group
OPG_GET_MENU_NAME	0x52	request the name of a menu
OPG_GET_MENU_OIDS	0x53	request the object IDs for a menu
OGP_GET_MENU_STATE	0x5A	request the display state for a menu (optional)
OGP_GET_MENU_STRING_OIDS	0x73	request the string OIDs for a menu ***Note**: String OID messages are not supported on CANbus connections*

For each menu group (group 0, followed by group 1), the menu-definition query sequence is as follows:

1. request menu group name (send OPG_GET_MENUSET_NAME request)
2. request number of menus (send OPG_GET_MENU_COUNT request)
3. for each menu (0 <= menu < menu_count)
   a. request menu name (send OPG_GET_MENU_NAME request)
   b. request menu oids (send OPG_GET_MENU_OIDS request)
   c. request menu state (send OPG_GET_MENU_STATE request)

The device must provide a valid response to each request.

OPG_GET_MENU_STATE is an optional feature. Devices that do not support this feature should return OGP_UNSUPPORTED; all menus will be set to the default state (visible and enabled).

Messaging Overview

Addressing Conventions

Every OGP message must include both a source address (the sender of the message) and a destination address (the intended recipient). OGP addresses are primarily used to route messages to the appropriate card within an openGear frame. The address is based upon the slot ID for a given device. TCP/IP DashBoard Connect devices are generally addressed as a card in Slot 0.

OGP addresses are 6-bit values, assigned as follows:

Device	Address
DashBoard	0x00
Fan Control Card (Slot 0)	0x10
Frame Network Controller (Slot 0)	0x10
TCP/IP DashBoard Connect Devices	0x10
openGear Cards	0x10 + SlotID

During initial TCP/IP handshake, DashBoard sends messages to Destination Address 0x10 (Slot 0).

In addition, the following broadcast addresses are defined:

Device	Value	Description
OGP_ADDR_PRINT	0x01	debug messages broadcast by cards
OGP_ADDR_TRAP	0x02	status messages broadcast by cards
OGP_ADDR_UPLOAD	0x3E	reserved for future use
OGP_ADDR_BCAST	0x3F	from openGear controller to all cards
Reserved	0x03 – 0x0F 0x31 – 0x3D	Reserved

Reception

Each card must receive and process all messages which are addressed to its own unique bus address or to OPG_ADDR_BCAST. All other messages must be ignored. Some CAN controllers permit this filtering to be performed in hardware.

Transmission

Each card may send debug and print messages to OPG_ADDR_PRINT. Change notifications and status messages must be sent to OPG_ADDR_TRAP. Responses to request messages must be sent to the address from which the request was received (i.e. the source address of the request). All messages must be tagged with the unique source address of the sender. A broadcast address is not a legal source address.

***Note**: Currently, card-to-card addressing is not supported, only card-to-controller, controller-to-card or broadcast may be used.*

Request / Response Model

The primary interaction between DashBoard and device is implemented using a point-to-point request-response model. DashBoard sends a request to a specific device, and the target device must return a response containing an acknowledgment or the requested data. This allows DashBoard to confirm that the request has been received and acted on. A device is always free to decline the action specified by a request, but it must reply to DashBoard in any case.

Requests have a message type in the range 0x20 to 0x7F. The message type for the corresponding response is obtained by setting the most significant bit of the message type, that is:

```
Response_type = request_type | 0x80
```

The device must respond to each request even if the request is not supported. The response must be sent to the address specified as the source address of the request. The message type of the response must be determined from the request type using the above formula. The first byte of each response is a return code with the following convention:

Response	Value	Description
OGP_OK	0	Request was successful
OGP_UNSUPPORTED	1	The device does not understand the request type
Other	Other	Other non-zero values indicate request-specific errors

Unless specified otherwise, the device must respond to each request within 500 milliseconds. If DashBoard does not receive a response within 1 second, it may send the request again (to a maximum of 3 attempts). Devices that do not respond to a request are assumed to be offline. Some transactions, such as file upload, have different timing requirements. See message definitions for details.

When a device responds to a message with an error code, it is insufficient for it to send only the error code with the message type and response flag. A response message must include enough information for DashBoard to correctly identify the request message. If the device does not recognize the message type, it may send back the entire original message with the response flag set and the return code changed to OGP_UNSUPPORTED.

If the device recognizes the message type, it can send a smaller amount of data in response. For all parameter messages, sending the response flag, message type, return code, OID, and Index (if applicable) is enough. When in doubt, resend the entire message content.

Broadcast Model

A device may send unsolicited messages at any time to one of the broadcast addresses defined above (OGP_ADDR_PRINT or OGP_ADDR_TRAP). Messages sent to a broadcast address are not acknowledged, so delivery is not guaranteed. In practice, however, the probability of losing a broadcast message is very low.

Devices send unsolicited messages to:

- report operating status periodically (send to OGP_ADDR_TRAP)
- report asynchronous parameter changes (send to OGP_ADDR_TRAP)
- report changes to parameter definitions or menu structure (send to OGP_ADDR_TRAP)
- report debug/print messages (send to OGP_ADDR_PRINT)

The openGear Frame Network Controller or DashBoard may broadcast messages to all devices using the broadcast address OGP_ADDR_BCAST. This is typically used to send system information such as time. Devices must not respond to broadcast messages.

Alarms and Status

Device status, errors and warnings are reported to DashBoard via parameters with the ALARM_TABLE constraint. The highest level active alarm (i.e. the most severe) on a device is presented in the DashBoard tree view through a colored icon and status message tool-tip.

Heartbeat

Each openGear card should send at least one message per second as a keep-alive. This should normally be an OGP_REPORT_PARAM message reporting the value of a status parameter (for example OID 0x0105, PRODUCT_NAME). During normal operating conditions, each device must send no more than 5 unsolicited messages per second.

Basic Messaging

Required Messages

OGP messaging is a simple request-response protocol. Transport definition for OGP is described in detail in the section, "Transport Layer" on page 2–1.

There are a few basic messages that a device needs to implement in order to implement the basic parameter-passing protocol. These basic messages are as follows:

Command	msg_type	Description
OGP_GET_NUMPARAMS	0x45	Request the number of parameters on the device
OGP_GET_PARAM_OIDS	0x46	Request the table of parameter OIDs
OGP_GET_DESCRIPTOR	0x47	Request the descriptor for a parameter
OGP_GET_PARAM_NAME	0x48	Request the name of a parameter
OGP_GET_PARAM	0x49	Request the value of a parameter
OGP_SET_PARAM	0x4A	Change the value of a parameter

Required Parameters

The list of parameters for each device is largely determined by the specific requirements and functionality of that device. However, the following parameters are required for operation with DashBoard and the SNMP agent:

Parameter Name	OID	Type	Description
OID_PRODUCT_NAME	0x0105	String	Product Type Name
OID_SUPPLIER	0x0102	String	Supplier Name
OID_SOFTWARE_REV	0x010B	String	Software Revision For more information, see "Version Number Encoding" on page 7–4.

***Note**: Several other OIDs are required to use features within DashBoard, DataSafe, and SNMP. A full list of reserved OIDs is provided in "Appendix D: Reserved Object IDs" on page 11–6.*

Participation Rules

In order to interact well with DashBoard and other devices, devices must comply with the following rules:

1. Request Handling
   - The device must respond within 500 milliseconds to every request message, except those that specifically allow a slower response.
   - The device must respond appropriately to all parameter and menu-related requests, except those specifically defined to be optional.
   - The device must respond appropriately to requests whenever they are received (because there may be multiple clients sending different requests concurrently).
   - For any unsupported request, the device must send a response with:
     - message type = request type | 0x80
     - message length = same as message received
     - message content = 0x01 (OGP_UNSUPPORTED) followed by remaining source bytes
   - For any response message with return code 0x00 (OGP_OK), the message content must be valid.
   - If a message type is not supported, the device should return OGP_UNSUPPORTED for all requests that are specific to that type.
   - Response error codes:
     - If the message type is unknown, return OGP_UNSUPPORTED (0x01)
     - If the message length is incorrect, return OGP_INVALID_LENGTH (0x02)
     - If the request is for an OID that is not defined, return OGP_PARAM_NOTFOUND (0x11)
     - If the request is for an array parameter and the index is out-of-bounds, return OGP_PARAM_BADINDEX (0x15)
     - If the request is an attempt to set a parameter, but the parameter is read-only, return OGP_PARAM_READONLY (0x13)
     - If the request is an attempt to set a parameter, but the value is illegal (outside of the constraint), return OGP_PARAM_BADVAL (0x12)
     - If a simple request (e.g. OGP_REBOOT) is refused and an error message is supplied, return OGP_REQUEST_DENIED followed by the error message Null-Terminated UTF-8 String.
2. Message Structure and Addressing:
   - All messages transmitted on the bus must be well-formed OGP messages.
   - All messages must be tagged with the correct source address of the sending device.
   - All broadcast messages must be sent to either OGP_ADDR_PRINT or OGP_ADDR_TRAP.
   - All response messages must be sent to the address from which the request was received.
   - Devices should not communicate directly with other devices.
3. Broadcast Messages:
   - Each device must broadcast an OGP_RESTART trap on bootup.
   - Each device must broadcast a message at least once every 5 seconds to support automatic detection of online devices. This should normally be an OGP_REPORT_PARAM message.
   - The device must not respond to any broadcast message received.
4. For openGear cards in an openGear frame:
   - All transmission on the CANbus must be at 1.0 Mbps.
   - All transmission on the bus must follow the extended CAN standard 2.0B.
   - Each CAN frame must be encoded as described in the section, "OGP via CAN" on page 2–2.
   - During normal operation, each device must not send more than five messages per second on average.

This does not apply when responding to GET/SET requests from DashBoard; those replies should be sent as quickly as possible.

- No individual parameter should be reported more than once per second.

OGP Reference

In This Chapter

This chapter describes the parameter management application protocol within OGP. It outlines how parameters are defined, constrained and updated.

The following subjects are discussed:

Application Protocols

Parameter Management

Command	Msg type	Description
OGP_GET_NUMPARAMS	0x45	Request the number of parameters on the device
OGP_GET_PARAM_OIDS	0x46	Request the table of numeric parameter OIDs
OGP_GET_DESCRIPTOR	0x47	Request the descriptor for a parameter
OGP_GET_PARAM_NAME	0x48	Request the name of a parameter
OGP_GET_PARAM	0x49	Request the value of a parameter
OGP_SET_PARAM	0x4A	Change the value of a parameter
OGP_GET_ARRAY_ELEMENT	0x4C	Requests one element of an array parameter
OGP_SET_ARRAY_ELEMENT	0x4D	Change one element of an array parameter
OGP_GET_EXTERNAL_OBJECT	0x59	Requests a generic data object
OGP_GET_STRING_OIDS ***Note**: String OID messages are not supported on CANbus connections.*	0x66	Request the table of string OIDs.
OGP_GET_STRING_DESCRIPTOR ***Note**: String OID messages are not supported on CANbus connections.*	0x67	Request the descriptor for a string OID
OGP_GET_STRING ***Note**: String OID messages are not supported on CANbus connections.*	0x69	Request the value of a string OID parameter
OGP_SET_STRING ***Note**: String OID messages are not supported on CANbus connections.*	0x6A	Change the value of a string OID parameter
OGP_GET_STRING_ARRAY_ELEMENT ***Note**: String OID messages are not supported on CANbus connections.*	0x6C	Requests one element of an array parameter
OGP_SET_STRING_ARRAY_ELEMENT ***Note**: String OID messages are not supported on CANbus connections.*	0x6D	Change one element of an array parameter

Menu Definition

Command	Msg type	Description
OGP_GET_MENUSET_NAME	0x50	Request the name of a menu group
OPG_GET_MENU_COUNT	0x51	Request the number of menus in a group
OPG_GET_MENU_NAME	0x52	Request the name of a menu
OPG_GET_MENU_OIDS	0x53	Request the object IDs for a menu
OPG_GET_MENU_STATE	0x5A	Request the display state for a menu (optional)

File Upload

Command	Msg type	Description
OGP_START_UPLOAD	0x40	Initiates a file upload
OGP_UPLOAD_PAGE	0x41	Uploads a 256-byte page of the file
OGP_VERIFY_UPLOAD	0x42	Prompts the card to verify the CRC
OGP_REBOOT	0x43	Instructs the device to reboot

See Also: *"File Upload" on page 7–1.*

SNMP

Command	Msg type	Description
OGP_GET_SNMP_BASE	0x56	Requests the card MIB OID
OGP_GET_SNMP_OID	0x57	Request the SNMP OID for a parameter
OGP_GET_SNMP_TRAP	0x58	Requests the SNMP trap OID for a parameter

See Also: *"SNMP" on page 9–1.*

Other Applications

Command	Msg type	Description
OGP_COMMAND	0x44	Send a command to the card
OGP_PRINT	0x00	encodes an printable string or debug message
OGP_REPORT_PARAM	0x10	Report the value of a parameter
OGP_TRAP	0x11	Notifies clients of a change in the device setup
OGP_BOOTLOAD	0x13	Indicates the device bootloader is running

Asynchronous Messaging

When DashBoard becomes aware of a new device, it uses the request-response protocol described above to obtain information about all parameters, external objects, and menus. It then creates the UI page based on the information provided. Each device may also send asynchronous messages (traps) to notify DashBoard of changes; this allows DashBoard to update the UI as required to reflect changes in the device state and options.

The following notification messages are supported:

Command	Msg type	Description
OGP_REPORT_PARAM	0x10	a numeric OID parameter values has changed
OGP_REPORT_STRING	0x14	a string OID parameter values has changed
OGP_TRAP	0x11	send an asynchronous TRAP. For details, see "Trap Messages" on page 6–3.

Trap Messages

A variety of other events can be reported by means of asynchronous TRAP message types. This includes informing DashBoard that a new card has been inserted in a slot, or that the parameter or menu structures have changed. Traps can also be used to hide, enable or disable, or reformat ***specific*** UI elements dynamically, as required by the device.

***Note**: Some trap types will interfere with the DataSafe restore process and should not be sent while the restore is in progress. For more information, see "DataSafe" on page 8–1.*

The following asynchronous messages are encoded using message type 0x11 (OGP_TRAP). These traps should be sent to address 0x02 (OGP_ADDR_TRAP).

Command	Msg type	Trap ID	Description
OGP_RESTART	0x11	0x01	the device has just booted up
OGP_PARAM_CHANGED	0x11	0x02	the descriptor of an OID has changed
OGP_MENU_CHANGED	0x11	0x03	a menu descriptor has changed (e.g. items have been added or removed)
OGP_EXTOBJ_CHANGED	0x11	0x04	an external object has changed
OGP_REVEAL_ELEMENT	0x11	0x05	request an element become visible in the UI
OGP_HIDE_ELEMENT	0x11	0x06	request an element to be hidden in the UI
OGP_RELOAD_UI_ELEMENT	0x11	0x07	causes an element to rebuilt in the UI
OGP_STRING_CHANGED ***Note**: String OID messages are not supported on CANbus connections.*	0x11	0x08	a string OID descriptor has changed

External Objects

In order to manipulate data which cannot be represented with DashBoard's internal data types, an external object may be used. The following messages are supported to handle the transport of External Objects:

Command	Msg type	Description
OGP_GET_EXTERNAL_OBJECT	0x59	Requests an External Object

OGP Parameter Messages

Parameter Query Messages

The following messages are required to exchange parameter information.

OGP_GET_NUMPARAMS Request

Message type: 0x45

Message length: 1

Response required: OGP_GET_NUMPARAMS Response

Description: This requests the number of parameters supported by the device.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
string-support-flag	1	1	unint8	a value of 1 indicates support for String-based OIDs (added in DashBoard 6.1)

OGP_GET_NUMPARAMS Response

Message type: 0xC5

Message length: 3

Response required: none

Description: This is the required response to an OGP_GET_NUMPARAMS request. It provides the number of parameters supported by the device.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK — normal return
numPar	1	2	uint16	number of parameters for this device
xmlDescriptionLen	3	1	uint8	length of xmlDescriptionURL (including null-terminator) to follow
xmlDescriptionURL	4	*	string	URL of a device definition XML file (if a URL is provided, the remainder of the parameter descriptor and menu queries are skipped by DashBoard)
numStringOIDs	4 + xmlDescriptionLen	4	int32	number of parameters with String-based OIDs for this device

OGP_GET_PARAM_OIDS Request

Message type: 0x46

Message length: 4

Response required: OGP_GET_PARAM_OIDS Response

Description: This requests the object IDs for a set of parameters. Because the response can contain a maximum of 128 OIDs, a separate request will be sent for each set of 128 parameters.

Field	Offset	Length	Format	Description
spare	0	1	unint8	placeholder for return code
first	1	2	uint16	index of first OID to return
count	3	1	uint8	number of OIDs requested (0 means send all or 128)

***Note**: Older versions of DashBoard (2.x) may send 2 extra bytes at the end of this message. These should be ignored.*

OGP_GET_PARAM_OIDS Response

Message type: 0xC6

Message length: 4 + 2 * number of OIDs

Response required: none

Description: This is the required response to an OGP_GET_PARAM_OIDS request. It provides the object IDs for up to 128 parameters, starting with the first index requested.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK — normal return • OGP_PARAM_NOTFOUND — requested starting index is greater than the number of parameters
first	1	2	uint16	index of first OID returned

Field	Offset	Length	Format	Description
count	3	1	uint8	number of OIDs returned (max 128)
oids	4	2 * count	uint16[]	array of object IDs

OGP_GET_STRING_OIDS Request

Message type: 0x66

Message length: 7

Response required: OGP_GET_PARAM_OIDS Response

Description: This requests the string object IDs for a set of parameters. Because the response can contain a maximum of 128 OIDs, a separate request will be sent for each set of 128 parameters.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
first	1	4	int32	index of first OID to return
count	5	2	uint16	number of oids requested (0 means send all or 128)

OGP_GET_STRING_OIDS Response

Message type: 0xE6

Message length: 7 + OID list length

Response required: none

Description: This is the required response to an OGP_GET_STRING_OIDS request. It provides the object IDs for up to 128 parameters, starting with the first index requested.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK — normal return • OGP_PARAM_NOTFOUND — requested starting index is greater than the number of parameters
first	1	4	int32	index of first OID returned
count	5	2	uint16	number of OIDs returned in this message
oids	7	*	string	array of strings (each preceded by a length that includes the null-terminator) Each OID can only contain: a-z, A-Z, 0-9, '.', and '_'. Any OID formatted 0x… will be treated as a hex integer. Any OID containing only numbers will be treated as a decimal integer.

Parameter Descriptor Messages

The descriptor for a parameter is provided as a response to the OGP_GET_DESCRIPTOR or OGP_GET_STRING_DESCRIPTOR message.

OGP_GET_DESCRIPTOR Request

Message type: 0x47

Message length: 3

Response required: OGP_GET_DESCRIPTOR Response

Description: This requests the descriptor for the parameter with the specified object ID

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
oid	1	1	uint16	object ID for which the descriptor is requested

OGP_GET_DESCRIPTOR Response

Message type: 0xC7

Message length: variable

Response required: none

Description: This is the required response to an OGP_GET_DESCRIPTOR request. It provides the descriptor for the specified parameter.

Field	Offset	Length	Format	Description
rc		1	uint8	return code: • OGP_OK — normal return • OGP_PARAM_NOTFOUND — parameter not found
oid	1	2	uint16	object ID for which descriptor is provided
len	3	1	uint8	length of the descriptor (max 255)
version	4	1	uint8	version of the parameter descriptor set version = 2 to use widget hints
type	5	1	uint8	data type
size	6	1	uint8	nominal size of the data field (see notes following this table)
access	7	1	uint8	read/write access indicator
precision	8	1	uint8	precision to be displayed for printed numbers
widget	9	1	uint8	widget type to be used to display this parameter this field is ignored if version < 2
namlen	10	1	uint8	name length in bytes including null terminator
name	11	namlen	string	parameter name (null-terminated string)
ctype	namlen + 11	1	uint8	constraint type for this parameter
clen	namlen + 12	1	uint8	length of constraint data in bytes
cdata	namlen + 13	clen	*	ctype-specific constraint data

Notes:

- for arrays, the size should be the array size in bytes (e.g. 2 * number of elements for INT16)
- for strings, the size is the maximum length of the string (the precision is ignored). This controls how many characters the user may enter into the field, and does not include the null-termination byte. The card should expect up to (size + 1) bytes to be received.

- For string arrays, the precision is the maximum length of a single element in the array. This controls how many characters the user may enter into the field, and does not include the null-termination byte. The card should expect up to (precision + 1) bytes to be received.
- the SNMP agent will ignore any information provided in the 'widget' field for this message
- for constraint data encoding, see "Constraint Definitions" on page 6–9.

OGP_GET_STRING_DESCRIPTOR Request

Message type: 0x67

Message length: 2 + oidlen

Response required: OGP_GET_STRING DESCRIPTOR Response

Description: his requests the descriptor for the parameter with the specified object ID.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
oidlen	1	1	uint8	length of OID string (including null terminator)
oid	2	oidlen	string	object ID for which the descriptor is requested including null terminator

OGP_GET_STRING_DESCRIPTOR Response

Message type: 0xE7

Message length: variable

Response required: none

Description: This is the required response to an OGP_GET_STRING_DESCRIPTOR request. It provides the descriptor for the specified parameter.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK — normal return • OGP_PARAM_NOTFOUND — parameter not found
oidlen	1	1	uint8	length of OID string (including null terminator)
oid	2	oidlen	string	object ID for which descriptor is provided
version	oidlen + 2	1	uint8	version of the parameter descriptor set version = 2 to use widget hints
type	oidlen + 3	1	uint8	data type (int, float, string)
size	oidlen + 4	1	uint8	nominal size of the data field (see notes following this table)
access	oidlen + 5	1	uint8	read/write access indicator
precision	oidlen + 6	1	uint8	precision to be displayed for printed numbers
widget	oidlen + 7	1	uint8	widget type to be used to display this parameter this field is ignored if version < 2
namlen	oidlen + 8	1	uint8	name length in bytes including null terminator
name	oidlen + 9	namlen	string	name of the parameter (null-terminated string)
ctype	oidlen + namlen + 9	1	uint8	constraint type for this parameter

Field	Offset	Length	Format	Description
clen	oidlen + namlen + 10	2	uint16	length of constraint data in bytes
cdata	oidlen + namlen + 12	clen	*	ctype-specific constraint data

Notes:

- for arrays, the size should be the array size in bytes (e.g. 2 * number of elements for INT16)
- for strings, the size is the maximum length of the string (the precision is ignored)
- for string arrays, the precision is the maximum length of an element in the array
- the SNMP agent will ignore any information provided in the 'widget' field for this message

Constraint Definitions

NULL_CONSTRAINT

Constraint type: 0x00

Constraint length: 0

Data type: any type

Description: A NULL_CONSTRAINT is used to specify that a parameter is unconstrained; that is, it can take any value legal for its data type. This constraint is encoded as shown in the following table:

Field	Offset	Length	Format	Description
ctype	0	1	uint8	constraint type = 0
clen	1	1	uint8	constraint length = 0

RANGE_CONSTRAINT

Constraint type: 0x01

Constraint length: 2 * data_size (standard), or 4 * data_size (optional)

Data type: INT16, INT32 or FLOAT32

Description: A RANGE_CONSTRAINT specifies the permitted range for a numeric value as a minimum and maximum value, where minValue <= param <= maxValue. This constraint is encoded as shown in the following table. Note that minValue and maxValue must be the same data type as the parameter.

Field	Offset	Length	Format	Description
ctype	0	1	uint8	constraint type = 1
clen	1	1	uint8	constraint length = 2*n
minValue	2	n	type	minimum permitted value for this parameter
maxValue	2+n	n	type	maximum permitted value for this parameter

Where `type` is the parameter data type (int16, int32 or float) and `n` is the length of the data type (2, 4 or 4 respectively.)

A ***RANGE_CONSTRAINT*** can optionally specify a display range for the parameter.

Field	Offset	Length	Format	Description
ctype	0	1	uint8	constraint type = 1
clen	1	1	uint8	constraint length = 4*n

Field	Offset	Length	Format	Description
minValue	2	2	type	minimum permitted value for this parameter
maxValue	2+n	2	type	maximum permitted value for this parameter
minDisp	2+2*n	2	type	minimum display value
maxDisp	2+3*n	2	type	maximum display value

Where `type` is the parameter data type (int16, int32 or float) and `n` is the length of the data type (2, 4 or 4 respectively.)

RANGE_STEP_CONSTRAINT

Constraint type: 0x05

Constraint length: 3 * data_size or 5 * data_size

Data type: INT16, INT32 or FLOAT32

Description: A RANGE_STEP_CONSTRAINT specifies the permitted range for a numeric value as a minimum and maximum value, where minValue <= param <= maxValue and param = minValue + n * step. This constraint is encoded as shown in the following table. Note that minValue, maxValue, and step size must be the same data type as the parameter.

Field	Offset	Length	Format	Description
ctype	0	1	uint8	constraint type = 5
clen	1	1	uint8	constraint length = 3*n
minValue	2	2	type	minimum permitted value for this parameter
maxValue	2+n	2	type	maximum permitted value for this parameter
step	2+2*n	2	type	step size for this parameter

Where `type` is the parameter data type (int16, int32 or float) and `n` is the length of the data type (2, 4 or 4 respectively.

A ***RANGE_STEP_CONSTRAINT*** can optionally specify a display range for the parameter.

