

|  |  |
| --- | --- |
|  | **Common Software** |
|  |  |
|  | Gorba I/O over Medi (GIOoM) |
|  |  |
|  | Architecture |
|  |  |
|  | Technical Description |
|  |  |
|  | Version: 0.2 |
|  | State: Draft |
|  | Classification: Internal use only |
|  | Author: WES |
|  | Creation date: 2015-06-11 |
|  | Repository: $/Gorba/Main/Common/Gioom/Documents/TD\_GIOoM.docx |
|  | Gorba AG  Sandackerstrasse  9245 Oberbüren  Switzerland |

**Table of contents**

[1 Introduction 4](#_Toc422301507)

[1.1 Scope 4](#_Toc422301508)

[1.2 Intended Audience 4](#_Toc422301509)

[2 Product Overview 4](#_Toc422301510)

[2.1 Purpose 4](#_Toc422301511)

[2.2 System Overview 4](#_Toc422301512)

[2.3 Components 4](#_Toc422301513)

[2.3.1 Ports 4](#_Toc422301514)

[2.3.2 Clients 4](#_Toc422301515)

[3 Ports 5](#_Toc422301516)

[3.1 Naming 5](#_Toc422301517)

[3.2 Supported Value Ranges 5](#_Toc422301518)

[3.2.1 Integer Values 5](#_Toc422301519)

[3.2.2 Flag Values 5](#_Toc422301520)

[3.2.3 Enum Values 5](#_Toc422301521)

[3.2.4 Enum Flag Values 5](#_Toc422301522)

[3.3 Readable and Writable Ports 6](#_Toc422301523)

[4 Clients 7](#_Toc422301524)

[4.1 Common Methods 7](#_Toc422301525)

[4.2 GioomClient 7](#_Toc422301526)

[4.3 RemoteGioomClient 7](#_Toc422301527)

[5 Creating Ports 8](#_Toc422301528)

[5.1 Port Creation 8](#_Toc422301529)

[5.2 Port Registration 8](#_Toc422301530)

[6 Using Ports 9](#_Toc422301531)

[6.1 Finding Ports using GioomClientBase 9](#_Toc422301532)

[6.2 Finding Ports using PortListener 9](#_Toc422301533)

**Modification management**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Version** | **Date** | **Name** | **Dept.** | **Modifications** | **State** |
| 0.1 | 2015-06-15 | WES | SW Dev | Initial Version | draft |
| 0.2 | 2015-06-17 | WES | SW Dev | Added chapters 5 & 6 after reviews | draft |

**Review**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Version** | **Date** | **Name** | **Dept.** | **Remarks** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Release**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Version** | **Date** | **Name** | **Dept.** | **Remarks** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

# Introduction

## Scope

This document is a technical description of the Gorba I/O over Medi (GIOoM) protocol. It gives all information to Gorba software developers to be able to develop with and for this component.

This document is not intended to be a user manual.

The goal is to give an understanding of the architecture as well as the usage of GIOoM.

## Intended Audience

This document is written to be understood by Gorba software developers. Advanced technical skills are required.

# Product Overview

## Purpose

The Gorba I/O over Medi protocol (GIOoM – pronounced like the French word “Guillaume”, /ɡijom/) was created to discover, observe and interact with inputs and outputs available in a system. Previously each component had to interact directly with the hardware or provide a specific API to other components to do so.

## System Overview

The GIOoM protocol is – as its name suggests – based on the Message Dispatcher (Medi) protocol. GIOoM is actually only a set of messages exchanged over Medi together with an API for components to find, provide, read and write I/Os.

GIOoM has no configuration files, it solely relies on Medi (and its configuration file) as well as ports created through the API (see next chapters).

## Components

Every input or output in GIOoM is called a “port”. Ports are managed through the main components GioomClient and RemoteGioomClient. They provide all necessary API to interact with local and remote ports as well as register new local ports.

### Ports

Ports must have a unique name per application per unit. They are addressed with:

1. Unit name
2. Application name
3. Port name

Ports can be read-only, write-only or read-write.

### Clients

The two GIOoM client implementations (GioomClient and RemoteGioomClient) allow the user to search for ports and interact with them. But only the GioomClient can be used to register new ports.

# Ports

Ports may be actual hardware ports or a software state that can be used by other components. Examples of hardware ports are the ignition or the LED state; examples of software ports are the system volume or the CPU temperature.

As explained above in chapter 2.3.1, ports have a unique name and can be found by using that name. Ports are always in a specific state which is internally stored (and transmitted) as an integer. Usually the integer value should not be used directly but rather through the IOValue wrapper. Each port has a specific type of value (ValueBase, see chapter 3.2).

Ports are always represented by the IPort interface; information about a port is available in the IPortInfo interface provided by IPort.Info.

## Naming

A port must be named uniquely within the same application. It is possible that another application (and especially on another unit) can have a port with the same name.

## Supported Value Ranges

The following four different value ranges are supported by GIOoM. Usually it shouldn’t be necessary to extend this list since it would render different versions of GIOoM incompatible with each other. The value range of a port can be