Field	Offset	Length	Format	Description
ctype	0	1	uint8	constraint type = 5
clen	1	1	uint8	constraint length = 5*n
minValue	2	2	type	minimum permitted value for this parameter
maxValue	2+n	2	type	maximum permitted value for this parameter
minDisp	2+2*n	2	type	minimum display value
maxDisp	2+3*n	2	type	maximum display value
step	2+4*n	2	type	step size for this parameter

Where `type` is the parameter data type (int16, int32 or float) and `n` is the length of the data type (2, 4 or 4 respectively.

CHOICE_CONSTRAINT

Constraint type: 0x02

Constraint length: variable

Data type: INT16 or INT32

Description: A CHOICE_CONSTRAINT is used to specify a list of named choices (enumeration) that the parameter is allowed to represent. Each choice consists of an unique integer value and an associated string. DashBoard displays the string associated with the parameter value. This constraint is encoded as shown in the following table:

Field	Offset	Length	Format	Description
ctype	0	1	uint8	constraint type = 0x02
clen	1	1	uint8	constraint length (number of bytes that follow)
nchoice	2	1	uint8	number of choices in this constraint
For each choice:				
value	x+0	2	type	numeric value for this choice
namlen	x+n	1	uint8	length of the name in bytes including terminating 0
name	x+n+1	namlen	char[]	associated name as a null-terminated UTF-8 string

Where *type* is the parameter data type (int16 or int32), n is the length of the data type (2 or 4 respectively) and x is the offset of each choice.

EXTENDED_CHOICE

Constraint type: 0x03

Constraint length: variable

Data type: INT16 or INT32

Description: An EXTENDED_CHOICE is identical to a CHOICE_CONSTRAINT, except the number of choices is encoded as a 2-byte value. This allows an EXTENDED_CHOICE to represent extremely long lists of choices. This constraint is encoded as shown in the following table.

Field	Offset	Length	Format	Description
ctype	0	1	uint8	constraint type = 0x03
clen	1	1	uint8	constraint length (number of bytes that follow)
nchoice	2	2	uint16	number of choices in this constraint
For each choice:				
value	x+0	2	type	value for this name-value pair
namlen	x+n	1	uint8	length of the name including terminating NULL
name	x+n+1	namlen	char[]	null-terminated name of this name-value pair

Where *type* is the parameter data type (int16 or int32), n is the length of the data type (2 or 4 respectively) and x is the offset of each choice.

***Note**: When the constraint is encoded in the parameter descriptor, keep in mind that the total message length may not exceed 260 bytes for (CANbus connections) or 8192 bytes (for TCP/IP connections). For long list of choices, an external object must be used to hold the constraint, and the parameter can then refer to it using an EXTERNAL_CONTRAINT.*

STRING_CHOICE

Constraint type: 0x04

Constraint length: variable

Data type: STRING

Description: A STRING_CHOICE is used to provide a list of predefined choices for a STRING. DashBoard treats these choices as suggestions, not an exclusive list (i.e. it will accept other values). The main purpose of this is to support the COMBO_ENTRY widget hint in which a user can either type in a string, or select from a list of values. This constraint is encoded as shown in the following table.

Field	Offset	Length	Format	Description
ctype	0	1	uint8	constraint type = 0x04
clen	1	1	uint8	constraint length (number of bytes that follow)
nchoice	2	2	uint16	number of choices in this constraint
For each choice:				
namlen	x+0	1	uint8	length of the choice in bytes including terminating 0
choice	x+1	namlen	char[]	choice encoded as a null-terminated UTF-8 string

***Note**: When the constraint is encoded in the parameter descriptor, keep in mind that the total message length may not exceed the maximum message length. For long list of choices, an external object must be used to hold the constraint, and the parameter can then refer to it using an EXTERNAL_CONTRAINT*

ALARM_TABLE

Constraint type: 0x0A

Constraint length: variable

Data type: INT16 or INT32

Description: An ALARM_TABLE is typically used for status or error reporting. With this constraint, each bit of the parameter value is interpreted as a separate error flag; making it possible to represent several concurrent error conditions with a single parameter. Each error bit has an associated name and a severity indicator:

- SEVERITY_OK = 0 (displays a GREEN icon, everything OK)
- SEVERITY_WARN = 1 (displays a YELLOW icon)
- SEVERITY_ERROR = 2 (displays a RED icon)

The most severe alarm set determines the color and text displayed by the widget when it appears on a menu page. Additional alarms can be seen in a tool tip when the user hovers the mouse over an alarm display in a menu.

If the value of the parameter is 0 (no bit set), DashBoard will display a GREEN icon with text “OK”. When the alarm bit value is set to 1, DashBoard will display an icon based upon the **severity** field for the alarm bit.

INT16 parameters contain up to 16 alarms. INT32 parameters contain up to 32 alarms.

This constraint is encoded as shown in the following table:

Field	Offset	Length	Format	Description
ctype	0	1	uint8	constraint type = 0x0A
clen	1	1	uint8	constraint length (number of bytes that follow)
nchoice	2	1	uint8	number of alarms in this constraint
For each alarm:				
alarm	x+0	1	uint8	the bit number for this alarm (least significant bit = 0)
severity	x+1	1	uint8	alarm severity (0 = OK, 1 = WARN, 2 = ERROR)

Field	Offset	Length	Format	Description
namlen	x+2	1	uint8	length of the name in bytes including terminating \0
name	x+3	namlen	char[]	associated name as a null-terminated UTF-8 string

EXTERNAL_CONSTRAINT

Constraint type: 0x0B

Constraint length: 2

Data type: any type

Description: An EXTERNAL_CONSTRAINT is used to indicate that the constraint for this parameter is provided in an external object, rather than embedded within the parameter descriptor.

This constraint simply provides a reference to the external object, encoded as shown in the following table:

Field	Offset	Length	Format	Description
ctype	0		uint8	constraint type = 0x0B
clen	1	1	uint8	constraint length = 2
oid	2	1	uint16	object ID of external object containing the constraint

***Note**: The **clen** field specifies the length of the **EXTERNAL_CONSTRAINT** only, not the length of the constraint object it points to.*

Parameter Value Messages

OGP_GET_PARAM_NAME Request

Message type: 0x48

Message length: 3

Response required: OGP_GET_PARAM_NAME Response

Description: This requests the name of the parameter with the specified object ID. This is not currently used by DashBoard.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
oid	1	2	uint16	object ID for which the name is requested

OGP_GET_PARAM_NAME Response

Message type: 0xC8

Message length: 4 + name length

Response required: none

Description: This is the required response to an OGP_GET_PARAM_NAME request. It provides the name of the specified parameter.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK – normal return • OGP_PARAM_NOTFOUND – parameter not found
oid	1	2	uint16	object ID of the requested object
namlen	3	1	uint8	name length in bytes including null terminator
name	4	namlen	char[]	parameter name as a null-terminated UTF-8 string

OGP_GET_PARAM Request

Message type: 0x49

Message length: 3

Response required: OGP_GET_PARAM Response

Description: This requests the value of the parameter with the specified object ID.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
oid	1	2	uint16	object ID for which the value is requested

OGP_GET_PARAM Response

Message type: 0xC9

Message length: 4 + data_size

Response required: none

Description: This is the required response to an OGP_GET_PARAM request. It provides the current value of the requested parameter.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK – normal return • OGP_PARAM_NOTFOUND – parameter not found
oid	1	2	uint16	object ID of the requested object
dlen	3	1	uint8	length of data value field, in bytes, ***Note**: Length is returned only if return code is OGP_OK.*
valus	4	dlen	*	binary encoded value of this parameter ***Note**: Value is returned only if return code is OGP_OK.*

OGP_SET_PARAM Request

Message type: 0x4A

Message length: 4 + data_size

Response required: OGP_SET_PARAM Response

Description: This sets the value of the parameter with the specified object ID. This request may be refused by the card for any reason (e.g. parameter is read only, value is out of range, etc.).

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
oid	1	2	uint16	object ID of the parameter to change
dlen	3	1	uint8	length of data value field, in bytes,
value	4	dlen	*	binary encoded data value to set

OGP_SET_PARAM Response

Message type: 0xCA

Message length: 4 + data_size

Response required: none

Description: This is the required response to an OGP_SET_PARAM request.

It provides the current value of the requested parameter, plus a return code indicating the result of the request. If the OGP_SET_PARAM request is refused, the return code should provide the reason. Please note that it in some cases, the value returned may differ from the requested value, even if the set operation is successful (e.g. if incrementing a value causes it to roll over).

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK – normal return • OGP_PARAM_NOTFOUND – parameter not found • OGP_PARAM_BADVAL – value out of range • OGP_PARAM_READONLY – parameter is read only • OGP_PARAM_LOCKED – edit permission is disabled
oid	1	2	uint16	object ID of the requested object
dlen	3	1	uint8	length of data value field, in bytes, ***Note**: Length is valid only if the parameter exists.*
value	4	dlen	*	binary encoded value of this parameter ***Note**: Value is valid only if the parameter exists.*

OGP_GET_ARRAY_ELEMENT Request

Message type: 0x4C

Message length: 5

Response required: OGP_GET_ARRAY_ELEMENT Response

Description: This requests the value of a single element of an array parameter.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
oid	1	2	uint16	object ID for which the value is requested
index	3	2	uint16	array index of the requested element

OGP_GET_ARRAY_ELEMENT Response

Message type: 0xCC

Message length: 6 + data_size

Response required: none

Description: his is the required response to an OGP_GET_ARRAY_ELEMENT request. It provides the current value of the specified element of the requested parameter.

Note that this is an ***optional*** message type. If the device returns OGP_UNSUPPORTED, then DashBoard will request data for the entire array using ***OGP_GET_PARAM***.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK – normal return • OGP_UNSUPPORTED – message not supported • OGP_PARAM_NOTFOUND – parameter not found • OGP_PARAM_BADINDEX – index out of bounds
oid	1	2	uint16	object ID of the requested object
index	3	2	uint16	array index of the requested element
dlen	5	1	uint8	length of data value field, in bytes, ***Note**: Length is valid only if the return code is OGP_OK.*
value	6	dlen	*	binary encoded value of this array element ***Note**: Value is valid only if the return code is OGP_OK.*

OGP_SET_ARRAY_ELEMENT Request

Message type: 0x4D

Message length: 6 + data_size

Response required: 6 + data_size

Description: This sets the value of a single element of an array parameter. This request may be refused by the card for any reason (e.g. parameter is read only, value is out of range, etc.).

Note that this is an ***optional*** message type. If the device returns OGP_UNSUPPORTED, then DashBoard will set the value for the entire array using ***OGP_SET_PARAM***. If the device supports this message type, it ***must*** also support the OGP_GET_ARRAY_ELEMENT message type.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
oid	1	2	uint16	object ID of the parameter to change
index	3	2	uint16	array index of the element to change
dlen	5	1	uint8	length of data value field, in bytes,
value	6	dlen	*	binary encoded data value to set

OGP_SET_ARRAY_ELEMENT Response

Message type: 0xCD

Message length: 6 + data_size

Response required: none

Description: This is the required response to an OGP_SET_ARRAY_ELEMENT request. It provides the current value of the requested element of the requested parameter, plus a return code indicating the result of the request. If the OGP_SET_ARRAY_ELEMENT request is refused, the return code should provide the reason. Note that it in some cases, the value returned may be different from the value in the set request, even if the set operation is successful (e.g. if incrementing a value causes it to roll over).

Note that this is an ***optional*** message type. If the device returns OGP_UNSUPPORTED, then DashBoard sets the value for the entire array using ***OGP_SET_PARAM***.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK – normal return • OGP_UNSUPPORTED – message not supported • OGP_PARAM_NOTFOUND – parameter not found • OGP_PARAM_BADINDEX – index out of bounds • OGP_PARAM_BADVAL – value out of range • OGP_PARAM_READONLY – parameter is read only • OGP_PARAM_LOCKED – edit permission is disabled
oid	1	2	uint16	object ID of the requested object
index	3	2	uint16	array index of the requested element
dlen	5	1	uint8	length of data value field, in bytes, ***Note**: Length is valid only if the parameter exists.*
value	6	dlen	*	binary encoded value of this array element ***Note**: Value is valid only if the parameter exists.*

OGP_GET_STRING Request

Message type: 0x69

Message length: variable

Response required: OGP_GET_STRING Response

Description: This requests the value of the parameter with the specified object ID. Not supported on CANbus connections.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
oidlen	1	1	uint8	length of OID string data
oid	2	oidlen	string	object ID of the requested object

OGP_GET_STRING Response

Message type: 0xE9

Message length: 4 + data_size

Response required: none

Description: This is the required response to an OGP_GET_STRING request. It provides the current value of the requested parameter. Not supported on CANbus connections.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK – normal return • OGP_PARAM_NOTFOUND – parameter not found
oidlen	1	1	uint8	length of OID string data
oid	2	oidlen	string	object ID of the requested object
dlen	3 + oidlen	1	uint8	length of data value field (in bytes), or 0 for 'long params'. ***Note**: Length is valid only if the return code is OGP_OK.*
value	4 + oidlen	variable		binary encoded value of this parameter ***Note**: Value is valid only if the return code is OGP_OK.*

OGP_SET_STRING Request

Message type: 0x6A

Message length: variable

Response required: GP_SET_PARAM Response

Description: his sets the value of the parameter with the specified object ID. This request may be refused by the card for any reason (e.g. parameter is read only, value is out of range, etc.). Not supported on CANbus connections.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
oidlen	1	1	uint8	length of OID string data
oid	2	oidlen	string	object ID of the parameter to change
dlen	3 + oidlen	1	uint8	length of data value field (in bytes), or 0 for 'long params'.
value	4 + oidlen	variable	*	binary encoded data value to set

OGP_SET_STRING Response

Message type: 0xEA

Message length: variable

Response required: none

Description: This is the required response to an OGP_SET_STRING request. It provides the current value of the requested parameter, plus a return code indicating the result of the request. If the OGP_SET_STRING request is refused, the return code should provide the reason. Note that it in some cases, the value returned may differ from the

requested value, even if the set operation is successful (e.g. if incrementing a value causes it to roll over). Not supported on CANbus connections.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK – normal return • OGP_PARAM_NOTFOUND – parameter not found • OGP_PARAM_BADVAL – value out of range • OGP_PARAM_READONLY – parameter is read only • OGP_PARAM_LOCKED – edit permission is disabled
oidlen	1	1	uint8	length of OID string data
oid	2	oidlen	string	object ID of the parameter to change
dlen	3 + oidlen	1	uint8	length of data value field (in bytes), or 0 for 'long params'. ***Note**: Length is valid only if the parameter exists.*
value	4 + oidlen	variable	*	binary encoded value of this parameter ***Note**: Value is valid only if the parameter exists.*

String Array Parameters

Each array element provides a 1-byte length followed by a null-terminated UTF-8 string value. The length count includes the null-termination byte.

Element length: 2 + string_size

Field	Offset	Length	Format	Description
For each element:				
element	x	1	uint8	length of the element value including terminating NULL
value	x + 1	namlen	char[]	null-terminated value of this element

Notes:

- Neither the length byte nor the terminating NULL is included when determining how many bytes are allowed in an individual array element. The maximum number of **character data** bytes allowed in an array element is defined by the **precision** field in the parameter descriptor and the maximum number of characters for all elements combined is defined by the **data_size** field. If the precision is 0, the **data_size** is considered the maximum length for all elements combined and the maximum length for an individual element.
- For example: A String Array parameter with a data_size of 30 and a precision of 5 would have 6 elements with a maximum number of 5 bytes of character data in each element. If the precision were 0, each element could contain up to 30 bytes of character data as long as all elements combined do not exceed 30 bytes of character data.

OGP_GET_STRING_ARRAY_ELEMENT Request

Message type: 0x6C

Message length: variable

Response required: OGP_GET_STRING_ARRAY_ELEMENT Response

Description: This requests the value of a single element of an array parameter. Not supported on CANbus connections.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for the return code
oidlen	1	1	uint8	length of OID string data
oid	2	oidlen	string	object ID for which the value is requested
index	3 + oidlen	2	uint16	array index of the requested element

OGP_GET_STRING_ARRAY_ELEMENT Response

Message type: 0xEC

Message length: variable

Response required: none

Description: This is the required response to an OGP_GET_ARRAY_ELEMENT request. It provides the current value of the specified element of the requested parameter. Not supported on CANbus connections.

Note that this is an ***optional*** message type. If the device returns OGP_UNSUPPORTED, then DashBoard will request data for the entire array using ***OGP_GET_PARAM***.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK – normal return • OGP_UNSUPPORTED – message not supported • OGP_PARAM_NOTFOUND – parameter not found • OGP_PARAM_BADINDEX – index out of bounds
oidlen	1	1	uint8	length of OID string data
oid	2	oidlen	string	object ID for which the value is requested
index	3 + oidlen	2	uint16	array index of the requested element
dlen	5 + oidlen	1	uint8	length of data value field (in bytes), or 0 for ‘long params’. *Note: Length is valid only if the return code is OGP_OK.*
value	6 + oidlen	variable	*	binary encoded value of this array element ***Note**: Value is valid only if the return code is OGP_OK.*

OGP_SET_STRING_ARRAY_ELEMENT Request

Message type: 0x6D

Message length: variable

Response required: OGP_SET_STRING_ARRAY_ELEMENT Response

Description: This sets the value of a single element of an array parameter. This request may be refused by the card for any reason (e.g. parameter is read only, value is out of range, etc.). Not supported on CANbus connections.

Note that this is an ***optional*** message type. If the device returns OGP_UNSUPPORTED, then DashBoard will set the value for the entire array using ***OGP_SET_STRING***. If the device supports this message type, it ***must*** also support the OGP_GET_ARRAY_ELEMENT message type.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
oidlen	1	1	uint8	length of OID string data
oid	2	oidlen	string	object ID of the parameter to change
index	3 + oidlen	2	uint16	array index of the element to change
dlen	5 + oidlen	1	uint8	length of data value field (in bytes), or 0 for 'long params'.
value	6 + oidlen	variable	*	binary encoded data value to set

OGP_SET_STRING_ARRAY_ELEMENT Response

Message type: 0xED

Message length: variable

Response required: none

Description: This is the required response to an OGP_SET_STRING_ARRAY_ELEMENT request. It provides the current value of the requested element of the requested parameter, plus a return code indicating the result of the request. If the OGP_SET_STRING_ARRAY_ELEMENT request is refused, the return code should provide the reason. Note that it in some cases, the value returned may be different from the value in the set request, even if the set operation is successful (e.g. if incrementing a value causes it to roll over). Not supported on CANbus connections.

Note that this is an ***optional*** message type. If the device returns OGP_UNSUPPORTED, then DashBoard will set the value for the entire array using ***OGP_SET_PARAM***.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK – normal return • OGP_UNSUPPORTED – message not supported • OGP_PARAM_NOTFOUND – parameter not found • OGP_PARAM_BADINDEX – index out of bounds • OGP_PARAM_BADVAL – value out of range • OGP_PARAM_READONLY – parameter is read only • OGP_PARAM_LOCKED – edit permission is disabled
oidlen	1	1	uint8	length of OID string data
oid	2	oidlen	string	object ID of the parameter to change
index	3 + oidlen	2	uint16	array index of the element to change
dlen	5 + oidlen	1	uint8	length of data value field (in bytes), or 0 for 'long params'. ***Note**: Length is valid only if the parameter exists.*
value	6 + oidlen	dlen	*	binary encoded value of this array element ***Note**: Value is valid only if the parameter exists.*

Connection Handshake

CONNECTION_OGP_SET_PARAM Request

Message type: 0x4A

Message length: 6 + data_size

Response required: CONNECT_OGP_SET_PARAM Response

Description: This message is sent by DashBoard to request a connection to the device. The device must respond with CONNECTION_OGP_SET_PARAM response, even if the connection is being rejected.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
oid	1	2	uint16	0xFF03
dlen	3	1	uint8	length in bytes of the remainder of the message
force	4	2	uint16	• 0x0000 — connect if a connection is available • 0x0001 — force a connection even if all connections are in use (disconnect from another DashBoard if necessary)
authen (optional field)	6	2	uint16	• 0x0000 — Connection without any authentication • 0x0001 — Trust the connection (DashBoard is able to contact its user rights server) • 0x0002 — Authenticate the connection with the following password (used when DashBoard cannot connect to a user rights server)
passlen (optional field)	8	1	uint8	The length of the password (including the null-terminator). Only sent when the authen field is set to 0x0002.
password (optional field)	9	*	string	Null-terminated password to validate. Only sent when the authen field is set to 0x0002.

CONNECTION_OGP_SET_PARAM Response

Message type: 0xCA

Message length: 4 + data_size

Response required: none

Description: This response message must only be sent to the connection that issued the CONNECTION_OGP_SET_PARAM Request. It can be used to tell DashBoard to close its connection and provide a reason why the connection was terminated. When DashBoard receives a value of 0x0000 in the allow field, both DashBoard and the device should close their connections.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK – normal return
oid	1	2	uint16	0xFF03
dlen	3	1	uint8	length in bytes of the remainder of the message
allow	4	2	uint16	• 0x0000 — Connection is refused • 0x0001 (or greater) — Connection is allowed

Field	Offset	Length	Format	Description
urmstate (optional field)	6	2	uint16	• 0x0000 — URM not available • 0x0001 — URM Disabled • 0x0002 — URM Enabled
reason (optional field)	8	2	uint16	• 0x0000 — No connection available • 0x0001 — Trusted connection required but not given • 0x0002 — Password incorrect ***Note**: The value of this field is ignored if allow is non-zero*

Parameter Value Change Messages

All parameter value change messages are asynchronous notifications.

These messages should be sent to address 0x02 (OGP_ADDR_TRAP).

OGP_REPORT_PARAM Message

Message type: 0x10

Message length: 4 + data_size

Response required: none

Description: This message should be sent when the value of a parameter has changed (except when changed by an OGP_SET_PARAM request). The message format is identical to that of the OGP_GET_PARAM response. It provides the object ID and value of the parameter that has changed.

To avoid overloading the CAN Bus in the openGear frame, an openGear card should not report an individual parameter more often than once per second. During normal operation, each card must not send more than 5 ***OGP_REPORT_PARAM*** messages per second.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code — must be 0
oid	1	2	uint16	object ID of the requested object
dlen	3	1	uint8	length of data value field, in bytes
value	4	dlen	*	binary encoded value of this parameter

OGP_REPORT_STRING Message

Message type: 0x14

Message length: variable

Response required: none

Description: This message should be sent when the value of a parameter has changed (except when changed by an OGP_SET_STRING request). The message format is identical to that of the OGP_GET_STRING response. It provides the object ID and value of the parameter that has changed. Not supported on CANbus connections.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code — must be 0
oidlen	1	1	uint8	length of OID string data
oid	2	oidlen	string	object ID of the requested object

Field	Offset	Length	Format	Description
dlen	3 + oidlen	1	uint8	length of data value field (in bytes), or 0 for ‘long params’
value	4 + oidlen	dlen	*	binary encoded value of this parameter

OGP_RESTART Trap

Message type: 0x11

Trap ID: 0x01 (OGP_RESTART)

Message length: 4

Response required: none

Description: This notification must be sent after the device has booted up. It may also be sent whenever the parameter or menu descriptors for the device have changed significantly. DashBoard will request all of the descriptive information from the device and completely update the display.

Field	Offset	Length	Format	Description
trapID	0	1	uint8	notification type = 1 (OGP_RESTART)
spare	1	3	n/a	reserved, should be 0

This trap is sufficient to allow DashBoard to update appropriately for any change on the device; however, for changes to individual parameters and menus, it is more efficient to use the specific change notifications described below.

***Note**: Sending this message causes DataSafe and SNMP agents to re-query the card. Designers must ensure that this trap is NEVER transmitted during the DataSafe restore process; otherwise an endless loop will occur, with the CAN bus saturated with traffic.*

OGP_PARAM_CHANGED Trap

Message type: 0x11

Trap ID: 0x02 (OGP_PARAM_CHANGED)

Message length: 4

Response required: none

Description: This notification should be sent when the descriptor for a parameter has changed. The nature of the change is indicated by the change code. If it is a simple permission change, DashBoard will simply enable or disable any controls for the parameter. If it is a more substantial change, DashBoard will request the descriptor for the specified parameter and update the display as required. This is typically used to change the constraint or display hint for a parameter dynamically, without having to refresh the entire card. Notifications for non-existent OIDs will be ignored.

***Note**: The data type of a parameter should **not** be changed at run-time.*

Field	Offset	Length	Format	Description
trapID	0	1	uint8	notification type = 2 (OGP_PARAM_CHANGED)
oid	1	2	uint16	object ID for the parameter that has changed

Field	Offset	Length	Format	Description
change	3	1	uint8	Change code: • 0 – parameter is now READONLY • 1 – parameter is now READWRITE • 2 – parameter descriptor has changed
descchange	4	2	uint16	Descriptor part change code (only used when the change field is set to "2"): • 0 – Request full reload of parameter descriptor • 1 – Request reload of choice constraint ***Note**: This field is optional and if it is not included, the entire parameter is reloaded.*

OGP_STRING_CHANGED Trap

Message type: 0x11

Trap ID: 0x08 (OGP_STRING_CHANGED)

Message length: variable

Response required: none

Description: This is identical to OGP_PARM_CHANGED trap, but for parameters with string rather than numeric OID. This notification should be sent when the descriptor for a parameter has changed. The nature of the change is indicated by the change code. If it is a simple permission change, DashBoard will simply enable or disable any controls for the parameter. If it is a more substantial change, DashBoard will request the descriptor for the specified parameter and update the display as required. This is typically used to change the constraint or display hint for a parameter dynamically, without having to refresh the entire device. Notifications for non-existent OIDs will be ignored. Not supported on CANbus connections.

***Note**: The data type of a parameter should **not** be changed at run-time.*

Field	Offset	Length	Format	Description
trapID	0	1	uint8	notification type = 2 (OGP_PARAM_CHANGED)
oidlen	1	1	uint8	length of OID String data
oid	2	oidlen	string	object ID for the parameter that has changed
change	3 + oidlen	1	uint8	Change code: • 0 – parameter is now READONLY • 1 – parameter is now READWRITE • 2 – parameter descriptor has changed
descchange	4 + oidlen	2	uint16	Descriptor part change code (only used when the change field is set to "2"): • 0 – Request full reload of parameter descriptor • 1 – Request reload of choice constraint ***Note**: This field is optional and if it is not included, the entire parameter is reloaded.*

OGP_MENU_CHANGED Trap

Message type: 0x11

Trap ID: 0x03 (OGP_MENU_CHANGED)

Message length: 4

Response required: none

Description: This notification should be sent when the descriptor for a menu has changed. The nature of the change is indicated by the change code. If it is a simple menu-state change, DashBoard will simply hide, show, enable or disable the menu as appropriate. If it is a more substantial change, DashBoard will request the information (name, oids and state) for the specified menu and update the display as required. Notifications for non-existent menuIDs will be ignored.

Field	Offset	Length	Format	Description
trapID	0	1	uint8	notification type = 3 (OGP_MENU_CHANGED)
group	1	1	uint8	group ID for the menu that has changed
menu	2	1	uint8	menu ID for the menu that has changed
change	3	1	uint8	Change code: • 0 – menu state is hidden • 1 – menu state is visible and disabled • 2 – menu state is visible and editable (normal state) • 3 – menu descriptor has changed

OGP_EXTOBJ_CHANGED Trap

Message type: 0x11

Trap ID: 0x04 (OGP_EXTOBJ_CHANGED)

Message length: 4

Response required: none

Description: This notification should be sent when an external data object has changed. DashBoard will request the data object and update any controls that depend on that object. Notifications for non-existent object IDs will be ignored.

Field	Offset	Length	Format	Description
trapID	0	1	uint8	notification type = 4 (OGP_EXTOBJ_CHANGED)
oid	1	2	uint16	object ID for the external object
change	3	1	uint8	Change code: • 0 – data content has changed

OGP_REVEAL_ELEMENT Trap

Message type: 0x11

Trap ID: 0x05 (OGP_REVEAL_ELEMENT)

Message length: variable

Response required: none

Description: This notification applies to devices and/or individual menus defined with OGLML documents. This trap causes the named element to become visible in the UI (generally only applies if that element is inside of a tab control). Multiple elements can be listed in the same trap.

Field	Offset	Length	Format	Description
trapID	0	1	uint8	5 (OGP_ REVEAL_ELEMENT)
idlen	1	1	uint8	Length of element ID to follow
id	3	idlen	char[]	UTF-8 Encoded ID of the element to reveal

Field	Offset	Length	Format	Description
...	...	...	...	
idlen	...	1	uint8	Length of element ID to follow
id	...	idlen	char[]	UTF-8 Encoded ID of the element to reveal

OGP_HIDE_ELEMENT Trap

Message type: 0x11

Trap ID: 0x06 (OGP_HIDE_ELEMENT)

Message length: variable

Response required: none

Description: This notification applies to devices and/or individual menus defined with OGLML documents. This trap causes the named element to become invisible in the UI (generally only applies if that element is inside of a tab control). Multiple elements can be listed in the same trap.

Field	Offset	Length	Format	Description
trapID	0	1	uint8	notification type = 6 (OGP_ HIDE_ELEMENT)
idlen	1	1	uint8	Length of element ID to follow
id	3	idlen	char[]	UTF-8 Encoded ID of the element to hide
...	...	...	...	
idlen	...	1	uint8	Length of element ID to follow
id	...	idlen	char[]	UTF-8 Encoded ID of the element to hide

OGP_RELOAD_UI_ELEMENT Trap

Message type: 0x11

Trap ID: 0x07 (OGP_RELOAD_ELEMENT)

Message length: variable

Response required: none

Description: This notification applies to devices and/or individual menus defined with OGLML documents. This trap causes the named element to be rebuilt in the UI. Multiple elements can be listed in the same trap. If certain parts of an OGLML document are dynamically generated or the OGLML document has changed, this trap will allow those changes to be reflected in the UI.

Field	Offset	Length	Format	Description
trapID	0	1	uint8	notification type = 6 (OGP_ HIDE_ELEMENT)
idlen	1	1	uint8	Length of element ID to follow
id	3	idlen	char[]	UTF-8 Encoded ID of the element to reveal
...	...	...	...	notification type = 7 (OGP_ RELOAD_ ELEMENT)
idlen	...	1	uint8	Length of element ID to follow
id	...	idlen	char[]	UTF-8 Encoded ID of the element to reveal

Menu-Definition Messages

OGP_GET_MENUSET_NAME Request

Message type: 0x50

Message length: 2

Response required: GP_GET_MENUSET_NAME Response

Description: This message requests the name of a menu group (i.e. a set of tabbed menus) by group number. The DashBoard generic device page displays two menu groups (0 = status parameters, 1 = user settings). A separate request will be sent for each group.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
group	1	1	uint8	which menu group is being requested

OGP_GET_MENUSET_NAME Response

Message type: 0xD0

Message length: 4 + name length

Response required: none

Description: This is the required response to an OGP_GET_MENUSET_NAME request. It provides the name of the specified menu group. Every device must provide a valid response (OGP_OK) for groups 0 and 1.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code • OGP_OK – normal return • OGP_NOTFOUND – the requested group does not exist
group	1	1	uint8	which menu group is being reported
spare	2	1	uint8	spare – should be zero
namlen	3	1	uint8	length of name including null termination
name	4	namlen	char[]	null-terminated string holding the menu group name

OGP_GET_MENU_COUNT Request

Message type: 0x51

Message length: 2

Response required: OGP_GET_MENU_COUNT Response

Description: This message requests the number of menus (tabs) in a menu group. The DashBoard generic device page displays two menu groups (0 = status parameters, 1 = user settings). A separate request will be sent for each group.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
group	1	1	uint8	which menu group is being requested

OGP_GET_MENU_COUNT Response

Message type: 0xD1

Message length: 3

Response required: none

Description: This is the required response to an OGP_GET_MENU_COUNT request. It provides the number of menus (tabs) in the specified menu group. Every device must provide a valid response (OGP_OK) for groups 0 and 1. If the menu count is zero, the group will not be displayed.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code • OGP_OK – normal return • OGP_NOTFOUND – the requested group does not exist
group	1	1	uint8	which menu group is being reported
count	2	1	uint8	number of menus in this group

OGP_GET_MENU_NAME Request

Message type: 0x52

Message length: 3

Response required: OGP_GET_MENU_NAME Response

Description: This message requests the name to be displayed on a specific menu tab, by group and menu number. A separate request will be send for each menu in groups 0 and 1.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
group	1	1	uint8	which menu group is being requested (requests are sequential)
menu	2	1	uint8	which menu is being requested

OGP_GET_MENU_NAME Response

Message type: 0xD2

Message length: 8 + name length + layoutlen

Response required: none

Description: This is the required response to an OGP_GET_MENU_NAME request. It provides the name to be displayed on the specific menu tab. Every device must provide a valid response (OGP_OK) for each menu in groups 0 and 1.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code • OGP_OK – normal return • OGP_NOTFOUND – the requested menu does not exist
group	1	1	uint8	which menu group is being reported
menu	2	1	uint8	which menu is being reported
namlen	3	1	uint8	length of name including null termination
name	4	namlen	char[]	null-terminated string holding the menu name

Field	Offset	Length	Format	Description
groupuniqueid	4 + namlen	2	uint16	unique identifier for the menu group being requested. The Group Unique ID field allows a menu to be identified regardless of the order in which the groups are returned. ***Note**: This field is optional but strongly recommended for all new designs.*
layoutlen	6 + namlen	1	uint8	The length of the OGLML URL to follow
layouturl	7 + namlen	layoutlen	char[]	An OGLML URL for the layout for the menu. ***Note**: This field is optional (for menus that use OGLML descriptors)*
numStringOIDs	7 + namlen + layoutlen	2	uint16	The number of String OIDs to use (if > 0, only string OIDs will be fetched) ***Note**: This field is optional (for menus that use OGLML descriptors)*

The **groupuniqueid** field was added in DashBoard 4.0. Devices which do not provide this field in their response (as determined by the overall message length) are treated by DashBoard as having an implied groupuniqueid following the formula:

```
groupuniqueid = group << 8 | menu
```

When adding **groupuniqueid** to an existing device, device designers must ensure that the value exactly matches the previously implied value for all pre-existing group/menu entries. For new menu entries, or new device designs, any value may be used, as long as they are unique, and remain fixed for the life of the product.

Menu order should not be changed dynamically unless the **groupuniqueid** field is defined for all menus in a device.

The optional fields beginning with **layouttype** were added in DashBoard v4.1, and are used to reference an OGLML document. For more information, see the ***DashBoard CustomPanel Development Guide (8351DR-007-xx)***.

OGP_GET_MENU_OIDS Request

Message type: 0x53

Message length: 3

Response required: OGP_GET_MENU_OIDS Response

Description: This message requests the object IDs of the parameters to be displayed on the specified menu. A separate request will be send for each menu in groups 0 and 1.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
group	1	1	uint8	which menu group is being requested
menu	2	1	uint8	which menu is being requested

OGP_GET_MENU_OIDS Response

Message type: 0xD3

Message length: 4 + 2 * number of parameters

Response required: none

Description: This is the required response to an OGP_GET_MENU_OIDS request. It provides the object IDs of the parameters to be displayed on the specified menu. Every device must provide a valid response (OGP_OK) for each menu in groups 0 and 1. Non-existent parameters are not displayed (but do not result in an error).

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code • OGP_OK – normal return • OGP_NOTFOUND – the requested menu does not exist
group	1	1	uint8	which menu group is being reported
menu	2	1	uint8	which menu is being reported
count	3	1	uint8	number of objects in this menu (max 128)
oids	4	2*count	uint16[]	array of object IDs to be displayed in this menu

OGP_GET_MENU_STRING_OIDS Request

Message type: 0x73

Message length: 7

Response required: OGP_GET_MENU_STRING_OIDS Response

Description: This message requests the object IDs of the parameters to be displayed on the specified menu. A separate request will be send for each menu in groups 0 and 1. Not supported on CANbus connections.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
group	1	1	uint8	which menu group is being requested
menu	2	1	uint8	which menu is being requested
first	3	2	uint16	index of first oid to return
count	5	2	uint16	number of oids requested (0 means send as many as will fit)

OGP_GET_MENU_STRING_OIDS Response

Message type: 0xF3

Message length: 4 + 2 * number of parameters

Response required: none

Description: This is the required response to an OGP_GET_MENU_STRING_OIDS request. It provides the object IDs of the parameters to be displayed on the specified menu. Every device must provide a valid response (OGP_OK) for each menu in groups 0 and 1. Non-existent parameters are not displayed (but do not result in an error). Not supported on CANbus connections.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code • OGP_OK – normal return • OGP_NOTFOUND – the requested menu does not exist
group	1	1	uint8	which menu group is being reported
menu	2	1	uint8	which menu is being reported
first	3	2	uint16	index of the first OID being returned

Field	Offset	Length	Format	Description
count	5	2	uint16	number of objects in this menu
oids	7	variable	uint16[]	array of String-based object IDs to be displayed in this menu each preceded by a uint8 length count

OGP_GET_MENU_STATE Request

Message type: 0x5A

Message length: 3

Response required: OGP_GET_MENU_STATE Response

Description: This message requests the display state for the specified menu. A separate request will be send for each menu in groups 0 and 1.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
group	1	1	uint8	which menu group is being requested
menu	2	1	uint8	which menu is being requested

OGP_GET_MENU_STATE Response

Message type: 0xDA

Message length: 4

Response required: none

Description: This is the required response to an OGP_GET_MENU_STATE request. It provides the display state for the specified menu. State may be one of the following:

- OGP_MENU_HIDDEN — menu is hidden
- OGP_MENU_READONLY — menu is visible, but disabled
- OGP_MENU_NORMAL — menu is visible and enabled

This is an ***optional*** feature. Devices that support this feature should provide a valid response for each menu in groups 0 and 1. Devices that do not support this feature should return OGP_UNSUPPORTED. The default menu state is visible and editable.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK – normal return • OGP_NOTFOUND – the requested menu does not exist • OGP_UNSUPPORTED – feature is not supported
group	1	1	uint8	which menu group is being reported
menu	2	1	uint8	which menu is being reported
state	3	1	uint8	current display state for this menu: • 0 – menu is hidden • 1 – menu is visible and disabled • 2 – menu is visible and editable (normal state)

Command Interface

The command interface is useful for sending simple UTF-8 text strings to the device. This is particularly useful for implementing debug commands (e.g. to peek and poke registers) that would typically be carried over a serial debug port. Providing this functionality via the openGear protocol allows debug access to all cards via an existing communications channel (i.e. the CAN bus). This eliminates the need for additional serial ports and cables, and allows debug access to cards while the frame is closed..

Command	Msg type	Description
OGP_COMMAND	0x44	send a command to the card
OGP_PRINT	0x00	send a print or debug message to control system

Messages

OGP_COMMAND Request

Message type: 0x44

Message length: variable

Response required: OGP_COMMAND Response

Description: This message is used to send a command string to the device. This allows implementing a device-specific command-line interface that communicates via OGP (rather than by debug port). Interpretation of the string is device-specific.

Field	Offset	Length	Format	Description
command	0	*	char[]	command as a null-terminated UTF-8 string

OGP_COMMAND Response

Message type: 0xC4

Message length: 1

Response required: none

Description: This is the required response to an OGP_COMMAND request. If the device supports the command interface, it should simply acknowledge the message by returning OGP_OK. Otherwise it should return OGP_UNSUPPORTED.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code • OGP_OK – command acknowledged • OGP_UNSUPPORTED – command interface not supported

OGP_PRINT Message

Message type: 0x00

Message length: variable

Response required: none

Description: This message may be used to send an unsolicited text string from a card to the control system. It contains a null-terminated UTF-8 string. Interpretation of the message is card-specific.

Field	Offset	Length	Format	Description
text	0	any	char[]	print text encoded as null-terminated UTF-8 string

openGear Frame Messages

The messages in this section are applicable only to CAN-connected openGear cards in an openGear Frame.

Time Service

If configured appropriately, the network card maintains current time through connection with an NTP (Network Time Protocol) server. The TransCAN program (v1.20 and later) broadcasts time every 10 seconds on the CAN bus as an OGP message with destination address **OGP_ADDR_BCAST**.

Individual openGear cards can use this time as required to synchronize internal operations. If a device requires accurate timing, it should implement a PLL to filter the time messages to reduce jitter associated with message timing.

***Note**: The Time Service is a function of the Frame Network Controller; this feature is not available for DashBoard Connect devices.*

This service has one associated message:

OGP_TIME Broadcast Message

Message type: 0x01

Message length: 8

Response required: none

Description: This message is used to broadcast the current time. The timestamp is written in immediately before the message is sent.

Field	Offset	Length	Format	Description
seconds	0	4	uint32	UTC time in seconds since January 1, 1970
millis	4	2	uint16	sub-seconds (milliseconds)
flags	6	1	uint8	Validity flags: • flags[7:1] — reserved (should be zero) • flags[0] —1: sender is synchronized via NTP — 0: sender is not synchronized
reserved	7	1	uint8	Reserved field. Should be ignored.

Fan Door Control

The frame controllers broadcast the fan door state every 10 seconds as an **OGP_REPORT_PARAM** message with the destination address **OGP_ADDR_BCAST.** Cards must NOT respond to these broadcast messages. Individual openGear cards can use these messages to track the fan door state.

In the DFR-8310 and DFR-8321 frames, control of fan speed is fully automatic. The MFC controller card measures power consumption and adjust the fan speeds accordingly. Under elevated ambient temperatures, the fans run a little faster.

In the OG3-FR high power frame, the legacy fan control algorithm remains present, and provides equivalent amount of cooling. Since the total power can be higher, the range of fan speed has been increased.

Any high power card (defined as 7.5 W per slot or higher) should implement OID 0xFF11 in order to inform the MFC controller of its cooling requirements. This value should be broadcast by the card, roughly once per minute.

The MFC controller will monitor the requests from all slots, and will set the fans according to the highest requested value from any slot.

This service has the following associated messages:.

Command	Msg type	OID	Description
OID_DOOR_STATE	0x10	0x0709 0xFE0B	reports whether fan door is open or closed
OID_FAN_AMBIENT_TEMP	0x10	0xFE0B 0xFE0C	ambient temperature at the inlet to the frame (in degrees Celsius)
OID_SPEED_REPORT	0x10	0xFE0D	reports the current fan speed
OID_FAN_SPEED_REQUEST	0x10	0xFF11	used by high-power cards to alter the amount of cooling

Note: *Fan Door control messages are only available to openGear cards installed in an openGear frame; they are not relevant for stand-alone DashBoard connect devices.*

OID_DOOR_STATE Parameter

Message type: 0x10

Message length: 6

Response required: none

Description: This message is used to broadcast the fan door state. The OID for the fan door state is historically 0x0709. New MFC controllers also transmit the same message on OID 0xFE0B. The data is the fan door state (1= closed, 2=open).

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code – must be 0
oid	1	2	uint16	object OID of the fan door state: • value is 0x0709 for legacy controllers • value is 0xFE0B for modern MFC controllers
dlen	3	1	uint8	Length in bytes of the data value – is 2
value	4	2	uint16	The fan door state: • 1 = closed • 2 = open

OID_FAN_AMBIENT_TEMP Parameter

Message type: 0x10

Message length: 6

Response required: none

Description: This message is used to broadcast ambient temperature at the inlet to the frame, as measured by sensors located in the door. Readings will be zero while the door is opened, and may take a minute to settle after the door has been closed. The realistic range of operation for openGear products is between 0°C to 40°C, however the sensor range is considerably larger.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code – must be 0
oid	1	2	uint16	object OID for ambient temperature 0xFE0C

Field	Offset	Length	Format	Description
dlen	3	1	uint8	Length in bytes of the data value – is 2
value	4	2	uint16	Current ambient temperature in degrees Celsius. Reading will be 0 (zero) when the door is opened.

This message has been added in MFC-83xx-N software v2.75 and MFC-832x-S v2.00.

OID_SPEED_REPORT

Message type: 0x10

Message length: 6 or 8

Response required: none

Description: This message is sent by the MFC to inform cards of the current fan speed. Cards may use this knowledge to alter the amount of cooling they request (see OID 0xFF11). The range of values depends on the frame type. The DFR-8321 series frames have 5 fan speeds, while the OG3-FR has 14 speeds, and future frames may have even more levels. Cards should be prepared for this.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code – must be 0
oid	1	2	uint16	object OID for fan speed 0xFE0D
dlen	3	1	uint8	Length in bytes of the data value – 2 or 4
valule	4	2	uint16	Current fan speed value: • Value of 0 means the door is open (fans off). • Range is 1 to 5 for DFR-8310 and DFR-832x frames. • Range is 1 to 14 for high power OG3-FR frame.
max	6	2	uint16	Maximum fan speed for this MFC controller: • Will be 5 on DFR-8310 and DFR-83xx frames. • Will be 14 on OG3-FR high power frame.

This message has been added in MFC-83xx-N software v2.75 and MFC-832x-S v2.00.

The “max” field was added in MFC-83xx-N software 2.90 and MFC-832x-S v2.11. Cards which receive OID 0xFE0D without the “max” field, may safely assume that the maximum speed is 14.

OID_FAN_SPEED_REQUEST

Message type: 0x10

Message length: 6

Response required: none

Description: This message is used by high-power cards to alter the amount of cooling. In order to have any effect, the requested value must be larger than the current fan speed. This message should be sent at most once per minute.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code – must be 0
oid	1	2	uint16	object OID for fan speed – must be 0xFF11

Field	Offset	Length	Format	Description
dlen	3	1	uint8	Length in bytes of the data value – is 2
value	4	2	uint16	Requested fan speed: • Range 1 to 5 applies to DFR-8310 and DFR-8321. • Range 1 to 14 applies to OG3-FR. • Future frames may have higher ranges.

Number of Occupied Slots

The openGear frame does not have a way to know which slots are occupied. Cards should therefore report the number of slots they occupy. This information may be used by the MFC controller for fan speed management.

The value consists of two 8-bit values, indicating the number of additional slots occupied to the left and right respectively. The value is encoded as:

```
Value = (left << 8) | (right)
```

For example, a card located in slot 6, which occupies slots 5 through 8, would have left=1 and right=2. This value to be reported would therefore be 0x0102.

Since cards generally do not know whether they are in a 10 or 20-slot frame, the value should always be reported for a 20-slot frame. The MFC will adjust values if necessary on a 10-slot frame.

OID_OCCUPIED_SLOTS Parameter

Message type: 0x10

Message length: 6

Response required: none

Description: Cards should define OID 0xFF12 in order to inform the MFC how many slots the card occupies in the openGear frame. A normal parameter may be used, or the card may periodically broadcast this value.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code – must be 0
oid	1	2	uint16	object OID for fan speed – must be 0xFF12
dlen	3	1	uint8	Length in bytes of the data value – is 2
value	4	2	uint16	Number of additional slots occupied (see above)

External Data Objects

This section outlines the transport of External Object data. External Objects allow larger data blocks to be utilized within DashBoard.

External Object Messages

The following messages are required to use external objects.

OGP_GET_EXTERNAL_OBJECT Request

Message type: 0x59

Message length: 5

Response required: OGP_GET_EXTERNAL_OBJECT Response

Description: This requests a fragment of an external object. DashBoard will request external objects only when they are referenced by parameters. DashBoard will first request fragment 0. Each subsequent request will be for the fragment specified in the "next" field of the response. If the next field is set to 0, it is considered the last response in the sequence.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
oid	1	2	uint16	object ID of the requested object
fragment	3	2	uint16	data fragment being requested

OGP_GET_EXTERNAL_OBJECT Response

Message type: 0xD9

Message length: 8 + data_size

Response required: none

Description: This is the required response to an OGP_GET_EXTERNAL_OBJECT request. It provides up to 252 bytes of data, representing one or more fragment of a larger object. DashBoard will first request fragment 0. Each subsequent request will be for the fragment specified in the "next" field. The final fragment must have the "next" field set to 0. Fragment indices are defined by the card. Each response must contain one or more fragments.

Field	Offset	Length	Format	Description
spare	0	1	uint8	return code: • OGP_OK – normal return • OGP_NOTFOUND – object not found
oid	1	2	uint16	object ID of the object being returned
fragment	3	2	uint16	fragment being returned
next	5	2	uint16	next fragment (0 if none)
data_size	7	1	uint8	length of this fragment in bytes
data	8	2	byte[]	data for this fragment (max 252 bytes)

External Object Format and Fragmentation

The first two bytes of the external object content is a type identifier. The remainder of the content (i.e. the object data) is encoded in a type-dependent format.

Field	Offset	Length	Format	Description
objtype	0	2	uint16	object type
objdata	2	varies	byte[]	object data in type-dependent format

The supported object types are as follows:

objtype	Description
0x0001	Constraint
0x0002	File
0x0003	Image
0x0004	OGLML or Index XML document

To support the transfer of objects larger than the openGear CANbus MTU (260 bytes), external objects are requested and transferred in fragments, where each fragment may contain up to 252 bytes. The object content is reassembled by concatenating the fragments, then parsed as appropriate for the object type. The fragmentation scheme is determined by the device. Each fragment is accompanied by the index number of the next available fragment. DashBoard first requests fragment 0, then repeatedly requests the next fragment until the entire object has been received (next fragment is 0). For example, the fragment index could be the byte offset from the start of data, an index into an array of strings, or similar measure.

Note that the 2-byte object type must be sent as the first two bytes of fragment 0.

Encapsulating a Constraint

Constraints are encoded into the External Object data as follows:

Field	Offset	Length	Format	Description
objtype	0	2	uint16	0x0001 – external constraint
ctype	2	1	uint8	constraint type
clen	3	1	uint8	length of constraint data in bytes (ignored). This is here for compatibility with embedded constraint. If constraint data is longer than 255 bytes, this field is ignored by DashBoard.
cdata	4	varies	*	ctype-specific constraint data

Encapsulating a File

Files are encoded External Object data as follows. Note: the data must use the standard openGear file header information defined in the section "File Format for OGP Upload" on page 7–3.

Field	Length	Format
objtype	2 (uint16)	0x0002 – data file
hlen	2 (uint16)	header length (variable length – number of bytes between this field and the 'content' section including the hlen field itself)
ver	1 (uint8)	version of file encapsulation encoding used (0x01)
minver	1 (uint8)	minimum decoder version required to understand the encapsulation (0x01)
no-cache	1 (uint8)	• 0x00 - Cache this external object data (static data) • 0x01 - Do not cache external object data (dynamically generated data)
header type	1 (uint8)	• 0x00 – Do not generate a header (content will follow immediately) • 0x01 – Tail header (a file header will follow at the end of the content) • 0x02 – Generate header (DashBoard will generate an upload header based on the information provided in the following fields)
APPLIES ONLY TO HEADER TYPE 1 (Tail header):		
taillen	2 (uint16)	The number of bytes at the end of the content to move to the beginning of the file to use as a header.
APPLIES ONLY TO HEADER TYPE 2 (Generate header):		

Field	Length	Format
load type	1 (uint8)	See "File Format for OGP Upload" on page 7–3.
bit order	1 (uint8)	
version	2 (uint16)	
compat	4 (uint16)	
target	4 (int32)	
pcount	2 (uint16)	
product[]	14*16	
APPLIES TO ALL:		
content	dlen	data to be uploaded (i.e. the original file)

Encapsulating an Image

Images are encoded into the External Object data as follows:

Field	Length	Description
objtype	2 (uint16)	0x0002 – data file
hlen	2 (uint16)	header length (variable length – number of bytes between this field and the 'content' section including the hlen field itself)
ver	1 (uint8)	version of file encapsulation encoding used (0x01)
objtype	2 (uint16)	0x0003 – image
hlen	2 (uint16)	Header length (currently 2 but may be extended to include more header fields in the future)
Hide status overlay	1 (uint8)	If image is used as device icon, indicates if DashBoard may draw a Status Indicator over the Icon. • 0 — DashBoard will draw a status indicator over the icon. • 1 — DashBoard should not draw a status indicator over the icon (the icon provides its own status information)
Image type	1 (uint8)	• 0 — Image data is embedded in the content section • 1 — Content section contains a URL for the icon
content	*	• If image type = 0: The binary image data encoded as a JPEG, GIF, or PNG • If image type = 1: length (uint16) followed by null-terminated UTF-8-Encoded URL for the image location.

Encapsulating an OGLML Descriptor or Index XML

OGLML and XML documents are encoded into the External Object data as follows:

Field	Length	Description
objtype	2 (uint16)	0x0004 – OGLML or Index XML Descriptor
desctype	1 (uint8)	How to interpret the following content. • 0 — does not reference an OGLML document (end of descriptor) • 1 — contains the ID of another external object (followed by uint16) • 2 — contains the URL to fetch the external object via http or https (UTF-8 String) • 3 — contains the OGLML document itself as an XML file • 4 — contains the OGLML document as a zipped XML file (gzip and deflate supported)
content	*	Contains the content based on the above desctype field.

File Upload

In This Chapter

The openGear protocol provides a simple, robust mechanism for uploading data to devices. This is typically used to upgrade software or firmware on the device, but may be used to upload any file type such as configuration files or license keys. This capability is implemented in DashBoard and may be initiated by clicking the **Upload** button at the bottom of the device page.

This chapter includes the following subjects:

Upload Control

DashBoard 2.1.0 and later provides a method for disabling or redirecting file upload. The card may control the behavior of the DashBoard **Upload** button on the device page using a read-only string parameter with reserved OID **UPLOAD_URL** = 0xFF02. This has the following behavior:

- If the parameter value is a valid URL (starting with http:// or https://), DashBoard will upload files to the specified URL via HTTP POST.
- If the parameter value is an empty string (""), DashBoard will upload files via OGP. Not supported on devices utilizing JSON messaging.
- If the parameter value is "Disabled" (case sensitive), DashBoard will disable the upload button for this device.
- All other values are reserved for possible future use.

The default behavior (if the **UPLOAD_URL** parameter is not defined), is to upload via OGP.

File Upload via HTTP

HTTP file upload is the preferred mechanism for devices with an Ethernet connection. The card may specify the upload location as an URL in a read-only string parameter having reserved OID 0xFF02. DashBoard will upload the file via HTTP POST to the specified location. The posted form will have enctype="multipart/form-data", and will include the selected file with name="uploaded_file".

DashBoard can upload any file using this mechanism; the file is not required to comply with the upload file format required for OGP upload. The receiving device must verify the integrity and validity of the file before accepting it. To make use of the multi-card upload feature in DashBoard 4.1 and later, the standard OGP file header should be used.

Required Response

After the file transfer is complete, the device must return (via HTTP) an ***HTTP 200 OK*** reply with Content-type: text/plain. The reply body must contain one progress report per line in the form:

```
[type]:  [content]
```

where `[type]` is one of the message types given below. These progress messages allow DashBoard to display the status and progress of the file processing on the device. The first message must specify the version; the last message

must define the upload result (Status: OK or Status: ERROR); intervening messages may be inserted to provide text or progress percentage for display to the user. These messages will update the upload dialog.

The messages currently supported are as follows:

Type	Required	Description
Version	Yes	displays the version of HTTP upload dialog (must be 1). Must be the first message.
Timeout	No	the time in milliseconds to wait for the next message (default 5000)
Report	No	message to be reported on the upload dialog
Percent	No	progress (%) to be reported on the upload dialog
Log	No	detailed report message (currently not displayed)
Status	Yes	provides the final result (OK, ERROR, WARN or SILENT) and a status message to be displayed. Must be the last message
RebootOption	No	the automatic reboot option (NONE, AUTO, or PROMPT)

Any other messages are silently discarded. When the Status message is received, DashBoard closes the HTTP connection and the upload dialog. If status is OK, ERROR or WARN, DashBoard then displays the status message in a success or failure dialog as appropriate.

The following is an example of an upload transcript:

```
Version: 1

Timeout: 15000

Report: Preparing for upgrade

Report: Unpacking the files

Report: Checking file integity

Report: Backing up user settings

Report: Computing upgrade size...

Log: There are 45 items to update

Report: Performing upgrade...

Percent: 10

Percent: 20

Percent: 42

Percent: 83

Percent: 100

Report: Flushing buffers

Status: OK Upgrade complete, please reboot now.
```

File Format for OGP Upload

Data files for upload via openGear protocol must be formatted by prepending a 256-byte header to the original file as outlined below. The header contains several information fields that are sent to the card at the beginning of the upload, plus a CRC to ensure that the file has not been accidentally corrupted.

Field	Offset	Length	Description
tag	0	4	File tag = 0x5A50414B
hlen	4	2	header length = 256
header crc	6	2	Header CRC computed for the 248 header bytes following this field. For the algorithm, see "Appendix E: CRC Computation" on page 11–12.
date	8	4	File creation date (seconds since Jan 1, 1970 UTC)
load type	12	1	File load type. The following values are defined: • 0 — Main board software • 1 — Main board FPGA • 2 — Option card software • 3 — Option card FPGA • 10 — Product key • 11 — OEM name table • 16 — Stored configuration file (i.e. DataSafe) Other values are permitted, but will be reported as "Unknown" by DashBoard.
bit order	23	1	Indicates the bit order of the image file (not used): • 0 — normal (most-significant bit first) • 1 — reversed (least-significant bit first)
version	14	2	2-byte version code (major:minor). Should correspond to card software version (0x010B, 0x010C, 0x010D, or 0x010E depending on type). Used by DashBoard to determine warnings for upload dialog.
compat	16	4	Hardware compatibility code. Card may interpret or ignore as appropriate.
target	20	4	Address for storing this file on the target card. Card may interpret or ignore as appropriate.
datalen	24	4	Length of the content field in bytes. This is the total number of bytes to be uploaded.
data crc	28	2	CRC16 of the content field
pcount	30	2	Number of product names provided in the following field (minimum 1).
product[]	32	14 * 16	Products for which this load is valid (up to 14 products)
content	256	dlen	Data to be uploaded (i.e. the original file)

Notes:

- All CRC computations are performed using the algorithm described in "Appendix E: CRC Computation" on page 11–12.
- A file with invalid tag, header crc or data crc is assumed to be corrupted, and is not uploaded.
- DashBoard displays date, load type and data length for confirmation before uploading the file.

- The **version**, **compat**, and **target** fields are sent to the card with the upload and can be used in a card-specific manner to interpret or refuse the upload.
- The content is sent to the file transparently, in sequential order, as 256-byte pages. There is no restriction on the format of the content. The maximum content length that can be sent in a single upload operation is 16 Mbytes (65536 pages of 256 bytes).
- Up to 14 products may be specified in the header. Each product identifier may be up 15 characters (plus null terminator - 16 bytes total).

DashBoard Upload Information

When a file upload is initiated in DashBoard 4.1 and later, the user can optionally select multiple devices to be uploaded. The list of eligible devices is determined primarily by the product name (OID 0x0105). The following options are available to devices to modify how DashBoard treats them when selecting valid upload targets.

Upload Name (Reserved OID 0xFF0A)

This optional parameter provides a means to specify an alternate product name for purposes of file upload. The file upload header provides room for up to 14 product names, which is sufficient in most cases. The names in the header are compared against the product name (OID 0x0105) by default.

If an upload file is applicable to more than 14 product types, then the device should define OID 0xFF0A, typically as a hidden parameter. This is an integer parameter with a choice constraint. The choice constraint is used to provide alternate device names to use for file upload.

When OID 0xFF0A is defined, the headers will be compared as follows. The upload type (byte 12 of the upload header) is used as an index into the choice constraint of the 0xFF0A parameter. If a choice matching the upload type exists, then its associated string is used for matching the header names (instead of OID 0x0105).

If no matching index is found in the choice constraint, the check is repeated using the *value* of parameter 0xFF0A instead of the upload type. If a matching choice is found, its associated string is used for matching the header names. Otherwise, if the parameter's value is not found in the choice constraint, the product name (OID 0x0105) is used.

Most devices do not need to define this parameter. They will compare against OID 0x0105 as before.

For devices which have many variations and product names, yet are based on a common software package, it is also possible to set OID 0x0105 to a common value for all products, and utilize the **WIDGET_NAME_OVERRIDE_APPEND** (see page 3–30) or **WIDGET_NAME_OVERRIDE_REPLACE** (see page 3–31) on OID 0xFF01 to override the device's display name at runtime.

Version Number Encoding

Once a device has been determined to be "compatible" according to its product name, a secondary version check is applied.

This describes version number evaluations/comparison for values stored in OIDs 0x010B (SOFTWARE_REV) and 0xFF05 (BACKWARDS_COMPATIBLE). Version numbers are also compared for FPGA_REV (0x010C), OPTION_SOFTWARE_REV (0x010D), and OPTION_FPGA_REV (0x010E).

Version numbers consist of blocks of alphanumeric characters separated by dots, dashes, underscores, or spaces and are expected to have the following format "[MAJOR].[MINOR].[MICRO]". The major version number is required; each additional block is optional and may extend beyond the Major/Minor/Micro specified here. Version numbers are compared by comparing blocks with each other starting with the major version number.

Comparison Rules:

- Numerical order take precedence over alphabetical order (e.g. "12.0" > "2.0").
- Characters following numerical data are compared in alphabetical order (e.g. "2.0b" > "2.0a" > "2.0").

- If two version numbers have the same value in a block (starting at the left), additional blocks are checked until a difference is found. (e.g. “2.0.1” > “2.0.0”).
- Missing blocks are considered to have a value of 0 (e.g. “2”, “2.0”, and “2.0.0” are equivalent).
- Blocks cannot be skipped (e.g. “2..0” is not considered valid).
- Block separators are all considered equivalent (e.g. “2.0.0”, “2-0-0”, “2_0_0”, “2.0-0”, and “2 0 0” are equivalent).
- Alphabetical comparisons are case-insensitive (e.g. “2.0a” and “2.0A” are equivalent).

OGP File Upload Protocol

For devices which cannot support the HTTP POST method, files may be uploaded via OGP. Note that OGP File Upload is not supported on devices utilizing JSON messaging. The OGP file upload protocol consists of the following requests (and their associated responses):

Command	Msg type	Description
OGP_START_UPLOAD	0x40	initiates a file upload
OGP_UPLOAD_PAGE	0x41	uploads a 256-byte page of the file
OGP_VERIFY_UPLOAD	0x42	prompts device to verify the CRC (upload is complete)
OGP_REBOOT	0x43	instructs the device to reboot
OGP_BOOTLOAD	0x13	indicates device is in bootloader mode

The message content and format is described in detail in the section, “File Upload Messages” on page 7–6.

The procedure for uploading a file is as follows:

1. Send ***OGP_START_UPLOAD*** request to device:
   - on OK – continue
   - on WAIT – retry after the specified interval
   - on INVALID_PRODUCT – stop (incompatible product)
   - on INVALID_UPLOAD – stop (invalid load)
2. while (more data to send):
   - Send ***OGP_UPLOAD_PAGE*** request with next 256 bytes of data
     - › on OK – continue
     - › on INVALID_UPLOAD – stop (upload error)
     - › on INVALID_PAGE – stop (upload error)
3. Send OGP_VERIFY_UPLOAD request:
   - on OK – continue
   - on INVALID_CRC – stop (upload error)
4. Send OGP_REBOOT request.
5. Exit (upload successful).

If the upload is refused with a return code of INVALID_PRODUCT, DashBoard will retry using the next product string in the file header.

If the device enters a reduced-functionality state during software upload, it should periodically send an ***OGP_BOOTLOAD*** message. For more information, see “OGP_BOOTLOAD Message” on page 7–9.

***Note**: The file upload protocol has relaxed response timing requirements. See the specific message definitions for details.*

File Upload Messages

OGP_START_UPLOAD Request

Message type: 0x40

Message length: 40

Response required: OGP_START_UPLOAD Response

Description: This message is sent to initiate a file upload. It contains most of the information provided in the file header: product name, file type, version, creation date and length, etc. The device can use this information to determine whether or not to accept the upload, and how to process the upload if it is accepted. The device should always check the file length to ensure that there is sufficient space to store the file. The CRC should be stored to allow the upload to be verified once it is complete. The version, compatibility, and target fields are sent as extra information; these can be used or ignored as appropriate for each device.

Field	Offset	Length	Format	Description
product	0	16	char[]	product name (null-terminated string)
type	16	1	uint8	file type to be uploaded
spare	17	1	uint8	reserved field – should be zero
version	18	2	uint16	2-byte version code (major:minor)
compat	20	4	uint32	hardware compatibility code
date	24	4	uint32	file creation time (seconds since Jan 1, 1970 UTC)
length	28	4	uint32	file length – total number of bytes to be uploaded
target	32	4	uint32	address for storing this file on the target device
crc	36	2	uint16	CRC16 of the file content
spare	38	2	uint16	reserved field – should be zero

OGP_START_UPLOAD Response

Message type: 0xC0

Message length: 3

Response required: none

Description: This is the required response to an OGP_START_UPLOAD request. The return code indicates whether the upload can proceed. If the upload is refused, the return code should indicate the reason. If the device can accept the upload but needs time to prepare (e.g. to erase flash memory or start a boot loader), it should set the return code to OGP_UPLOAD_WAIT and provide an estimate of the delay time required. DashBoard will send the OGP_START_UPLOAD request again after the specified delay.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK — upload can proceed now • OGP_INVALID_PRODUCT — wrong product name • OGP_INVALID_UPLOAD — unsupported file type • OGP_INVALID_FILESIZE — file is wrong size • OGP_INVALID_STATE — device is in wrong state • OGP_UPLOAD_WAIT — try again after N milliseconds
wait	1	2	uint16	retry time in milliseconds (only used if rc = OGP_UPLOAD_WAIT)

OGP_UPLOAD_PAGE Request

Message type: 0x41

Message length: 260

Response required: OGP_UPLOAD_PAGE Response

Description: This message is used to send a 256-byte page of the file. The request contains the file type, the page number, and page content. The content length can be determined from the message length. The content length is always 256 bytes, except for the last page of the file, which may be less. The file offset to the start of the data is (256 * page) bytes.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
type	1	1	uint8	file type being uploaded
page	2	2	uint16	page number being sent (0 to 65535)
content	4	2	byte[]	page content

OGP_UPLOAD_PAGE Response

Message type: 0xC1

Message length: 4

Response required: none

Description: This is the required response to an OGP_UPLOAD_PAGE request. The return code should normally be OGP_OK, to acknowledge that the page has been received and stored. If the page is not accepted, the return code should indicate the reason. Please note that the file upload is aborted immediately if a page is refused. To ensure that all pages sent are valid, the device must check the file length and type before accepting the upload. If an OGP_UPLOAD_PAGE request is received when an upload is not in progress, the return code should be set to OGP_INVALID_STATE.

DashBoard will wait up to 5 seconds for a response to an *OGP_UPLOAD_PAGE* request.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK – page received and stored • OGP_INVALID_UPLOAD – unsupported file type • OGP_INVALID_PAGE – page number is invalid • OGP_INVALID_STATE – no upload is in progress for this file type
type	1	1	uint8	file type being uploaded
page	2	2	uint16	page number received

OGP_VERIFY_UPLOAD Request

Message type: 0x42

Message length: 2

Response required: OGP_VERIFY_UPLOAD Response

Description: This message is sent to indicate that the upload is complete, and to instruct the device to verify that the file has been received and stored correctly.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
type	1	1	uint8	file type being uploaded

OGP_VERIFY_UPLOAD Response

Message type: 0xC2

Message length: 2 + data

Response required: none

Description: This is the required response to an OGP_VERIFY_UPLOAD request. The return code indicates if the upload was successful or not. If no upload is in progress, the return code should be set to OGP_INVALID_STATE.

The device may verify the upload in any suitable manner. Typically, it should read back the file from its internal storage, compute the CRC on the stored data, and compare the result with the CRC value sent in the *OGP_START_UPLOAD* request. For a large file, this may be a time-consuming process. DashBoard will wait up to 15 seconds for a response to an *OGP_VERIFY_UPLOAD* request.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK – file verified OK • OGP_INVALID_CRC – CRC check failed • OGP_INVALID_STATE – no upload is in progress for this file type • OGP_REQUEST_DENIED – Update the status of the upload with the provided text and send OGP_VERIFY_UPLOAD request again. • UPLOAD_WAIT – Wait n milliseconds and send OGP_VERIFY_UPLOAD request again.
type	1	1	uint8	file type being uploaded
data	2	dlen	char[] or uint16 (depending on rc)	• If return code is OGP_REQUEST DENIED, the device must provide a null-terminated UTF-8 String with updated status text for the upload (char[]). • If return code UPLOAD_WAIT, retry time in milliseconds (uint16) ***Note**: Used only if rc == OGP_REQUEST_DENIED or UPLOAD_WAIT*

OGP_REBOOT Request

Message type: 0x43

Message length: 0

Response required: OGP_REBOOT Response

Description: This message instructs the device to reboot. This request is sent at the end of the file upload sequence, or when the Reboot button on the DashBoard device page is clicked.

Field	Offset	Length	Format	Description
n/a	0	0	n/a	message has no content

OGP_REBOOT Response

Message type: 0xC3

Message length: 1

Response required: none

Description: This is the required response to an OGP_REBOOT request. The return code indicates whether the request was successful:

- If the request is accepted, the device should return OGP_OK, then reboot.
- If the request is denied, the device should return OGP_REQUEST_DENIED, plus an optional string providing the reason that the request is not accepted.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK — request accepted, device rebooted • OGP_REQUEST_DENIED — reboot request denied
reason	1	N	char[]	null-terminated UTF-8 string providing the reason that the reboot request is denied

OGP_BOOTLOAD Message

Message type: 0x13

Message length: variable

Response required: OGP_ADDR_TRAP

Description: This message should be sent periodically (every 5 to 10 seconds) when the device is in a reduced functionality state waiting for a software upload. The message must contain the product name as an UTF-8 string, to allow DashBoard to identify the device and provide a skeleton device page through which to upload software. In this state, DashBoard will not query the device for parameter and menu information.

Field	Offset	Length	Format	Description
text	0	any	char[]	product name encoded as null-terminated UTF-8 string

DataSafe

In This Chapter

DataSafe is a feature of the openGear Frame Network Controller which allows a customer to pull a failed openGear card out, replace it with a working version of the same card and have all of the old card's parameter values restored to the new card. This feature can be enabled or disabled on a slot-by-slot basis, and currently defaults to OFF.

***Note**: DataSafe will not preserve card data if the frame power is interrupted. Cards must still provide non-volatile storage of their parameters.*

***Note**: DataSafe is a feature of the Frame Network Controller and applies to openGear cards only. It is not applicable to DashBoard Connect devices.*

DashBoard also allows a customer to save card parameters to a file, and later to restore them to one or more cards of the same type. Multiple configuration sets can be stored, if desired. This feature can also be used to configure a large number of cards identically from a single stored configuration.

DataSafe and DashBoards parameter saving feature are designed to work with existing openGear cards in an openGear Frame. To improve compatibility between cards and this feature, this chapter explains the various control mechanisms available to cards.

Using the DataSafe Feature

Operation

DataSafe works by monitoring the card parameters as they are relayed through the Frame controller. It keeps a copy of the most recent value of each parameter values in RAM. When a new card is inserted, the product name and version number OIDs are retrieved, and are compared against the saved parameters. If they match, the saved parameters are restored to the card, and then the normal monitoring of card parameters resumes.

In case of a mismatch, for example if a different type of card is inserted, DataSafe will monitor the new card parameters, while keeping the original parameters unmodified. A mismatch error will be flagged on the DataSafe status tab. The user can clear this mismatch by clicking the Update Slot button, which will cause the newly collected parameters to replace the original set.

Sets of configurations for cards can also be saved/restored to/from files in DashBoard.

Parameter Restore Order

Parameter values will be restored to a card in the order they were returned to the control system via the OGP_GET_PARAM_OIDS response message. The order is provided by the OLD CARD, not the NEW CARD. DataSafe does not restore parameters which are READ_ONLY, and it also does not restore WIDGET_BUTTON_PROMPT or WIDGET_BUTTON_NO_PROMPT types (i.e. stateless buttons).

A card can optionally limit the range of OIDS that will be restored, using the RESTORE_START and RESTORE_STOP OIDs defined in the reserved OID table below. Note that only one such range per card may be defined.

When performing a restore, DataSafe will send RESTORE_START with a value of 1 to mark the beginning of the restore. Similarly, it will send RESTORE_STOP with a value of 1 at the completion of the restore process. These reserved OIDs are sent even if the card does not define them in its OID list.

Traps during Restore

Cards must not send certain TRAP messages during DataSafe restore process. Specifically, trap types OGP_RESTART and OGP_PARAM_CHANGED must not be sent while DataSafe restore is taking place. Any

such traps must be withheld until the restore is complete (marked by the RESTORE_STOP reserved OID). A good practice is to disable the traps when OID 0xFF07 is received, and restore them after OID 0xFF08 is received.

Edit Permission Parameter (Reserved OID 0x0601)

During a DataSafe restore, the value of the Edit Permission parameter is set to 0 (unlocked) so that the data restore may proceed. After the restore is complete, the Edit Permission value will be restored to its previous setting.

Cards that wish to disable DataSafe must return a value of **-1** of RESTORE_SET_DELAY (Reserved OID 0xFF06)

If a card has another mechanism for preventing parameter modification (such as a jumper or DIP switch), DataSafe will be unable to restore parameters. An error condition will be flagged.

DataSafe Name Append (Reserved OID 0xFF09)

This optional parameter affects how DataSafe determines whether two cards are “compatible”. When present, this parameter specifies an addition to the name for the card, appended to the product name (OID 0x0105) by DataSafe only. This has no effect on the user-visible card name.

Most cards do not need to define this parameter. DataSafe will use the regular product name (OID 0x0105) by default.

The intention of this parameter is to prevent DataSafe compatibility between products sharing the same product name (OID 0x0105). This typically happens when there are product variants, distinguished using NAME_OVERRIDE, for which the parameters are not compatible. Setting OID 0xFF09 to different strings for these variants will allow DataSafe to recognize the variants as different products.

Using BACKWARDS_COMPATIBLE (0xFF05)

The backwards compatible OID allows a card to change its version number and indicate to DataSafe that its parameter OIDs and values are still compatible with a previous version. If no value is provided for this OID, only the current software version (as specified by OID 0x010B) is considered compatible.

When determining if two cards have OID compatibility, DashBoard will check the information on both cards. If either card indicates compatibility, they are considered compatible. See the following table of examples for details:

Card A SOFTWARE_REV (0x010B)	Card A BACKWARDS_COMPATIBLE (0xFF05)	Card B SOFTWARE_REV (0x010B)	Card B BACKWARDS_COMPATIBLE (0xFF05)	Compatible
1.0	—	1.0	—	YES
1.0	—	1.5	—	NO
1.0	—	1.5	1.0	YES
1.0	—	1.5	0.8	YES
1.5	—	2.0	1.0	YES
1.0	—	2.0	1.0a	NO
1.0a	—	2.0	1.0	YES
1.5	—	1.0	—	NO
1.5	1.0	1.0	—	YES
1.5	0.8	1.0	—	YES
2.0	1.0	1.5	—	YES
2.0	1.0a	1.0	—	NO
2.0	1.0	1.0a	—	YES

***Note**: It is NOT valid for a card to have a **BACKWARDS_COMPATIBLE** value greater than its **SOFTWARE_REV** value (e.g. if Card A specifies **SOFTWARE_REV** 1.0 and **BACKWARDS_COMPATIBLE** 1.5, its **BACKWARDS_COMPATIBLE** value is ignored).*

Diagnostics

The DataSafe tab on the MFC device page provides status information for each slot. The left-most column indicates the status of the primary stored configuration. It cycles through the following messages:

```
Product Name :: Product Version :: loading...

Product Name :: Product Version :: restoring..

Product Name :: Product Version :: Done
```

If an error occurs during any of the above stages, the status text will indicate “Unable to ...” and a DataSafe error will be flagged. The errors correspond closely to the standard OGP commands for retrieving the number of OIDs, the list of OIDs, the descriptors and values. DataSafe operation for the slot is suspended, until a new card is inserted.

When a non-matching card is inserted into a slot for which configuration has been stored, DataSafe status appears in the right hand status column instead of the left-most. The Update Slot button between these columns can be used to overwrite the primary configuration with that of the new card. If the button is not pressed within 24 hours, it will be automatically activated.

DataSafe can be disabled on a per-slot basis using the Disable button at the far right of each slot. Even when disabled, DataSafe will still monitor each slot, but it will not restore parameters.

The “Mask Warning” box can be used to prevent a slot error from being reported in the overall card status.

SNMP

In This Chapter

The Simple Network Management Protocol (SNMP) is an Internet-standard protocol for managing devices on IP networks. The openGear network card provides an optional SNMP agent which enables SNMP management of participating cards within an openGear frame.

***Note**: SNMP is a feature of the openGear Frame Network Controller, and not applicable to DashBoard Connect devices. There is no support in the JSON messaging protocol for SNMP-related messages.*

This is useful to many broadcasters, because it allows their openGear device management to be integrated within a facility-wide SNMP-based management system.

This chapter describes the SNMP implementation for openGear, and defines:

- openGear MIBs for frames and generic cards
- guidelines for developing a card-specific MIB
- openGear to SNMP object ID mappings
- SNMP-specific openGear messages

This chapter is not intended as an SNMP tutorial. For a good introduction to SNMP concepts and administration, see the book ***Essential SNMP (2nd edition)*** by Mauro, D.R & K.J. Schmidt. O'Reilly, 2009.

openGear SNMP Agent

An SNMP management system comprises an SNMP manager (the monitoring and control system) and one or more SNMP agents, which provide management information about specific devices. Most SNMP communications follow a request-response model, in which the agent reports or changes the value of a parameter in response to a GET or SET command from the manager. However, the agent can also send unsolicited asynchronous messages (notifications or TRAPs) to notify the manager of an important event on the device.

The SNMP communications architecture is shown in **Figure 9.1**.

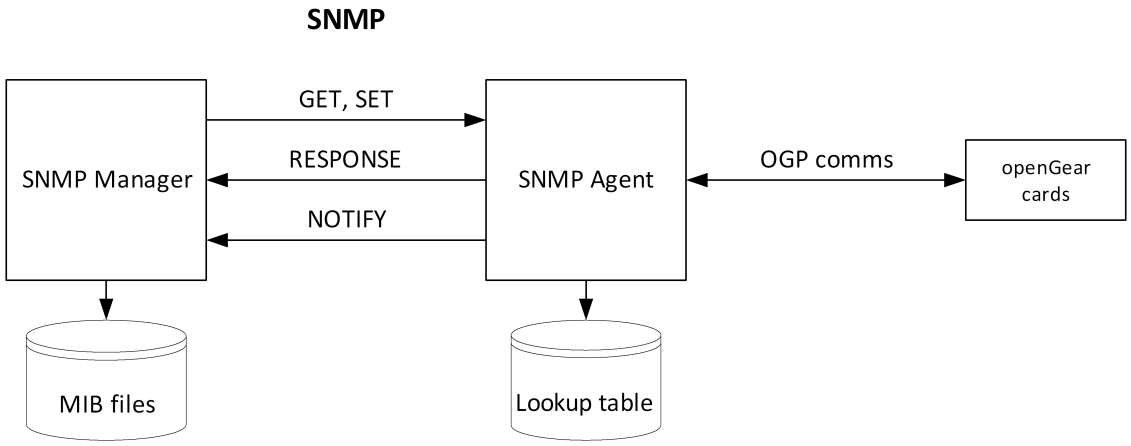


***Figure 9.1** - SNMP Communications Architecture*

Management data within SNMP is structured as a tree, as illustrated in **Figure 9.2**. The subtree defining all of the data for a specific device is called a management information block (MIB). To query a device, an SNMP management system typically traverses the tree, starting at the root node of the device MIB, requesting each object in turn using a "GET" or "GET_NEXT" command. This query retrieves values only; to interpret the values, the management system must have access to a text file that describes the MIB. The term MIB is also used to refer to these MIB-description files.

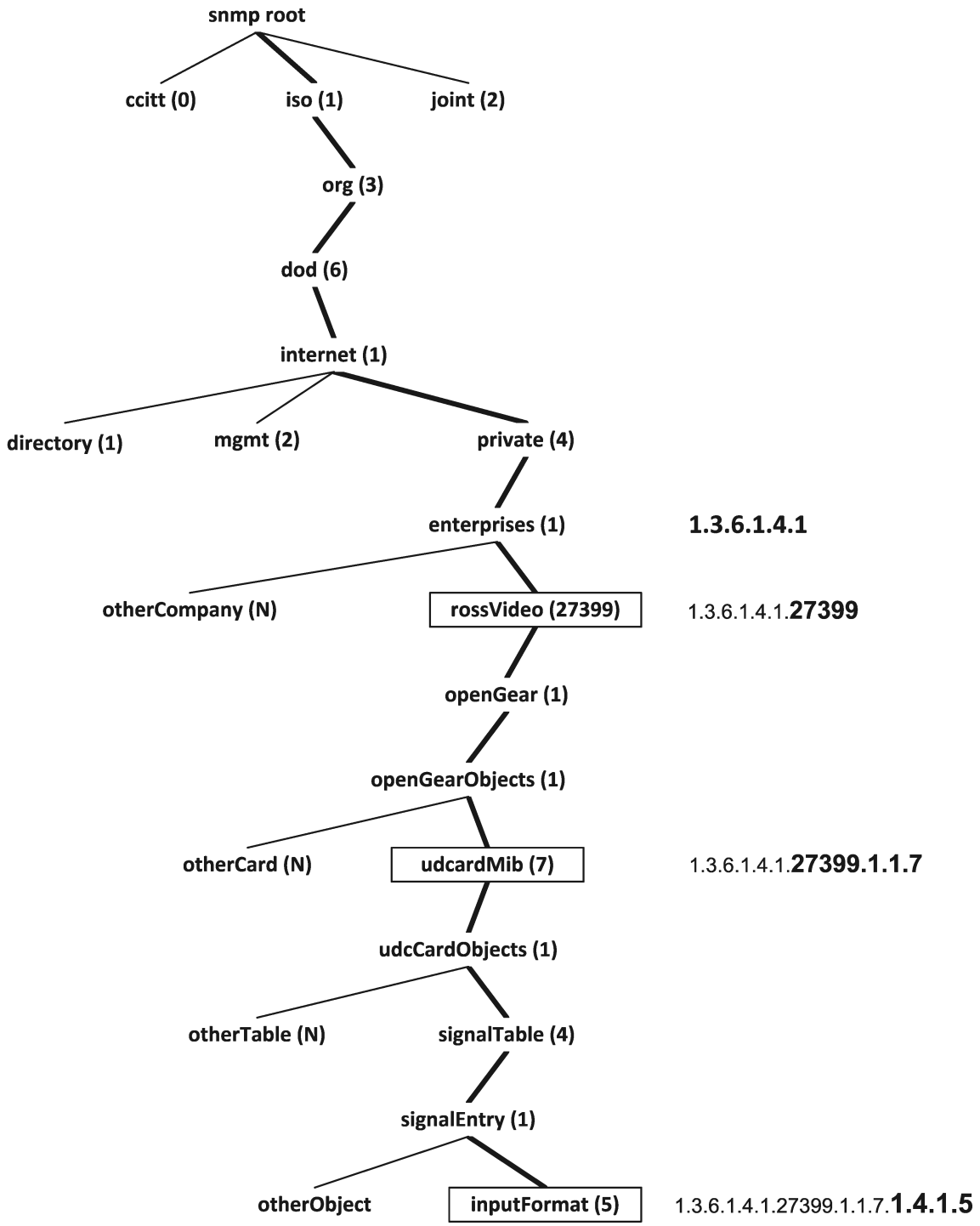


Figure 9.2 *- SNMP Object Tree*

Each node in the SNMP tree is assigned an object ID (OID). SNMP object IDs are represented as dotted decimal strings. The OID for a specific node is obtained by appending its node number to its parent node's OID. To allow inter-operation of arbitrary assemblies of equipment, all objects are organized into a single tree. Corporations and private organizations that want to define MIBs must obtain an ***enterprise number*** by registering with the Internet Assigned Numbers Authority (IANA). The enterprise number defines a private branch of the SNMP tree (under the ***private.enterprises*** node) to be used for the organization's MIBs. For example, Ross Video's enterprise number is 27399, so all Ross Video MIBs are contained within the sub-tree with root node:

iso.org.dod.internet.private.enterprises.rossVideo = 1.3.6.1.4.1.27399

You can obtain an SNMP enterprise number by applying to IANA at http://pen.iana.org.

The openGear SNMP agent uses a lookup table to cross-reference SNMP OIDs to openGear OIDs. When the agent receives a request for a specific SNMP OID, it finds the corresponding openGear OID in the lookup table, and relays the request to the card via OGP. The lookup table must be provided by the card; the agent requests the table when it first establishes communications with the card, or whenever the card sends an OGP_RESTART trap. The message format for encoding the lookup table is provided in the section, "openGear to SNMP Object ID Mapping" on page 9–11.

In order to provide full SNMP monitoring and control support for a specific openGear card, you must:

- define an SNMP MIB that complies with the structure defined in "openGear MIBs" on page 9–3.
- create a lookup table that specifies the SNMP OID for each openGear parameter to be exposed via SNMP.
- provide this table to the SNMP agent when requested via the openGear protocol.
- publish a MIB-description file in standard format to allow third-party management tools to interpret values returned by the SNMP agent.

Cards that do not provide specific SNMP support will still be reported by the SNMP agent using the generic openGearCard MIB described in the next section.

openGear MIBs

Basic SNMP support for openGear is provided under the RossVideo enterprise number, and is defined through five MIBs (Management Information Bases):

- ***rossVideo***: defines Ross Video's enterprise number
- ***openGear***: defines openGear as a branch of the Ross Video MIB
- ***openGearNode***: defines the parameters available through a single access point (IP address)
- ***openGearFrame***: defines parameters specific to a frame
- ***openGearCard***: defines common parameters shared by most openGear cards

The most recent version of the Ross Video SNMP MIB files are available for download from the Ross Video website at http://www.rossvideo.com/terminal-equipment/opengear/control-monitoring/snmp.html.

***Note**: SNMP is a feature of the openGear Frame Network Controller, and not applicable to DashBoard Connect devices.*

The top-level structure of the openGear MIB is shown in **Figure 9.3**. Essentially, this MIB just defines openGear to be a branch of the rossVideo MIB, and defines openGearObjects as the first branch of the openGear MIB.

```
rossVideo::= { enterprises 27399 }
      openGear ::= { rossVideo 1 }
            openGearObjects ::= { openGear 1 }
```

***Figure 9.3** - Structure of the **openGear** MIB*

Three specific openGearObjects have been defined to manage information at the frame level. The openGearNode MIB, shown in **Figure 9.4**, defines the parameters available through a single access point (IP address). These objects primarily relate to the network configuration for the network card.

```
openGearNodeMib ::= { openGearObjects 1 }
      openGearNodeObjects (1)
            openGearNodeName (1)
            openGearNodeSoftwareVersion (2)
            openGearNodeUseDHCP (3)
            openGearNodeIPAddress (4)
            openGearNodeNetMask (5)
            openGearNodeGateway (6)
            openGearNodeTimeServer (7)
```

Figure 9.4 *- Structure of the* ***openGearNode*** *MIB*

The openGearFrame MIB, shown in **Figure 9.5**, provides access to the status parameters for the frame (power, fan state, alarms, etc.). This is defined separately from the openGearNode to support a possible future extension in which several frames could be accessed through a single IP connection. The frame data are organized in tables, indexed by the ***openGearFrameIndex*** (currently always 1). The openGearFrame MIB also defines notifications (traps) to be sent when the fan door is opened or closed, or when the alarm status of the frame changes.

```
openGearFrameMib ::= { openGearObjects 2 }
      openGearFrameNotifications (0)
            openGearFrameStatusEvent (1)
            openGearFrameFanDoorEvent (2)
      openGearFrameObjects (1)
            openGearNumberOfFrames (1)
            openGearFrameTable (2)
                  openGearFrameEntry (1)
                        openGearFrameIndex (1)
                        openGearFrameVersion (2)
                        openGearFrameName (3)
                        openGearFrameSupplier (4)
                        openGearFrameProduct (5)
                        openGearFrameBoardRev (6)
                        openGearFrameSoftware (7)
            openGearFrameHardwareTable (3)
                  openGearFrameHardwareEntry (1)
                        openGearFrameStatus (1)
                        openGearFrameFanDoor (2)
                        openGearFrameFanSpeed (3)
                        openGearFrameAudioAlarm (4)
            openGearFramePowerTable (4)
                  openGearFramePowerEntry (1)
                        openGearFramePower (1)
                        openGearFramePSU1Current (2)
                        openGearFramePSU2Current (3)
                        openGearFrameMaxSlotCurrent (4)
                        openGearFramePSU1Temp (5)
                        openGearFramePSU2Temp (6)
                        openGearFrameAmbientTemp (7)
```

Figure 9.5 *- Structure of the* ***openGearFrame*** *MIB*

The openGearCard MIB, shown in **Figure 9.6**, defines a small set of parameters common to all openGear cards. This allows the card type and basic status to be displayed even if the card does not provide SNMP information to the SNMP agent. These parameters are organized in a table that is indexed by the frame index (***openGearFrameIndex***), and the slot number (***openGearCardIndex***).

```
openGearCardMib ::= { openGearObjects 3 }
      openGearCardNotifications (0)
            openGearCardOnlineEvent (1)
            openGearCardOfflineEvent (2)
      openGearCardObjects (1)
            openGearNumberOfCards (1)
            openGearCardTable (2)
                  openGearCardEntry (1)
                        openGearCardIndex (1)
                        openGearCardSupplier (2)
                        openGearCardProduct (3)
                        openGearCardSoftwareRev (4)
                        openGearCardOnline (5)
                        openGearCardOid (6)
```

***Figure 9.6** - Structure of the **openGearCard** MIB*

The SNMP agent reports the following values for each parameter. Supplier, product and software revision are derived from openGear parameters with the reserved objectIDs defined in "Appendix D: Reserved Object IDs" on page 11–6.

SNMP Object	openGear OID	Description
openGearCardIndex	n/a	slot number
openGearCardSupplier	0x0102	supplier name
openGearCardProduct	0x0105	product name (same as reported on DashBoard)
openGearCardSoftwareRev	0x010B	software revision
openGearCardOnline	n/a	card status: • true (1) — card is online • false (2) — card is offline
openGearCardOid	n/a	fully qualified root OID of the card MIB

The openGearCard MIB also defines notifications to be sent when a card goes offline or comes back online. The card is determined to be offline if it has not sent any message for 10 seconds, or if it has failed to respond to a query from the agent.

Defining a Card-specific MIB

The openGear SNMP agent expects each card MIB to have a well-formed structure in order to correctly process SNMP queries. This section describes the required structure of a card-specific MIB. The examples are derived from the UDC8225-CARD MIB.

The high-level structure of a typical card MIB is shown in **Figure 9.7**. The root node of the MIB is the top level node containing all of the card's management information. Each openGear partner should define card MIBs relative to their own enterprise number. For example, the UDC card MIB is defined as branch 7 of the openGearObjects node under the Ross Video number, as shown in**Figure 9.2**, "- SNMP Object Tree" on page 9–2.

Under the root node, the card MIB must have three major branches (notifications, objects, and conformance), with branch numbers 0, 1 and 2, respectively, as shown in **Figure 9.7**:

```
myCardMib (root node)
      myCardNotifications (0)
            notificationOne (1)
            notificationTwo (2)
            more notifications
      myCardObjects (1)
            numCards (1)
            tableOne (2)
                  tableOneEntry (1)
                        objectOne (1)
                        objectTwo (2)
                        more objects
            tableTwo (3)
                  tableTwoEntry (1)
                        objects
            arrayGroupOne (4)
                  numElements (1)
                  groupOneTableOne (2)
                        groupOneTableOneEntry (1)
                              groupOneIndex (1)
                              object (2)
                              more objects
                  groupOneTableTwo (3)
                        groupOneTableTwoEntry (1)
                              object (1)
                              more objects
                  more tables for this group
            more groups or tables for this card
      myCardConformance (2)
            myCardMibCompliances (1)
                  myCardCompliances (1)
            myCardMibGroups (2)
                  myCardInfoGroup (1)
                  myCardNotifyGroup (2)
```

Figure 9.7 *- High-level Structure of a Typical Card MIB*

Notifications

The ***notifications*** branch defines notifications or traps to be sent asynchronously by the SNMP agent to report change events on the card. Each event must be associated with an SNMP parameter; the SNMP agent sends a notification to the configured trap address when the associated parameter changes value. The following example shows the syntax for defining a notification event. In this example, the ***udc8225InputStatusEvent*** will be sent whenever the ***udc8225InputStatus*** parameter changes. The notification includes the object ID of the parameter that changed, including the frame number and slot index.

```
udc8225InputStatusEvent   NOTIFICATION-TYPE
   OBJECTS { udc8225InputStatus }
   STATUS current
   DESCRIPTION
      "An indication that status of the video input has
       changed. The status value is in udc8225InputStatus"
  ::= { udc8225CardNotifications 3 }
```

Objects

The ***objects*** branch defines configuration and status parameters for the card. The first object in this branch (OID myCardMib.1.1) must be a read-only integer parameter representing the number of cards of this type in the frame. The SNMP agent determines this automatically from the number of cards reporting the specified card MIB. The card count parameter must be defined using the following syntax:

```
udc8225NumberOfCards    OBJECT-TYPE
   SYNTAX      Integer32
   MAX-ACCESS  read-only
   STATUS      current
   DESCRIPTION
     "The total number of UDC-8225 cards in the frame."
   ::= { udc8225CardObjects 1 }
```

Structure of a Parameter Table

The configuration and status parameters for each card must be organized into one or more tables. There is no firm rule for assigning parameters to tables; most cards will define several tables roughly corresponding to the menu organization in DashBoard. Array parameters must be handled separately as described below. Each ***table*** is defined as a sequence of ***entries*** indexed using frame and slot index, where each ***entry*** contains one or more parameters.

Figure 9.8 shows the structure of one table for the UDC. The ***entry*** is defined as subnode (1) of the table; each parameter is defined as a subnode of the entry. The frame and slot index are appended to the object ID to provide the unique SNMP address for a parameter on a specific card.

For example, for a UDC in frame 1, slot 8, the input format would be accessed as OID 1.3.6.1.4.1.27399.1.1.7.1.4.1.5.1.8, where:

- 1.3.6.1.4.1 is the private.enterprises node
- 27399.1.1.7 specifies the UDC MIB, relative to the enterprises node
- 1.4.1.5 specifies the table (Signal) and parameter (InputFormat)
- 1.8 specifies the frame index and slot index

```
udc8225SignalTable ::= { udc8225CardObjects 4 }
      udc8225SignalEntry ::= { udc8225SignalTable 1 }
            udc8225ReferenceStatus ::= { udc8225SignalEntry 1 }
            udc8225ReferenceFormat (2)
            udc8225GenlockState (3)
            udc8225InputStatus (4)
            udc8225InputFormat (5)
            udc8225AudioStatus (6)
```

Figure 9.8 *- Structure of a Typical Table*

Each table must be defined following the syntax shown **Figure 9.9**.

In the example, ***Udc8225SignalEntry*** defines a custom data type containing a sequence of objects. The table itself, ***udc8225SignalTable***, is defined as a sequence of objects of this type, and is assigned an object ID immediately under the card objects node. Each table entry (row), ***udc8225SignalEntry,*** is defined to be an object of this type, and is assigned the first branch under the table node. The entry definition also specifies that entries are indexed by ***openGearFrameIndex*** and ***openGearCardIndex***. This ensures that there is an entry for each card of this type at this IP address, and that it is directly addressable by frame and slot number. Note that ***openGearFrameIndex*** and ***openGearCardIndex*** are defined in the openGearFrame MIB and the openGearCard MIB, respectively.

```
udc8225SignalTable     OBJECT-TYPE
   SYNTAX      SEQUENCE OF Udc8225SignalEntry
   MAX-ACCESS  not-accessible
   STATUS      current
   DESCRIPTION
      "A table containing signal status for all UDC-8225
       cards at this node."
   ::= { udc8225CardObjects 4 }

udc8225SignalEntry     OBJECT-TYPE
   SYNTAX      Udc8225SignalEntry
   MAX-ACCESS  not-accessible
   STATUS      current
   DESCRIPTION
      "A conceptual row within the 'udc8225SignalTable'."
   INDEX    { openGearFrameIndex, openGearCardIndex }
   ::= { udc8225SignalTable 1 }

Udc8225SignalEntry ::= SEQUENCE {
   udc8225ReferenceStatus     INTEGER,
   udc8225ReferenceFormat     DisplayString,
   udc8225GenlockState        INTEGER,
   udc8225InputStatus         INTEGER,
   udc8225InputFormat         DisplayString,
   udc8225AudioStatus         DisplayString
```

Figure 9.9 *- Defining a Table*

Each object within a table entry represents a single openGear parameter. To ensure that the SNMP agent is able to correctly respond to SNMP get and set commands, it is essential that the object definitions in the MIB correspond closely to the openGear parameter definitions:

- INT16 and INT32 parameters must be represented as INTEGER or Integer32 objects.
- STRING parameters must be represented as **DisplayString**.
- FLOAT32 parameters must be represented as **DisplayString** objects; the floating point value will be UTF-8 encoded in the string using the precision defined by the parameter descriptor.

If the SNMP agent receives an SNMP set command containing the wrong data type for a parameter, the command will be rejected.

The SNMP object definitions should also match the parameter constraints as far as possible. That is, a CHOICE constraint should map to an SNMP enumeration, a RANGE constraint should map to the same range in SNMP, etc.

The following example of SNMP object definitions demonstrates the syntax for several example variables:

```
--  INT16 with CHOICE constraint
--
udc8225OutputFormat    OBJECT-TYPE
  SYNTAX      INTEGER  {
    output480i5994  (0),
    output720p5994  (1),
    output1080i5994 (2),
    output576i50    (3),
    output720p50    (4),
    output1080i50   (5)
  }
  MAX-ACCESS  read-write
```

```
  STATUS      current
  DESCRIPTION
    "The video output format for this conversion card."
  DEFVAL  { output480i5994 }
  ::= { udc8225OutputEntry 1 }
--
--  INT16 with RANGE constraint [0, 200]
--
udc8225VideoGain    OBJECT-TYPE
  SYNTAX      Integer32 (0..200)
  MAX-ACCESS  read-write
  STATUS      current
  DESCRIPTION
    "The video gain of the output proc amp, in percent."
  DEFVAL  { 100 }
  ::= { udc8225ProcAmpEntry 1 }
--
--  FLOAT32 with RANGE constraint and precision 3
--
udc8225AzimuthValue    OBJECT-TYPE
  SYNTAX      DisplayString (SIZE(6))
  UNITS       "Radians"
  MAX-ACCESS  read-write
  STATUS      current
  DESCRIPTION
    "The azimuth value, the range is 1.000 - 10.000."
  DEFVAL  { "1.000" }
  ::= { udc8225ProductEntry 11 }
--
--  Read-only STRING
--
udc8225InputFormat    OBJECT-TYPE
  SYNTAX      DisplayString
  MAX-ACCESS  read-only
  STATUS      current
  DESCRIPTION
    "The format of the input video signal for this card."
  ::= { udc8225SignalEntry 5 }
)
```

Array parameters

Array parameters must be organized into groups as shown in **Figure 9.7** (see page 9–6). All arrays in each group must have the same number of elements. Each group must contain the array size and one or more tables containing array parameters. The full syntax for defining an array group is shown in the following example:

```
--
--  Define the group
--
audioInputGroup
   OBJECT IDENTIFIER ::= { myCardObjects 3 }
```

```
--
--  Define the number of elements in the group
--
audioInputCount    OBJECT-TYPE
   SYNTAX      Integer32
   MAX-ACCESS  read-only
   STATUS      current
   DESCRIPTION
     "The number of audio inputs on each card."
   ::= { audioInputGroup 1 }
--
--  Define a table in the group
--  Indexed by frameIndex, cardIndex, audioIndex
--
audioInputTable     OBJECT-TYPE
    SYNTAX      SEQUENCE OF AudioInputEntry
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
      "A table containing audio input settings. "
    ::= { audioInputGroup 2 }

audioInputEntry     OBJECT-TYPE
    SYNTAX      OpenGearCardEntry
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
      "A row within the audioInputTable."
    INDEX { openGearFrameIndex, openGearCardIndex, audioIndex }
    ::= { audioInputTable 1 }

AudioInputEntry ::= SEQUENCE {
    audioIndex         Unsigned32,
    audioGain          Integer32,
    audioDelay         Integer32,
    audioEnabled       TruthValue
}--
--  Define the object containing the index
--
audioIndex    OBJECT-TYPE
    SYNTAX      Unsigned32
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
      "Audio input channel index."
    ::= { audioInputEntry 1 }
```

The first object in each group (OID myGroup.1) must be a read-only integer parameter representing the number of elements in each array. The SNMP agent determines the value of this object automatically based on the array size.

Tables are defined using the same syntax outlined above, with the addition of a third index. The first object in the first table in each group (OID = myGroup.2.1.1) must be a read-only integer object representing the array index.

Conformance

The ***conformance*** branch of the MIB defines groups of required and optional parameters, and any compliance or conformance issues (e.g. parameters which may not be defined for certain variants of the device). This section is primarily for information purposes and does not need to be specifically linked to the openGear parameters. To see an example, download the Ross Video SNMP MIB files and examine one of the UDC MIBs (UDC8225-CARD.mib or UDC8625-CARD.mib). The Ross Video MIB files are available at http://www.rossvideo.com/terminal-equipment/opengear/control-monitoring/snmp.html.

openGear to SNMP Object ID Mapping

In order to provide plug-and-play SNMP capability, each card must provide the SNMP agent with a lookup table to cross-reference SNMP OIDs to openGear OIDs. Specifically, each card must define:

- the root node of its MIB.
- the SNMP object ID for each openGear parameter to be exposed through SNMP.
- the SNMP object ID of the trap associated with each openGear parameter.

When the SNMP agent first establishes communication with a card, or when it receives the restart trap from the card, it will query the card for this information using messages, as described in "SNMP Agent Initialization" on page 9–12.

The root node must be defined relative to the ***private.enterprises*** node. For the UDC MIB described above and shown below, the relative object ID of the root node is "27399.1.1.7":

```
rossVideo (27399)
     openGear (1)
          openGearObjects (1)
               udcCardMib (7)
```

Each SNMP object ID must be defined relative to the root node of the card MIB. This will always be a 4-part OID. For example, the inputFormat parameter shown below should report its SNMP OID as "1.4.1.5":

```
udcCardObjects (1)
     signalTable (4)
          signalEntry (1)
               inputFormat (5)
```

Array parameters always have a 5-part relative OID. For the audioDelay parameter shown below, the relative OID is "1.3.2.1.3":

```
myCardObjects (1)
     audioInputGroup (3)
          audioInputTable (2)
               audioInputEntry (1)
                    audioDelay (3)
```

SNMP trap OIDs should also be specified as relative values; these are always 2-part OIDs. For the example shown below, the inputStatus parameter should report its SNMP trap OID as "0.3":

```
udc8225CardNotifications (0)
     udc8225InputStatusEvent (3)
```

Some parameters may not be exposed to SNMP, and most parameters will not have an associated trap. For these parameters, the card should return OGP_NOTFOUND.

The SNMP agent automatically creates index and size objects (e.g. for number of cards). It is necessary to define these objects in the MIB, but not to associate them with an openGear parameter.

SNMP Agent Initialization

The SNMP agent creates or recreates the MIB and the OID cross-reference table when it first establishes communication with the card, or whenever the card sends an OGP_RESTART trap. The initialization protocol involves several of the standard parameter requests, and three SNMP-specific requests:

Command	Msg type	Description
OGP_GET_SNMP_BASE	0x56	requests the card MIB OID. (see page 9–13)
OGP_GET_SNMP_OID	0x57	request the SNMP OID for a parameter. (see page 9–13)
OGP_GET_SNMP_TRAP	0x58	requests the SNMP trap OID for a parameter. (see page 9–14)
OGP_GET_SNMP_TYPE	0x5B	requests a hint to tell the SNMP agent the data type. (see page 9–15)

The content and format of these requests and their responses is described in "SNMP-Related Messages" on page 9–13.

The initialization protocol is as follows:

1. Request basic card information defined by the openGearCard MIB:
   a. Request productName (parameter ID 0x0105).
   b. Request supplierName (parameter ID 0x0102).
   c. Request software revision (parameter ID 0x010B).
   d. Request card MIB (***OGP_GET_SNMP_BASE***).
2. If the card provides a MIB OID:
   a. Request number of parameters (***OGP_GET_NUMPARAMS***).
   b. Request parameter OIDs (***OGP_GET_PARAM_OIDS***).
   c. For each parameter:
      - Request descriptor (***OGP_GET_DESCRIPTOR***).
      - Request value (***OGP_GET_PARAM***).
      - Request SNMP OID (***OGP_GET_SNMP_OID***).
      - Request SNMP trap (***OGP_GET_SNMP_TRAP***).

If the card does not provide a MIB (i.e. returns **OGP_UNSUPPORTED**), the agent does not request any parameter info and does not communicate further with the card until the card sends an **OGP_RESTART** trap.

The SNMP agent ignores invalid information, including:

- OIDs that do not parse correctly as dotted decimal strings.
- OIDs that do not meet the requirements given in the previous section.
- references to reserved OIDs (e.g. myCardMib.1.1).

The SNMP agent ignores parameter constraints and does not request external objects.

The SNMP agent does not request menu information.

SNMP-Related Messages

The SNMP agent uses the following requests to query each card for its SNMP root node and its openGear-to-SNMP parameter mappings.

OGP_GET_SNMP_BASE Request

Message type: 0x56

Message length: 1

Response required: OGP_GET_SNMP_BASE Response

Description: This requests the SNMP base node of the card MIB as a UTF-8 string.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code

OGP_GET_SNMP_BASE Response

Message type: 0xD6

Message length: 2 + name length

Response required: none

Description: This is the required response to an OGP_GET_SNMP_BASE request. It provides the SNMP base node of the card MIB as an UTF-8 string. The base node should be expressed relative to the private.enterprises node of the SNMP tree (1.3.6.1.4.1). The base node would typically start with your company's enterprise number. For example, if the actual base node (OID) of the card MIB is "1.3.6.1.4.1.27399.1.1.7", then it should return "27399.1.1.7".

If the card does not support SNMP, it should set the return code to **OGP_UNSUPPORTED**.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK – normal return • OGP_UNSUPPORTED – SNMP not supported
lens	1	1	uint8	string length in bytes including null terminator
base	2	len	char[]	SNMP base oid as a null-terminated UTF-8 string

OGP_GET_SNMP_OID Request

Message type: 0x57

Message length: 3

Response required: OGP_GET_SNMP_OID Response

Description: This requests the SNMP OID for the parameter with the given openGear object ID.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
oid	1	2	uint16	openGear object ID of the requested parameter

OGP_GET_SNMP_OID Response

Message type: 0xD7

Message length: 4 + name length

Response required: none

Description: This is the required response to an OGP_GET_SNMP_OID request.

It provides the SNMP object ID for the requested parameter. The SNMP OID should be expressed relative to the base node of the card MIB. For example, if the SNMP OID for a parameter is “1.3.6.1.4.1.27399.1.1.7.1.3.1.3” and the base node is “1.3.6.1.4.1.27399.1.1.7”, then the OID returned should be “1.3.1.3”.

If the card does not support SNMP, it should set the return code to **OGP_UNSUPPORTED**.

If there is no SNMP object for the requested parameter, the return code should be set to **OGP_PARAM_NOTFOUND**.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK – normal return • OGP_UNSUPPORTED – SNMP not supported • OGP_PARAM_NOTFOUND – there is no SNMP object for this parameter
oid	1	2	uint16	openGear object ID of the requested parameter
len	3	1	uint8	string length in bytes including null terminator
snmpOid	4	len	char[]	SNMP OID as a null-terminated UTF-8 string

***Note**: len and snmpOid fields are sent only for return code OGP_OK.*

OGP_GET_SNMP_TRAP Request

Message type: 0x58

Message length: 3

Response required: OGP_GET_SNMP_TRAP Response

Description: This requests the SNMP OID for any trap (notification) associated with the requested parameter. The trap will be sent whenever the parameter value changes.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
oid	1	2	uint16	openGear object ID of the requested parameter

OGP_GET_SNMP_TRAP Response

Message type: 0xD8

Message length: 4 + name length

Response required: none

Description: This is the required response to an OGP_GET_SNMP_OID request.

It provides the SNMP trap (notification) object ID associated with the requested parameter. The trap should be expressed relative to the base node of the card MIB. For example, if the SNMP OID for the trap is “1.3.6.1.4.1.27399.1.1.7.0.2” and the base node is “1.3.6.1.4.1.27399.1.1.7”, then the trap OID returned should be “0.2”.

If the card does not support SNMP, it should set the return code to **OGP_UNSUPPORTED**.

If there is no SNMP trap for this parameter, the return code should be **OGP_PARAM_NOTFOUND**.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK – normal return • OGP_UNSUPPORTED – SNMP not supported • OGP_PARAM_NOTFOUND – there is no SNMP object for this parameter
oid	1	2	uint16	openGear object ID of the requested parameter
namlen	3	1	uint8	string length in bytes including null terminator
name	4	namlen	char[]	SNMP OID as a null-terminated UTF-8 string

***Note**: The namlen and name fields are sent only when return code is OGP_OK.*

OGP_GET_SNMP_TYPE_HINT Request

Message type: 0x5B

Message length: 3

Response required: OGP_GET_SNMP_TYPE_HINT Response

Description: This requests a hint to tell the SNMP agent which data type to report for the parameter with the given openGear object ID.

Field	Offset	Length	Format	Description
spare	0	1	uint8	placeholder for return code
oid	1	2	uint16	openGear object ID of the requested parameter

OGP_GET_SNMP_TYPE_HINT Response

Message type: 0xDB

Message length: 5

Response required: none

Description: This is the required response to an OGP_GET_SNMP_TYPE_HINT request.

It provides information about the SNMP data type to use. By default, the SNMP agent will determine the SNMP data type from the 'type' field in the OGP_GET_DESCRIPTOR response (message 0xC7). In some cases, the parameter may need to use an SNMP data type not defined in the openGear protocol. In such cases, this message will tell the SNMP agent which data type to report.

If the card does not support **OGP_SNMP_TYPE_HINT** message, it should set the return code to **OGP_UNSUPPORTED**.

If there is no SNMP type hint for this parameter, the return code should be **OGP_PARAM_NOTFOUND**.

Field	Offset	Length	Format	Description
rc	0	1	uint8	return code: • OGP_OK – normal return • OGP_UNSUPPORTED – SNMP not supported • OGP_PARAM_NOTFOUND – there is no SNMP support for this parameter
oid	1	2	uint16	openGear object ID of the requested parameter

Field	Offset	Length	Format	Description
version	3	1	uint8	This value should be set to ‘1’.
hint	4	1	uint8	• 0 — Use the default data type as implied by the openGear parameter data type • 1 — use the SNMP IpAddress data type (INT32 and INT32 Array parameters only)

***Note**: The SNMP agent ignores any information provided in the ‘widget’ field of the OGP_GET_DESCRIPTOR response.*

Help in DashBoard

In This Chapter

This chapter describes how to provide manuals for openGear and DashBoard Connect devices in the DashBoard control system.

The following topics are discussed:

DashBoard Help Overview

DashBoard provides a context-sensitive help system as well as a general table-of-contents-based help system. Manuals for openGear cards and DashBoard Connect devices can be installed into the help systems as add-on features available as separate downloads provided by openGear card manufacturers.

DashBoard Help Table of Contents

Manuals describing the DashBoard control system is installed by default. The contents of these default menus are listed in the table of contents of the main Help window. The Help window is launched when a user clicks the “Help Contents” option in the main DashBoard window’s “Help” menu.

The Help table of contents also shows a list of any openGear card manuals currently installed in the help system. These manuals are not included with the DashBoard installer and must be installed after the main DashBoard application. Information necessary for a user to download and install card manuals is provided by default under the section “Importing openGear Help Files”.

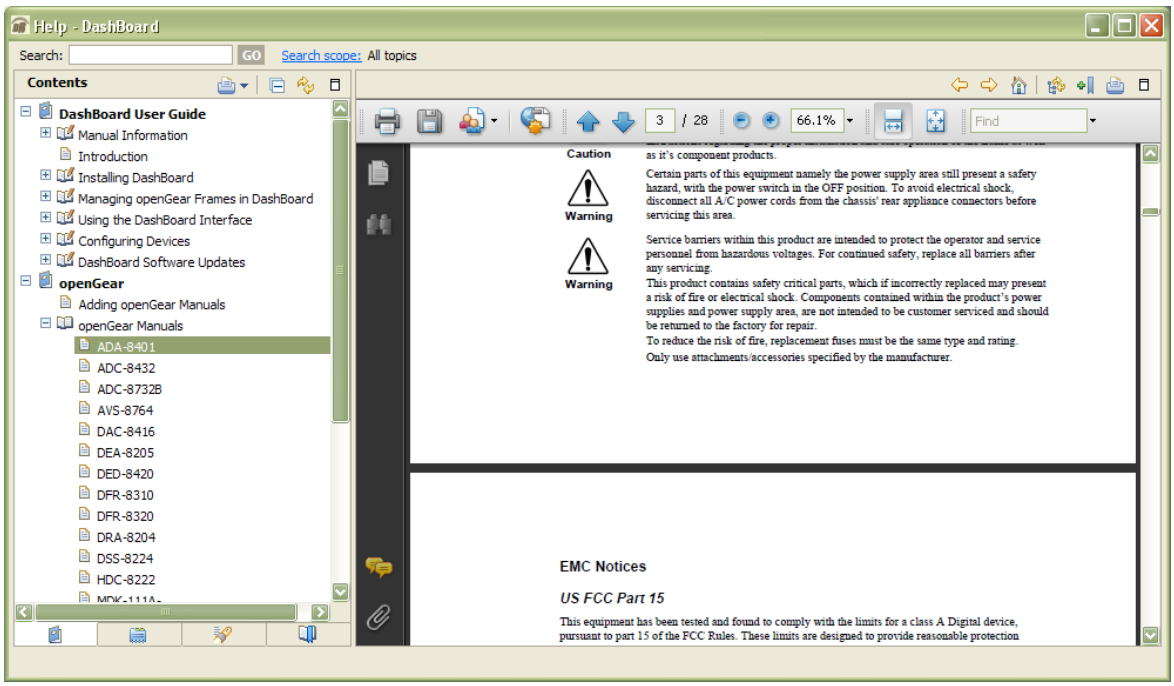


Figure 10.1 *- DashBoard Help Table of Contents*

Context-Sensitive Help

In addition to the table-of-contents-based help system, there is also a context sensitive help system to provide access to card manuals when a user presses the **F1** key while configuring a device. The context-sensitive help system shows as a docked window inside of the main DashBoard window.

When a card's manual is installed, a link to the manual is shown in the context-sensitive help view. If no manual can be located, a link to the "Adding openGear Help" information is shown. The context-sensitive help for a device can be shown when a user presses **F1** while a card's configuration page has the focus or when the DashBoard tree views have the focus.

If multiple devices are selected in a tree view, the links provided by all selected devices are shown.

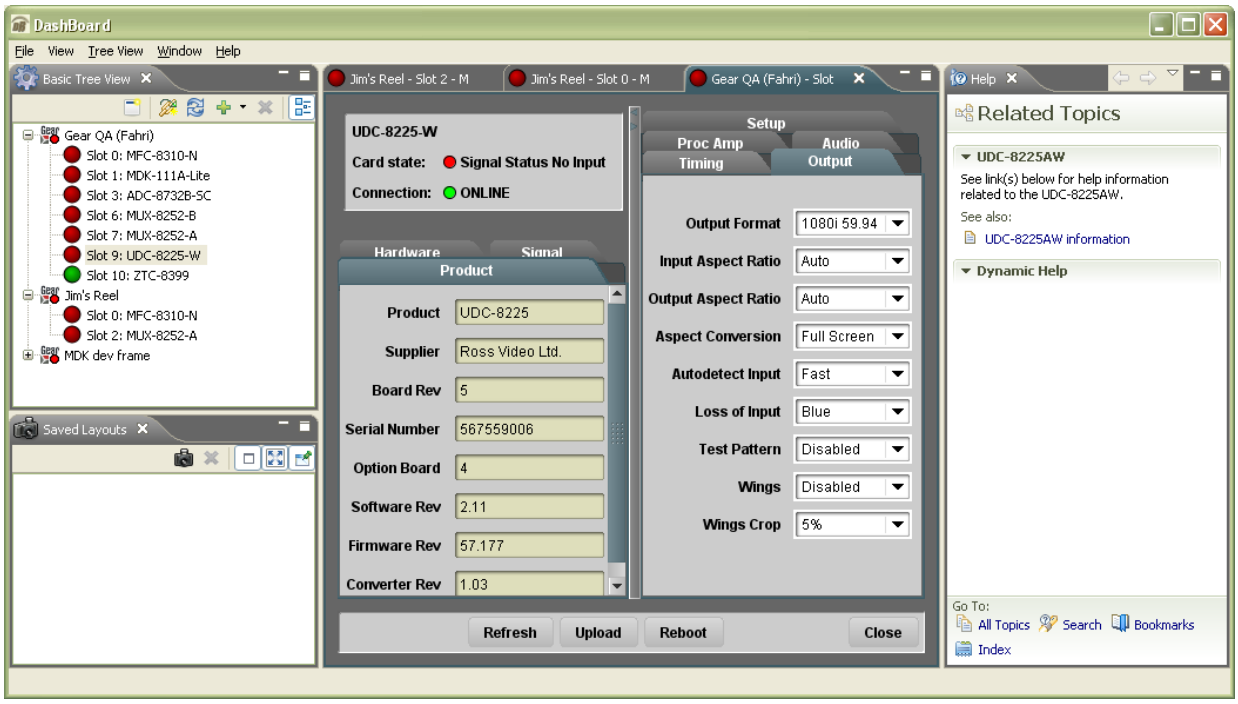


Figure 10.2 *- Context-Sensitive Help*

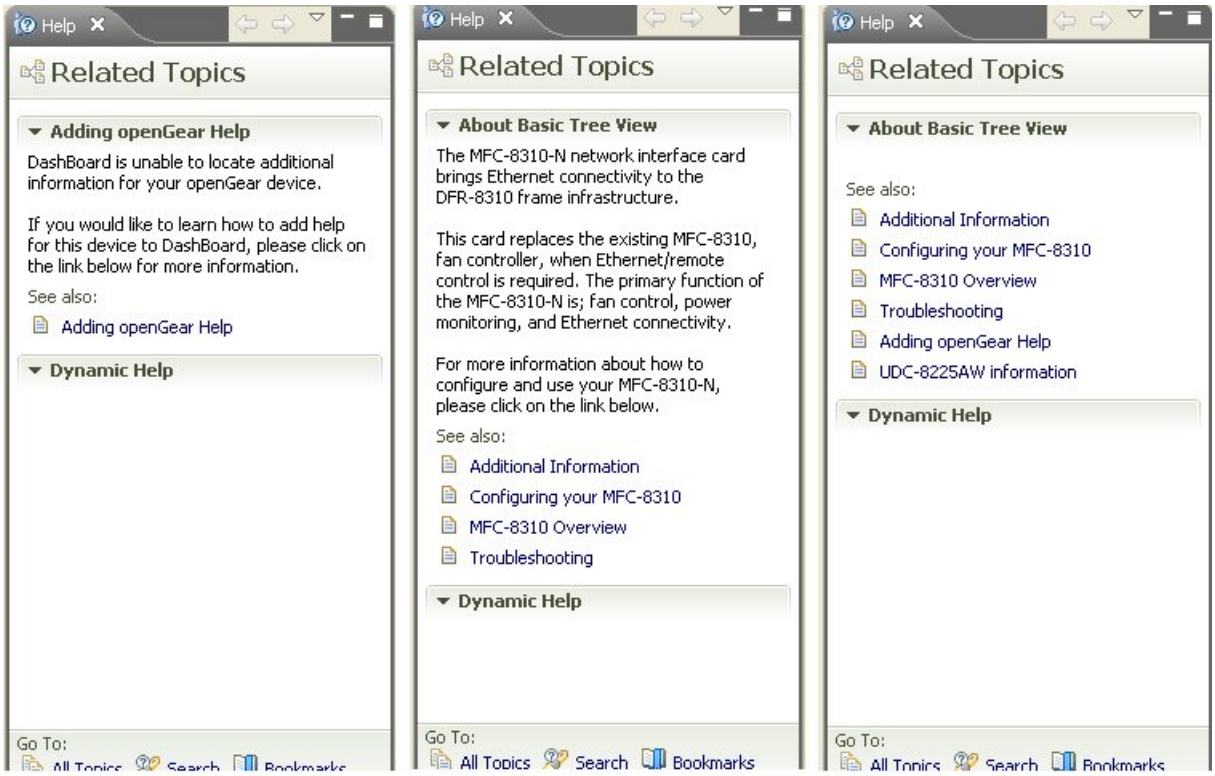


Figure 10.3 *- Context-Sensitive Help Examples (L-R: no manual, complex manual, multiple items selected in a tree)*

Acceptable File Types

Basic File Types

Basic manual files provide basic support for openGear manuals. Basic files are expected to be entirely self-contained where one simple file contains the entire manual for an openGear card (there are no external file dependencies). DashBoard allows the following basic file types to be used as manuals:

Type	Extension(s)	Description
HTML File	*.htm *.html	Simple HTML files can be included. In their simple form, the HTML files should not depend on any external files (such as style sheets or graphics). For html with additional resources, see the “Folder” advanced file type.
Adobe PDF	*.pdf	PDF files can be displayed in the help browser provided the end user has installed the Adobe Acrobat Reader plug-in for their system web browser.
Text Files	*.txt	This is a simple text file containing help information. Lines are wrapped. Carriage returns and spacing in the document are used.
Flash	*.swf	Simple Shockwave Flash files can be used as a manual. These flash files should not depend on any external files (such as graphics). For Flash with additional resources, see the “Folder” advanced file type.

Folder File Type

In addition to the simple file types, folders can also be used to contain more complex help information. In this case, the top-level help file for a manual is a folder instead of one of the basic file types. The contents of the folder can be anything.

By default, all simple files inside the root of the folder are listed as help topics. If an index file is provided, all other topics are hidden and the display will match the display of a basic file.

Required/Special Files

The following files have special significance in a folder:

File	Required	Description
.openGearHelp	Yes	This marks the folder as a valid openGear Help file.
_description.txt	No	Override the default text provided before the help links (e.g. “See link(s) below for help information related to the UDC-8225A-W.”) with the text provided in the description file.
index.*	No	A file named **index** with one of the basic file extensions (.htm, .html, .pdf, .txt, or .swf) will prevent additional files in the folder from being listed as topics. Instead, a single topic named after the card will point to this index file. There should only be one index file in a folder. If no index file is provided, each basic file in the root of the folder will be listed as a help topic.

Help Links

Often a single manual may be appropriate for multiple cards. To prevent the need for multiple copies of the same manual files, a help link file can tell DashBoard to use the help provided for a different product as the help for this product.

The help link file has the format *.helplink and contains a list of device identification information for the device that may provide the manual for this card. The device identification information should follow the naming convention defined below. A help link can provide multiple possible matches by putting one possible match on each line in the file. Help links that point to other help links are skipped.

Example:

If the ABC-9876, ABC-9877, ABC-9878, and ABC-9879 all share the same manual, you could create:

- ABC_9876.pdf
- ABC_9877.helplink
- ABC_9878.helplink
- ABC_9879.helplink

Where the *.helplink files all contain the following line:

```
ABC_9876
```

Zip Files

Zip files are not allowed as card manuals but may be used to distribute them. Zip files can be installed using the **Import openGear Help Files** wizard in DashBoard. A zip file can contain one or more manual files/folders to be extracted into the help directory. The zip must contain an empty marker file in its root called **openGearHelp** to mark it as a valid openGear manual package.

File Naming Conventions

DashBoard uses the product and main software parameter values provided by cards to look-up manual information (provided by parameters with OIDs 0x0105 and 0x010B). These two parameters define the product name and product version sections of the filename. If an exact match cannot be found, the closest match is returned.

Restricted Characters

The characters ".", "-", "(", ")", and "_" all have special meanings in openGear manual files and must be used carefully:

Character	Description
. (dot/period)	This is used to separate the rest of the filename from the file extension. Any "." characters in a card's product name or software version are replaced with "_" characters during the look-up. This character must appear directly before the file's extension.
- (dash)	Separates sections of file information. Any "-" characters in a card's product name or software version are replaced with "_" during the look-up.
_ (underscore)	Where the "-" separates sections, the "_" separates parts within a section. "." And "-" characters that may appear in the product name or software version are replaced with this character. For final display, "_" characters in the product name section are converted back to "-" characters.

Filename Structure

Manual files have the following structure:

```
[product name]-[software version]-[locale (language and country)]-[revision
string]-([display name]).[file extension].
```

The product name is required and if a display name is provided, it must be the final element and surrounded by parenthesis; other fields are optional.

Example: `ABC_1234-1_10-en_CA-Issue 01-(ABC Broadcast Card).pdf` provides a Canadian English manual the manual file for ABC-1234 running software version 1.10 and displays as "ABC Broadcast Card" in the table of contents.

Matching Order

The matching order is as follows:

Product Name > Language > SoftwareVersion > Country > Revision String

File	Required	Description
product name	Yes	This must match the product name contained in parameter 0x0105. All "." and "-" characters in the product name must be replaced with "_" characters.
software version	No	This will match against the value provided by parameter 0x010B. All "." and "-" characters in the version number must be replaced with "_" characters. If this field is not provided, it is assumed to be version 0. If there is no exact match for the version number, a partial match will be used where DashBoard attempts to match major/minor version numbers or the most recent version of the manual provided.
language	No	This is matched against language information provided from the machine on which DashBoard is running. Matching can occur on both the language code and the country code. An exact match will override a partial match unless the version information provides a better match. See the matching order, listed immediately before this table.

File	Required	Description
revision string	No	If the same manual has had multiple revisions, the String that occurs last in the alphabet is returned. For example, B > A > 9.
display name	No	This will override the name of the help topic shown in DashBoard. It must always be surrounded by parenthesis and may contain other restricted characters such as "-", "_", and ".". Unlike the language and revision string fields, this field may be used even if the fields preceding it are not provided. The display name is always the final element before the file extension and is always surrounded by parenthesis. For example, `ABC_1234-(Name for ABC Card).htm` is acceptable, but **ABC_1234-en_US.htm** is not.
file extension	Yes (Unless the file is a folder)	Must be one of ".htm", ".html", ".pdf", ".swf", or ".helplink".

Help File Packaging

Packaging Types

Help files can be distributed as individual help files, zipped packages of help files, or as help packs using a stand-alone installer.

Individual help files require no special packaging and simply must conform to the naming convention specified above. Zipped packages may contain multiple help files/folders and require an empty marker file named ".openGearHelp" to indicate that it is a valid openGear help file pack.

The help pack installer and zip files can be created using the Help Pack Generator application described in this section.

Help Pack Generator

The Help Pack Generator can be purchased from Ross Video as part of the openGear Software Development Kit. Please contact Ross Video for more information about how to obtain this feature.

Installing the Generator

From the openGear Development Kit CD, run the HelpPackGenerator/Help Pack Generator-setup.exe file to install the help pack generator in the location of your choice. All help packs will be output in the install location and all help files must be in a folder under the "help" directory in the install location.

Note *The Help Pack Generator requires Java version 6 or higher to be installed.*

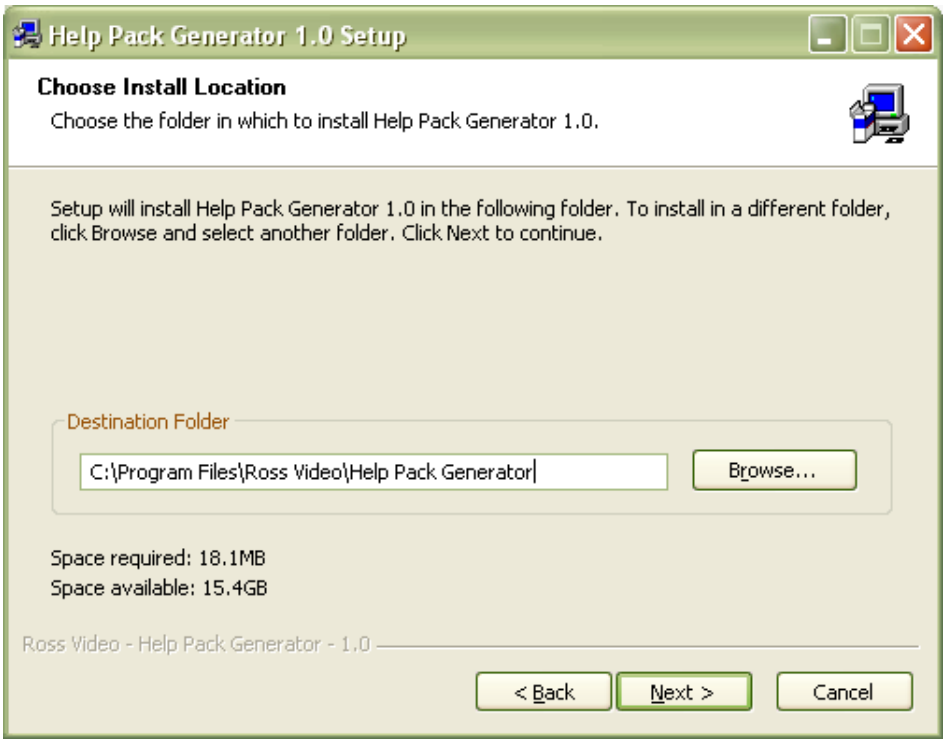


Figure 10.4 *- Installer for the Help Pack Generator*

For the example shown in **Figure 10.4**, the help packs will be placed in c:\Program Files\Ross Video\Help Pack Generator root directory. The source files for the help should be located under c:\Program Files\Ross Video\Help Pack Generator\[pack name]*

Using the Generator

Launch the Help Pack Generator by clicking on the shortcut created by the installer. When the application starts, the **Help Pack Compiler** dialog box appears, as shown in **Figure 10.5**:

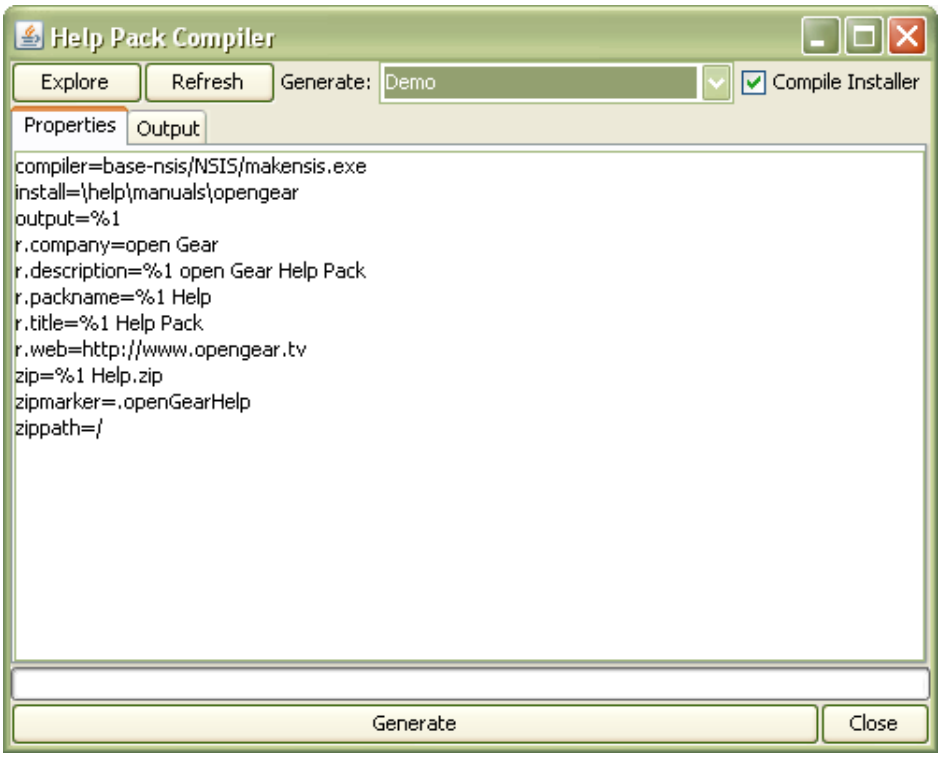


***Figure 10.5** - Help Pack Compiler Dialog Box*

To create a help pack:

1. Click the **Explore** button to navigate to the help pack source folder.
2. Create and name a new folder for your help.
3. Copy manual files to the new folder.
4. In the **Help Pack Compiler** dialog box, click the **Refresh** button.
5. Select the help pack folder from the combo box.
6. On the **Properties** tab, edit any properties that must be modified.

   Tip: Modifications are automatically saved for later use.

   The properties are as follows:

Property	Default	Description
compiler	base-nsis/NSIS/makensis.exe	This is the path to the NSIS compiler used to generate the help pack. A copy is installed in the default location with the help pack generator.
install	help\manuals\opengear	The path relative to the DashBoard install root where the files will be extracted. The default is the openGear manual directory.
output	%1	The name to use as the prefix for the .nsi file generated. The default uses the name of the pack folder.
r.company	open Gear	The name of the company producing the help pack.
r.description	%1 open Gear Help Pack	The description that appears in the installer. The default is “[help pack folder name] openGear Help Pack”.

Property	Default	Description
r.packname	%1 Help	This is used as the prefix for the generated installer application. The pack name is also used as the registry key for the uninstaller in Windows.
r.version	not included	If this property is not defined, a date stamp is included in the generated installer. If the property is defined, the value is expected to be the "version" of this help pack.
r.title	%1 Help Pack	The title to use in the installer window and the name for the uninstaller shortcut.
r.web	http://www.opengear.tv	The website of the help pack manufacturer. By default, the openGear website is used.
zip	%1 Help.zip	The filename to use for the generated help pack zip file. If no value is specified, zip generation will be skipped.
zipmarker	.openGearHelp	Indicates that a marker file to validate the zip file is necessary and generates the provided file (by default, this is the ".openGearHelp" marker file).
zippath	/	The path prefix for files in the generated zip file.

Note: The String "%1" is automatically replaced with the name of the selected folder shown in the combo box wherever it occurs.

7. Click the **Generate** button to begin compiling.

   Three files are generated: **[Help Pack Name].nsi**, **[Help Pack Name]-setup.exe**, and **[Help Pack Name].zip**. You may delete the **.nsi** file. The **.exe** file is a standalone installer for the help pack. The **.zip** file is a compressed version of the help pack that conforms to the format used by the **DashBoard Import openGear Help Files** wizard.

   The **Output** tab of the **Help Pack Compiler** dialog box displays details about the generated files, as shown in **Figure 10.6**:

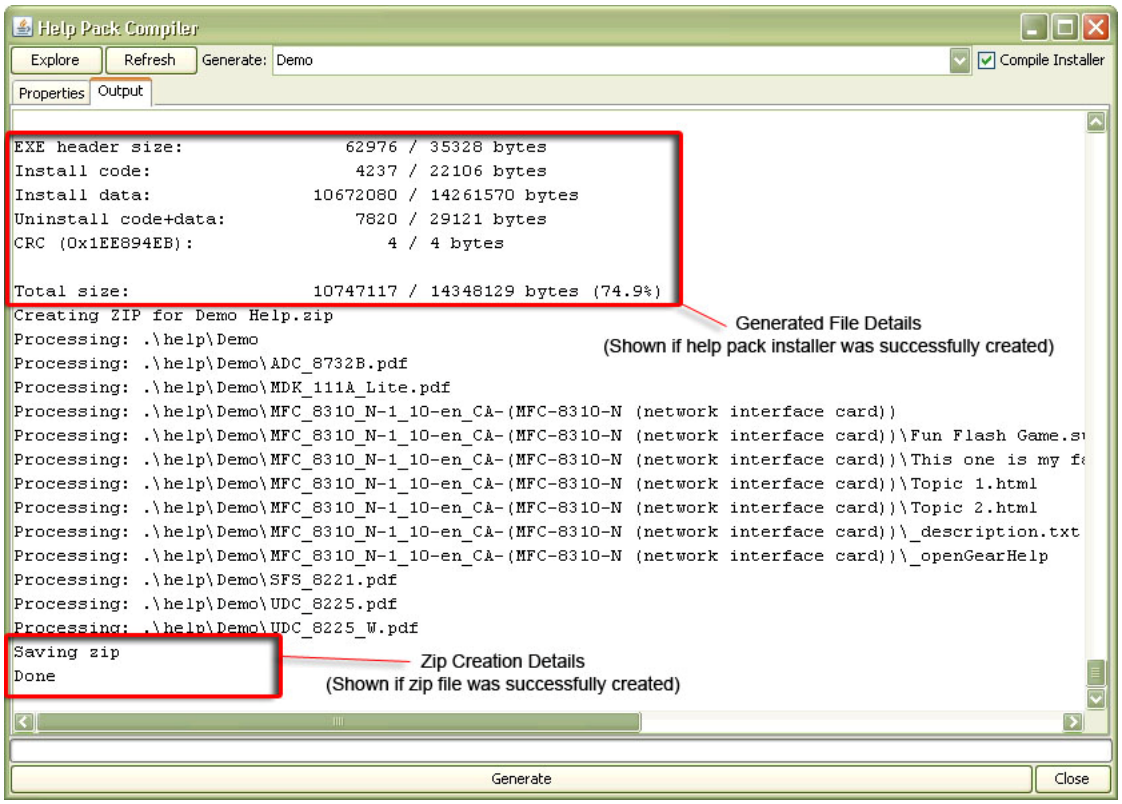


Figure 10.6 *- Application Output Text*

8. Click Close.

The **Help Pack Generator** window appears.

Note that the files created by the Help Pack Generator are available in the output folder you specified, as shown in **Figure 10.7**:

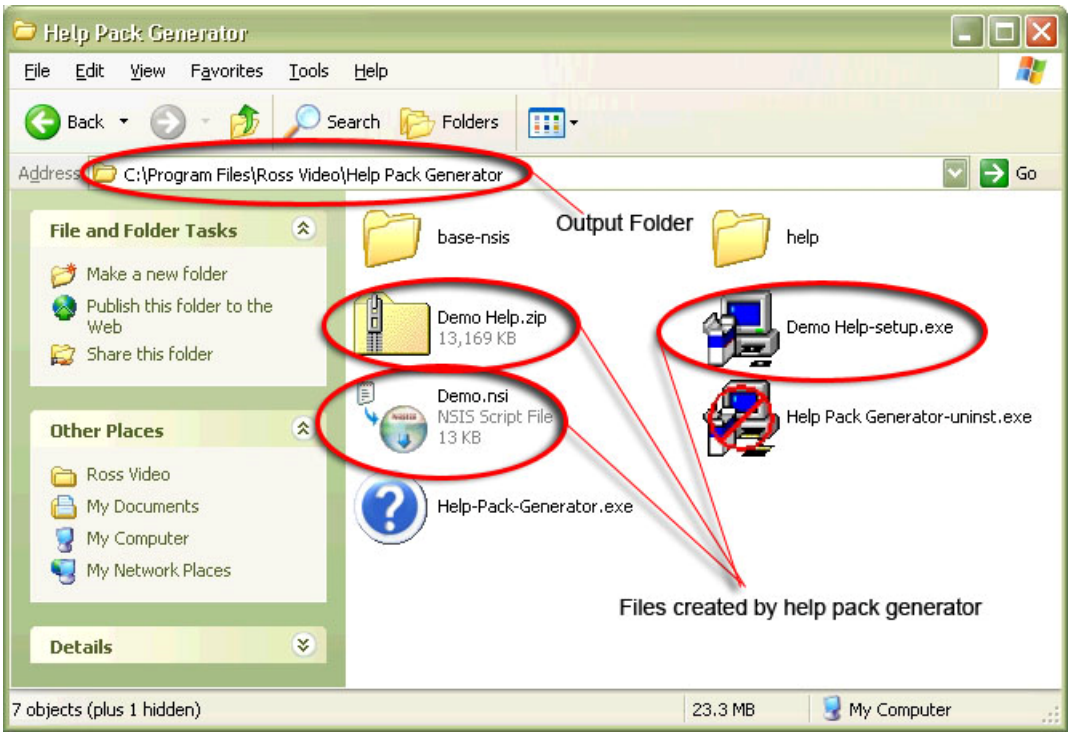


Figure 10.7 *- Output Folder Contents, including Files Generated by the Help Pack Generator*

End-User Installation

Help Pack Installer

To install a help pack from one of the generated installers, the end user should close DashBoard and run the [Pack Name]-setup.exe file and follow the wizard. The DashBoard installation path should be automatically detected by the installer.

If DashBoard is running, the new files will not appear in the main help table of contents until DashBoard is restarted.

Import openGear Help Wizard

The import wizard is launched in DashBoard when a user clicks on **File** > **New** > **Other** and selects **Import openGear Help Files** from the list of choices. The wizard allows the user to install basic help files, folders, or zipped help packs. For more information, see the ***DashBoard User Guide (8351DR-004-xx)***.

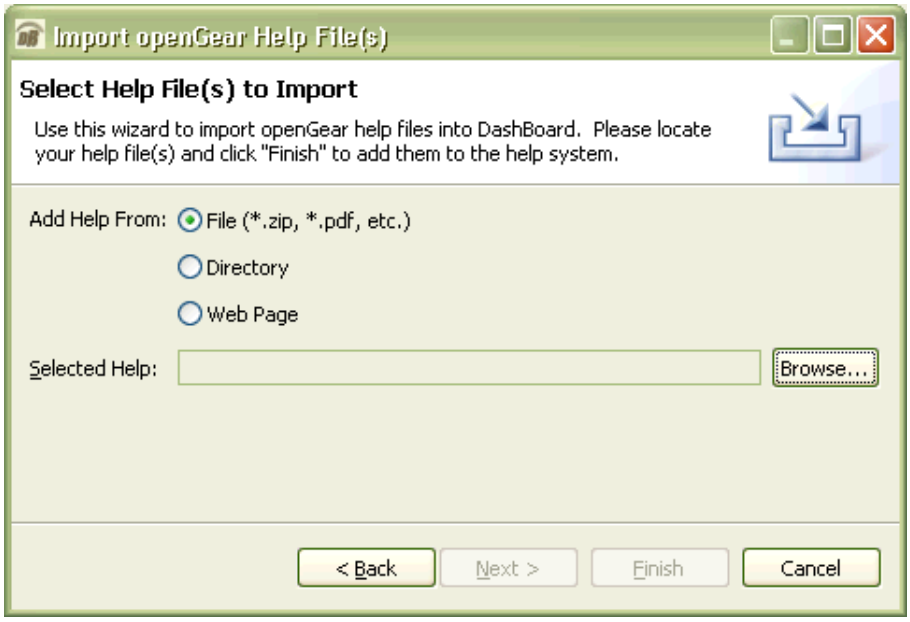


Figure 10.8 *- The Import openGear Help Wizard*

Appendices

This chapter includes the following appendices:

Appendix A: Message and Return Code Definitions

```
/*****************************************************************************
 * FILENAME:
 *    common/ogpdef.h
 *
 * PROJECT:
 *    ZeusAVR common
 *
 * FILE DESCRIPTION:
 *    Message type definitions for the openGear communications protocol (OGP).
 *
 * REVISION HISTORY:
 *    Glenn Greig (22 January 2008)
 *        Adapted from canmsg.h
 *****************************************************************************/

#ifndef __OGPDEF_H
#define __OGPDEF_H

// maximum length of OGP messages
#define  OGP_MTU         260

// mask limiting OGP addresses to 6 bits
#define  OGP_ADDRESS_MASK    0x3f

// well-defined OGP addresses
#define  OGP_ADDR_BCAST    0x3F  // OGP broadcast address
#define  OGP_ADDR_UPLOAD   0x3E  // OGP multicast address for file upload
#define  OGP_ADDR_NONE     0x00  // Invalid OGP address for cards
#define  OGP_ADDR_FINC     0x00  // OGP address for the FINC
#define  OGP_ADDR_PRINT    0x01  // OGP address for debug messages
#define  OGP_ADDR_TRAP     0x02  // OGP address for inbound messages (e.g. traps)

// this is used to map a request to a response
#define  OGP_RESPONSE           0x80

// messages broadcast to openGear devices
#define  OGP_TIME               0x01  // UTC time broadcast

// messages broadcast by openGear devices
#define  OGP_PRINT              0x00  // debug / print message
#define  OGP_DEBUG              0x00
#define  OGP_REPORT_PARAM       0x10  // report the value of a param
#define  OGP_TRAP               0x11  // report a status event
#define  OGP_EVENT_LOG          0x12  // report a log message - not implemented
```

```
#define  OGP_BOOTLOAD           0x13  // boot loader is running
#define  OGP_REPORT_STRING 0x14 // message for stress-test - do not use

// messages related to file upload (software upgrade)
#define  OGP_START_UPLOAD       0x40  // start the upload process
#define  OGP_UPLOAD_PAGE        0x41  // send a single page of code
#define  OGP_VERIFY_UPLOAD      0x42  // verify the upload was successful
#define  OGP_REBOOT             0x43  // reboot the processor

// command-line interface
#define  OGP_COMMAND            0x44  // UTF-8 command

// messages to exchange parameter info
#define  OGP_GET_NUMPARAMS      0x45  // ask how many params in table
#define  OGP_GET_PARAM_OIDS     0x46  // request the object IDs
#define  OGP_GET_DESCRIPTOR     0x47  // request the descriptor for a param
#define  OGP_GET_PARAM_NAME     0x48  // request the parameter name
#define  OGP_GET_PARAM          0x49  // request the value of a param
#define  OGP_SET_PARAM          0x4A  // set the value of a param (absolute)
#define  OGP_SET_RELATIVE       0x4B  // set the value of a param (obsolete)
#define  OGP_GET_ARRAY_ELEMENT  0x4C  // get the value of an array element
#define  OGP_SET_ARRAY_ELEMENT  0x4D  // set the value of an array element
// spare 0x4E and 0x4F

// messages related to the menu structure
#define  OGP_GET_MENUSET_NAME   0x50  // get the name of a menu group
#define  OGP_GET_MENU_COUNT     0x51  // get the number of menus in a group
#define  OGP_GET_MENU_NAME      0x52  // get the name of a menu
#define  OGP_GET_MENU_OIDS      0x53  // get the object IDs for a menu

// messages to request product information
#define  OGP_GET_PRODUCT_INFO   0x54  // get the product info (internal use only)
#define  OGP_GET_PRODUCT_KEY    0x55  // get the product key (internal use only)

// messages to request SNMP mappings
#define  OGP_GET_SNMP_BASE      0x56  // get the SNMP base OID
#define  OGP_GET_SNMP_OID       0x57  // get the SNMP OID for a param
#define  OGP_GET_SNMP_TRAP      0x58  // get the SNMP TRAP for a param

// messages to request external objects
#define  OGP_GET_EXTERNAL_OBJ   0x59  // get the object with specified OID

// message to request menu state
#define  OGP_GET_MENU_STATE     0x5A  // get the visibility of a menu

// spare 0x5B to 0x7F

// generic error codes
#define  OGP_OK                  0x00  // no error
#define  OGP_UNSUPPORTED         0x01  // unsupported OGP message
#define  OGP_INVALID_LENGTH      0x02  // message is wrong length
#define  OGP_REQUEST_DENIED      0x03  // request denied (provide reason)
#define  OGP_NOTFOUND            0x11  // parameter or menu not found
#define  OGP_NO_RESP             0xFF  // no response received

// error codes for parameter messages (should not be here)
#define  OGP_PARAM_NOTFOUND      0x11  // parameter not found
#define  OGP_PARAM_BADVAL        0x12  // parameter value out of range
#define  OGP_PARAM_READONLY      0x13  // parameter is read only
```

```
#define  OGP_PARAM_LOCKED        0x14  // parameter is read only
#define  OGP_PARAM_BADINDEX      0x15  // array index out of bounds

// error codes for file upload (should not be here)
#define  OGP_INVALID_PRODUCT     0x11  // wrong product name for upload
#define  OGP_INVALID_UPLOAD      0x12  // unsupported file type
#define  OGP_INVALID_STATE       0x13  // wrong message for current state
#define  OGP_UPLOAD_WAIT         0x14  // try again after N milliseconds
#define  OGP_INVALID_PAGE        0x15  // data out of range (past length)
#define  OGP_INVALID_CRC         0x16  // CRC mismatch
#define  OGP_INVALID_FILESIZE    0x17  // upload file size is wrong size
#define  OGP_INVALID_KEY         0x18  // product key mismatch

// well-defined trap messages (to be used with OGP_TRAP
#define  OGP_RESTART             0x01  // product has just booted up
#define  OGP_PARAM_CHANGED       0x02  // a parameter descriptor has changed
#define  OGP_MENU_CHANGED        0x03  // a menu descriptor has changed
#define  OGP_EXTOBJ_CHANGED      0x04  // an external object has changed

// macros defining the allowable content for OGP_PARAM_CHANGED trap
#define  OGP_PARAM_ACCESS_READONLY  0  // access changed to read only
#define  OGP_PARAM_ACCESS_READWRITE 1  // access changed to read/write
#define  OGP_PARAM_DESC_CHANGED     2  // descriptor changed

// macros defining the allowable content for OGP_MENU_CHANGED trap
#define  OGP_MENU_HIDDEN            0  // menu is hidden
#define  OGP_MENU_READONLY          1  // menu is visible, but disabled
#define  OGP_MENU_NORMAL            2  // menu is visible and enabled
#define  OGP_MENU_DESC_CHANGED      3  // descriptor changed

// length of fixed-length messages (should not be here)
#define  OGP_START_UPLOAD_LEN      40

#endif
```

Appendix B: Parameter Type Definitions

```
// parameter types supported
#define INT8U_PARAM_TYPE      1  // not implemented
#define INT16_TYPE      2
#define INT16U_PARAM_TYPE     3  // not implemented
#define INT32_TYPE      4
#define INT32U_PARAM_TYPE     5  // not implemented
#define FLOAT32_TYPE      6
#define STRING_TYPE     7
#define BOOLEAN_PARAM_TYPE    8  // not implemented
#define COMMAND_PARAM_TYPE    9  // not implemented
#define ALARM_PARAM_TYPE     10  // not implemented

#define INT16_ARRAY_TYPE     12  // array of 16-bit integers
#define INT32_ARRAY_TYPE     14  // array of 32-bit integers
#define FLOAT32_ARRAY_TYPE     16  // array of 32-bit floats
#define STRING_ARRAY_TYPE    17  // array of strings
#define BINARY_DATA_TYPE     18  // binary data

// length for each parameter type
#define INT8U_PARAM_SIZE      1
#define INT16_SIZE      2
```

```
#define INT16U_PARAM_SIZE     2
#define INT32_SIZE      4
#define INT32U_PARAM_SIZE     4
#define FLOAT32_SIZE      4
#define STRING_SIZE     1  // use the maximum permitted string length
#define BOOLEAN_PARAM_SIZE    1
#define COMMAND_PARAM_SIZE    1
#define ALARM_PARAM_SIZE      2

#define NULL_LENGTH           0  // length of a NULL object

// settings for the access flag
#define ACCESS_READWRITE   0x01
#define ACCESS_READONLY    0x00

// precision is used for numerical displays
#define PRECISION_PCT      0x80  // display as percent - not implemented
#define PRECISION_NONE     0x00
#define PRECISION_0        0x00  // display whole numbers only
#define PRECISION_1        0x01  // display with one digit after the decimal
#define PRECISION_2        0x02
#define PRECISION_3        0x03

// constraint types
#define NULL_CONSTRAINT       0  // unconstrained
#define RANGE_CONSTRAINT      1  // constrained by min and max
#define CHOICE_CONSTRAINT     2  // choice of several values (max 255)
#define EXTENDED_CHOICE       3  // choice of several values (more than 255)
#define STRING_CHOICE         4  // choice of several strings
#define RANGE_STEP_CONSTRAINT 5  // constraint by min/max with defined step size
#define ALARM_TABLE          10  // alarm encoded as a bit field
#define EXTERNAL_CONSTRAINT  11  // reference to constraint table

// external object types
#define CONSTRAINT_OBJECT     1  // external object containing a constraint
```

Appendix C: Widget Hint Definitions

```
// widget hints for all parameter types
#define WIDGET_DEFAULT                  0  // let DashBoard decide
#define WIDGET_TEXT_DISPLAY             1  // display as text, read only
#define WIDGET_HIDDEN                   2  // do not display

// widget hints for numeric types with NULL_CONSTRAINT or RANGE_CONSTRAINT
#define WIDGET_SLIDER_HORIZONTAL        3  // slider (RANGE only)
#define WIDGET_SLIDER_VERTICAL          4  // slider (RANGE only)
#define WIDGET_SPINNER                  5  // spinner
#define WIDGET_TEXTBOX                  6  // numeric entry field
#define WIDGET_PROGRESS_BAR            17  // progress bar (RANGE only)
#define WIDGET_AUDIO_METER             19  // audio meter (RANGE only)
#define WIDGET_MENU_POPUP              20  // popup menu with the ID(INT ONLY)
#define WIDGET_TIMER                   21  // countdown/up timer (RANGE only)
#define WIDGET_SLIDER_H_NO_LABEL       24  // unlabeled slider (RANGE only)
#define WIDGET_SLIDER_V_NO_LABEL       25  // unlabeled slider (RANGE only)
#define WIDGET_VERTICAL_FADER          26  // vertical fader bar (RANGE only)
#define WIDGET_TOUCH_WHEEL             27  // touch wheel (RANGE only)
#define WIDGET_HEX_SPINNER             28  // base 16 spinner (RANGE only)
```

```
#define WIDGET_ABSOLUTE_POSITIONER     29  // absolute x,y positioner
#define WIDGET_CROSSHAIR          30  // joystick-like x,y positioner
#define WIDGET_JOY_STICK          34  // joystick x,y positioner

// widget hints for integer types with CHOICE_CONSTRAINT
#define WIDGET_COMBO_BOX                7  // combo box - usually the default
#define WIDGET_CHECKBOX                 8  // two choices
#define WIDGET_RADIO_HORIZONTAL         9  // radio buttons
#define WIDGET_RADIO_VERTICAL          10  // radio buttons
#define WIDGET_BUTTON_PROMPT           11  // single choice
#define WIDGET_BUTTON_NO_PROMPT        12  // single choice
#define WIDGET_BUTTON_TOGGLE           13  // two choices
#define WIDGET_FILE_DOWNLOAD           18  // external object OID/filename pairs
#define WIDGET_RADIO_TOGGLE_BUTTONS    22  // display a toggle button for choices
#define WIDGET_TREE                    31  // display a tree with choices
#define WIDGET_TREE_POPUP              32  // display a tree in a combo box

// widget hints for INT32
#define WIDGET_IP_ADDRESS              14  // nnn.nnn.nnn.nnn
#define WIDGET_COLOR_CHOOSER           23  // argb color chooser
#define WIDGET_COLOR_CHOOSER_POPUP     33  // argb color chooser in popup

// widget hints for integer arrays
#define WIDGET_ARRAY_HEADER_VERTICAL   15  // array layout specification
#define WIDGET_ARRAY_HEADER_HORIZONTAL 16  // array layout specification

// widget hints for STRING
#define WIDGET_TEXT_ENTRY               3  // normal text entry field
#define WIDGET_PASSWORD                 4  // uses password entry field
#define WIDGET_TITLE_LINE               5  // layout hint - read only
#define WIDGET_LINE_ONLY                6  // layout hint - read only
#define WIDGET_TITLE_ONLY               7  // layout hint - read only
#define WIDGET_PAGE_TAB                 8  // layout hint - read only
#define WIDGET_LICENSE                  9  // RossKeys license adapter
#define WIDGET_TITLE_HEADER            10  // layout hint - read only
#define WIDGET_COMBO_ENTRY             11  // combo box plus entry field
#define WIDGET_ICON_DISPLAY            12  // icon plus text display
#define WIDGET_RICH_LABEL              13  // multi-line display (html format)
#define WIDGET_MULTILINE_TEXT_ENTRY    14  // multi-line text entry (non-html)

// widget hints for STRING (used with special OID 255.1)
#define WIDGET_NAME_OVERRIDE_APPEND     0
#define WIDGET_NAME_OVERRIDE_REPLACE    1

// deprecated names - here for backward compatibility
#define WIDGET_NONE        WIDGET_DEFAULT
#define WIDGET_COMBO       WIDGET_COMBO_BOX
#define WIDGET_RADIO       WIDGET_RADIO_HORIZONTAL
#define WIDGET_HSLIDER     WIDGET_SLIDER_HORIZONTAL
#define WIDGET_VSLIDER     WIDGET_SLIDER_VERTICAL
```

Appendix D: Reserved Object IDs

Reserved OIDs

All parameter OIDs in the set 0xFF00 to 0xFFFF are reserved for current and future protocol messages. Apart from these, there are several other OIDs that have special significance in DashBoard.

The following table lists the reserved OIDs.

OID	Name	Type	Constraint	Function
0x0102	SUPPLIER_NAME	String	N/A	Name of the card manufacturer or OEM supplier (i.e. who customer should call for support). Reported as a generic card parameter by SNMP. Required for SNMP.
0x0105	PRODUCT_NAME	String (32-bytes max)	N/A	The product name used to identify the card in DashBoard. This name should not change. For display purposes, an alternate name can be provided via OID 0xFF01. Reported as a generic card parameter by SNMP. Required by DashBoard and SNMP.
0x0106	SERIAL_NUMBER	String	N/A	Unique serial number.
0x010B	SOFTWARE_REV	String (20-bytes max)	N/A	This value is used by a card to report information about its software load. The value should be meaningful to the people supporting the card. Reported as a generic card parameter by SNMP. Required for SNMP. Version numbers are important for software uploads and DataSafe. For more information, see "Version Number Encoding" on page 7–4.
0x010C	FPGA_REV	String	N/A	This value is used by DashBoard compare software versions when uploading the Main Board FPGA Type (upload type 1). Version numbers are important for software uploads and DataSafe. For more information, see "Version Number Encoding" on page 7–4.
0x010D	OPTION_SOFTWARE_REV	String	N/A	This value is used by DashBoard compare software versions when uploading the Option Board Software Type (upload type 2). Version numbers are important for software uploads and DataSafe. For more information, see "Version Number Encoding" on page 7–4.
0x010E	OPTION_FPGA_REV	String	N/A	This value is used by DashBoard compare software versions when uploading the Option Board FPGA Type (upload type 3). Version numbers are important for software uploads and DataSafe. For more information, see "Version Number Encoding" on page 7–4.
0x0201	SMPTE_STATUS	Int16	N/A	Card status to be reported via frame fault LEDs: • Value of 0 indicates no error. • Non-zero values indicate error state. **Note**: The use of SMPTE_STATUS is optional, but to avoid misinterpretation, OID 0x0201 should not be used for other parameters.

OID	Name	Type	Constraint	Function
0x0205	CURRENT_MILLIS	Int16	N/A	Current consumption in milliamps (mA) at 12 V. This may be used by the fan controller to adjust fan speed for high-current cards. **Note**: The use of CURRENT_MILLIS is optional, but to avoid misinterpretation, OID 0x0205 should not be used for other parameters
0x0601	EDIT_PERMISSION	Int16	Choice	Tells DashBoard that the card is editable. If this OID is used, parameters on the card will be editable only if the parameter value is 0. If the parameter value is non-zero, the card will display as read-only. **Note**: The use of EDIT_PERMISSION is optional, but to avoid misinterpretation, OID 0x0601 should not be used for other parameters
0xFE0F	FRAME_POWER_CAPABILITY	Int16	N/A	This OID is broadcast regularly to every card in the frame. The value of the parameter is the power available to each slot a card occupies. This value is calculated using the power rating of the power supplies installed in the frame (if the power supplies are different, the lowest rating is used), minus some overhead for the frame and frame controller card, divided by the number of slots in the frame. [(Power supply rating - overhead) / Number of slots in frame.] A card may consume the power of multiple slots, if the card occupies multiple slots. For example, a card occupying two slots may use two times the parameter value.
0xFF01	NAME_OVERRIDE	String	N/A	With a widget hint of 0, the value in this String will be appended to the device name (0x0105) when displayed in the DashBoard tree and tabs. With a widget hint of 1, the value in this String will be displayed instead of the value in 0x0105 in the DashBoard tree and tabs.
0xFF03	CONNECT_VERIFY	Mixed	N/A	This parameter is used for communicating DashBoard's connection handshake and response. For more details, see "Initiating Connection to OGP Devices" on page 5–1, and "Connection Handshake" on page 6–22.
0xFF02	UPLOAD_URL	String	N/A	Alternate file upload target. This overrides the behavior of the DashBoard upload button. • If this value is "disable", DashBoard will disable the upload button on the device page. • If this value is a valid URL, DashBoard will upload files to this location via HTTP POST.
0xFF04	FRAME_ID	String	N/A	Reserved for use by an openGear frame's Network Interface Card. If this parameter is provided, its value MUST match the unique ID provided by SLP and manual SLP attribute queries. If it does not, DashBoard will close its connection to the frame.
0xFF05	BACKWARDS_COMPATIBLE	String (20-bytes max)	N/A	Specifies the lowest software version to maintain OID-compatibility with this software version. If this OID is not supplied, the lowest software version is assumed to be the version specified in the SOFTWARE_REV OID (0x010B). The card guarantees that all software versions bounded by the version numbers specified between 0xFF05 and 0x010B can be restored using the same stored set of parameter values.

OID	Name	Type	Constraint	Function
0xFF06	RESTORE_SET_DELAY	Int16	N/A	Specifies the delay to use between each parameter set request during a card restore. The restore set messages will not be sent any faster than the specified delay. This number must be between 0 and 1000 milliseconds. If this value is not specified, a default of 0 is used. Parameters will be restored as quickly as the card can process the PARAM_SET commands. If the value is -1, DataSafe is disabled for this card. Other negative values are not valid and should not be used.
0xFF07	RESTORE_START	Int16	N/A	A parameter set request with a value of 1 will be sent to this parameter before the card data is restored (the equivalent of a button press in DashBoard). If this parameter is provided, its position in the list of OIDs returned by the OGP_GET_PARAM_OIDS response defines where the range of saved parameter values should start. No parameters whose OID was returned before this OID will be restored by DataSafe.
0xFF08	RESTORE_STOP	Int16	N/A	A parameter set request with a value of 1 will be sent to this parameter after the card data is restore is complete (the equivalent of a button press in DashBoard). If this parameter is provided, its position in the list of OIDs returned by the OGP_GET_PARAM_OIDS Response defines where the range of saved parameter values should stop. No parameters whose OID was returned after this OID will be restored by DataSafe.
0xFF09	DATASAFE_NAME	String	N/A	Alternative card name for determining DataSafe compatibility. For more details, see “DataSafe” on page 8–1.
0xFF0A	UPLOAD_NAME	Int16	Choice	Alternative card name for file upload purposes. For more details, see “DataSafe” on page 8–1.
0xFF0B	DISPLAY_OPTIONS	Int16 ARRAY	n/a	Each array element is used to define a different display option. Element 0 controls display of the card: • 0 (Default) = Display the card in the tree view • 1 = Hide the card in the tree view Element 1 controls the display of the slot name before the card name: • 0 (Default) = Display the slot name (e.g. Slot 1: UDC-8225-W) • 1 = Hide the slot name (e.g. UDC-8225-W) All other array elements are reserved for future use.
0xFF0C	DEVICE_ICON	Int16	N/A	Contains an external object ID for an encapsulated icon. For more information, see “node-image-url” on page 2–19.
0xFF0D	DEVICE_INDEX_URL	String	N/A	URL for a DashBoard Connect XML Definition. For Ultritouch see, “0xFFFD” on page 11–10.
0xFF0E	OGLML_DESCRIPTOR	String	N/A	Provides an OGLML URL that describes a layout to use in place of the standard configuration screen in DashBoard. For Ultritouch see, “0xFFFE” on page 11–11.

OID	Name	Type	Constraint	Function
0xFF0F	DEDICATED_CONNECTION	Binary	N/A	Allows a card that has its own Ethernet port to communicate directly with DashBoard, bypassing the CAN bus and MFC card. This allows traffic offloading from the CAN bus, and also allows messages to be sent to specific DashBoards rather than all of them. When connected, DashBoard will use this connection to send all messages to the card. DashBoard will continue to receive updates from both the dedicated OGP connection and the CAN Bus connection. UTF-8 String for the hostname UINT16 for the port UINT8 for the use 0 = Do not use 1 = Connect when UI is visible
0xFF10	DEVICE_IP_ADDRESS	Int32	IP_ADDRESS	Cards that have their own Ethernet port should use this OID to report their current IPv4 address.
0xFF11	FAN_SPEED_REQUEST	Int16	N/A	Used by cards in OG3-FR high power frame to request additional fan cooling. Card must send OGP_REPORT_PARAM for this OID periodically (not to exceed once per minute). Value of the parameter varies depending on the cooling capabilities of the frame.
0xFF12	OCCUPIED_SLOTS	Int16	N/A	Report the number of slots this card occupies. Value consists of two 8-bit fields, representing the number of additional slots to the left and right. Value = (left << 8) | (right)
0xFF13	UPLOAD_FILE_EXTENSIONS	String Array	N/A	Extensions of file types allowed to be sent to the device. Array elements have the format: “[Description]<ext:[extension without dot]>”
0xFF14	WIDGET_DESCRIPTORS	String	N/A	The URL (eo:// or http://) for the external object or HTTP to provide a widget descriptor file for widgets associated with the device.
0xFF16	DEVICE_CLASSES	String Array	N/A	List of device classes provided by your device — typically implemented by joystick, MIDI, or other game controller devices that provide a control surface. For example, a joystick might provide the “ptzjoystick” device class and associate its x-axis with “ptzjoystick.pan” and its y-axis with “ptzjoystick.tilt”.
0xFF17	DEVICE_AUTOWIRE _CLASSES	String Array	N/A	List of device classes that are consumed by your device, and can be used to tie a device to a control surface. The control surface usually contains one or more external physical controls that can be assigned to parameters in the software, allowing tactile control of the software. For example, a PTZ camera might consume the “ptzjoystick” device classes and use “ptzjoystick.pan” to control its tilt. If a device is using 0xFF16 to provide the “ptzjoystick” device class and has mappings for “ptzjoystick.pan” or “ptzjoystick.tilt” these parameters will be combined together so that the joystick controller can drive the camera.
0xFF18	DEVICE _ALIAS_TABLE	String	N/A	The URL (eo:// or http://) of a file containing an XML table to define string aliases for a device with numeric OIDs. This allows a device that uses numeric OIDs to participate in using device classes so that they can be controlled by external control surfaces, like a joystick controller or MIDI panels. The file format is in XML.

OID	Name	Type	Constraint	Function
0xFF19	FIRST_BIRTHDAY_FLAG	Int16	Choice	Indicates whether a connected device has been previously configured. Upon the first use of a device, DashBoard displays either the default panel of the device or a configuration wizard. You can define the default panel in the device tree and the configuration wizard using the wizard reserved OIDs. If a device has a configuration wizard, that wizard is only displayed when the device is in first birthday mode. • A value of 0 indicates that the device is not in first birthday mode. • A value of 1 indicates that the device is in first birthday mode. This allows devices to show a configuration wizard panel on the first use of the device. Once the user completes the configuration, the first birthday mode parameter value can be changed to 0. DashBoard checks the state of the first birthday parameter. It is never set by DashBoard, and it is up to the device to maintain its' first birthday state. When a user double-clicks on a node, and its first birthday (0xFF19) is set to a value of 1, then the following is checked, in order: • If the node has a “wizard-url”, the wizard is displayed. • If any of its parent nodes has a wizard-url, the closest parent’s wizard is displayed. • If none of the parents have a “wizard-url” and the WIZARD reserved OID (0xFF1A) is set to a valid URL, that wizard is displayed. • If none of the options above apply, then it is assumed that no configuration wizard exists. The default “config-url” panel is displayed.
0xFF1A	WIZARD	String	N/A	Provides an OGLML URL that describes a layout to use in place of the standard configuration screen in DashBoard. In first birthday mode, the wizard displays instead of the regular device user interface. Otherwise, the wizard can be launched by right-clicking on the device in the Tree View and selecting "Run Configuration".
0xFF1B	WIZARD_TOUCH	String	N/A	Provides an OGLML URL that describes a layout to use in Ultritouch in place of the standard configuration screen in DashBoard. In first birthday mode, the wizard will open instead of the regular device user interface. Otherwise, the wizard can be launched by navigating to the Connected Devices Interface, tapping on the ellipsis (three consecutive dots) to the right of the device you wish to configure, then tapping on "Run Configuration". **Note**: If WIZARD_TOUCH is not defined, the layout defined in WIZARD will be used.
0xFFFD	DEVICE_INDEX_URL_TOUCH	String	N/A	URL for a DashBoard Connect XML Definition for Ultritouch. **Note**: If 0xFFD is not defined, Ultritouch uses 0xFF0D. If you are not using Ultritouch see, “0xFF0D” on page 11–8.

OID	Name	Type	Constraint	Function
0xFFFE	DEVICE_OGLML_DESCRIPTOR_TOUCH	String	N/A	Provides an OGLML URL that describes a layout to use in place of the standard configuration screen in DashBoard for Ultritouch. **Note**: If 0xFFFE is not defined, Ultritouch uses 0xFF0E. If you are not using Ultritouch see, "0xFF0E" on page 11–8.
0xFF1C to 0xFFFF	RESERVED	...	...	Reserved for current or future use.

Reserved MFC and DashBoard Connect (slot 0) OIDs

Parameter OIDs in the range 0xFE00 to 0xFEFF have special significance for the MFC network controller (Slot 0) device. These also apply to any DashBoard Connect devices reporting on slot 0.

OID	Name	Type	Constraint	Function
0x0709	DOOR_STATE	Int16	N/A	**DEPRECATED OID** — Replaced by OID 0xFE0B (see page 11–12). Broadcast by the MFC every 10 seconds to indicate door status. 1= closed and 2= open.
0x803	SLOT_NAMES	Int16_Array	N/A	This array has one element for each slot in the frame. Each element's value is the OID of a String parameter whose value should be used as the name for the device in the given slot.
0x802	SLOT_DATA_SAFE	Int16_Array	N/A	This array has one element for each slot in the frame. 0 = DataSafe is enabled for the slot [Element #]. Default = DataSafe is disabled for slot [Element #] by the frame.
0xFE01	URM_STATE	Int16	N/A	States whether the frame requires a User Rights Management (URM) Enabled DashBoard (or a master password) to connect to DashBoard: • 0 (Default) — URM is not supported by the frame • 1 — URM is disabled/not required. • 2 — URM is enabled/required.
0xFE02	MASTER_PASSWORD	String (20-bytes max)	N/A	This is the value of the master password required by DashBoard users to connect when the User Rights Management server is not available and the URM State is enabled.
0xFE03	APPLY_BUTTON	Int16	Choice	The button DashBoard must press to apply changes to the master password or URM state parameters.
0xFE04	CANCEL_BUTTON	Int16	Choice	The button DashBoard can press to cancel any changes to the master password or URM state parameters. After the apply button has been pressed, this button does nothing.
0xFE05	DEVICE_CATEGORY	String	N/A	Default: "openGear Devices" Controls how items are grouped in User Rights Management and in the DashBoard tree view. Items sharing the same category are kept together.
0xFE06	FRAME_ICON	Int16	N/A	Contains an external object ID for an encapsulated icon. For more information, see "node-image-url" on page 2–19.
0xFE07	CONFIG_SLOT	Int16	N/A	Default: 0 The slot number for the device to open when the frame is 'opened' for configuration.

OID	Name	Type	Constraint	Function
0xFE08	CONFIG_URL	String	N/A	Default: [none] If defined and non-empty, the URL of a web page to open when the frame is 'opened' for configuration.
0xFE09	INDEX_URL	String	N/A	URL for a DashBoard Connect XML Definition.
0xFE0A	MASTER_PASSWORD_SAVE	String	N/A	Same as MASTER_PASSWORD OID 0xFE02 (see page 11–11) above, but used for internal storage on the MFC controller.
0xFE0B	FAN_DOOR_STATUS	Int16	N/A	Broadcast by the MFC every 10 seconds to indicate door status: • 1= closed • 2= open This replaces deprecated legacy OID 0x0709 (see page 11–11).
0xFE0C	FAN_AMBIENT_TEMP	Int16	N/A	Broadcast by the MFC every 10 seconds to report the ambient temperature of inlet air. 0 = fan door is open Otherwise, temperature in degrees Celsius
0xFE0D	FAN_SPEED_REPORT	Int16	N/A	Broadcast by the MFC every 10 seconds to report current door fan speed. 0 = minimum speed (or fan door open) Higher values indicate increasing speed. Max value depends on DFR frame type.
0xFE0E to 0xFEFF	RESERVED	…	…	Reserved for future use

Appendix E: CRC Computation

This section contains the algorithm used to compute the message content CRC for fragmented messages.

The 16-bit CRC for verifying message content is computed using the following algorithm (initial value = 0).

```
// Table used for computing the CRC
const int crcTable[] =
{
  0x0000,  0x1189,  0x2312,  0x329B,  0x4624,  0x57AD,  0x6536,  0x74BF,
  0x8C48,  0x9DC1,  0xAF5A,  0xBED3,  0xCA6C,  0xDBE5,  0xE97E,  0xF8F7,
  0x1081,  0x0108,  0x3393,  0x221A,  0x56A5,  0x472C,  0x75B7,  0x643E,
  0x9CC9,  0x8D40,  0xBFDB,  0xAE52,  0xDAED,  0xCB64,  0xF9FF,  0xE876,
  0x2102,  0x308B,  0x0210,  0x1399,  0x6726,  0x76AF,  0x4434,  0x55BD,
  0xAD4A,  0xBCC3,  0x8E58,  0x9FD1,  0xEB6E,  0xFAE7,  0xC87C,  0xD9F5,
  0x3183,  0x200A,  0x1291,  0x0318,  0x77A7,  0x662E,  0x54B5,  0x453C,
  0xBDCB,  0xAC42,  0x9ED9,  0x8F50,  0xFBEF,  0xEA66,  0xD8FD,  0xC974,
  0x4204,  0x538D,  0x6116,  0x709F,  0x0420,  0x15A9,  0x2732,  0x36BB,
  0xCE4C,  0xDFC5,  0xED5E,  0xFCD7,  0x8868,  0x99E1,  0xAB7A,  0xBAF3,
  0x5285,  0x430C,  0x7197,  0x601E,  0x14A1,  0x0528,  0x37B3,  0x263A,
  0xDECD,  0xCF44,  0xFDDF,  0xEC56,  0x98E9,  0x8960,  0xBBFB,  0xAA72,
  0x6306,  0x728F,  0x4014,  0x519D,  0x2522,  0x34AB,  0x0630,  0x17B9,
  0xEF4E,  0xFEC7,  0xCC5C,  0xDDD5,  0xA96A,  0xB8E3,  0x8A78,  0x9BF1,
  0x7387,  0x620E,  0x5095,  0x411C,  0x35A3,  0x242A,  0x16B1,  0x0738,
  0xFFCF,  0xEE46,  0xDCDD,  0xCD54,  0xB9EB,  0xA862,  0x9AF9,  0x8B70,
  0x8408,  0x9581,  0xA71A,  0xB693,  0xC22C,  0xD3A5,  0xE13E,  0xF0B7,
  0x0840,  0x19C9,  0x2B52,  0x3ADB,  0x4E64,  0x5FED,  0x6D76,  0x7CFF,
  0x9489,  0x8500,  0xB79B,  0xA612,  0xD2AD,  0xC324,  0xF1BF,  0xE036,
  0x18C1,  0x0948,  0x3BD3,  0x2A5A,  0x5EE5,  0x4F6C,  0x7DF7,  0x6C7E,
  0xA50A,  0xB483,  0x8618,  0x9791,  0xE32E,  0xF2A7,  0xC03C,  0xD1B5,
```

```
  0x2942,  0x38CB,  0x0A50,  0x1BD9,  0x6F66,  0x7EEF,  0x4C74,  0x5DFD,
  0xB58B,  0xA402,  0x9699,  0x8710,  0xF3AF,  0xE226,  0xD0BD,  0xC134,
  0x39C3,  0x284A,  0x1AD1,  0x0B58,  0x7FE7,  0x6E6E,  0x5CF5,  0x4D7C,
  0xC60C,  0xD785,  0xE51E,  0xF497,  0x8028,  0x91A1,  0xA33A,  0xB2B3,
  0x4A44,  0x5BCD,  0x6956,  0x78DF,  0x0C60,  0x1DE9,  0x2F72,  0x3EFB,
  0xD68D,  0xC704,  0xF59F,  0xE416,  0x90A9,  0x8120,  0xB3BB,  0xA232,
  0x5AC5,  0x4B4C,  0x79D7,  0x685E,  0x1CE1,  0x0D68,  0x3FF3,  0x2E7A,
  0xE70E,  0xF687,  0xC41C,  0xD595,  0xA12A,  0xB0A3,  0x8238,  0x93B1,
  0x6B46,  0x7ACF,  0x4854,  0x59DD,  0x2D62,  0x3CEB,  0x0E70,  0x1FF9,
  0xF78F,  0xE606,  0xD49D,  0xC514,  0xB1AB,  0xA022,  0x92B9,  0x8330,
  0x7BC7,  0x6A4E,  0x58D5,  0x495C,  0x3DE3,  0x2C6A,  0x1EF1,  0x0F78
};

/**  Compute the CRC for a byte - private inline version */
#define updateCrc(data, crc) (((crc >> 8) & 0xff) ^ crcTable [(crc ^ data) & 0xff])

/**
 *  Compute the CRC for a message (array of bytes).
 *  @param data - byte array to process
 *  @param length - number of bytes to process
 *  @param crc - the starting crc value
 *  @return - the updated crc value
 */
int getMessageCrc(char* data, int length, int crc)
{
  int ix;
  for (ix = 0; ix < length; ix++)
    crc = updateCrc(data[ix], crc);
  return crc;
}
```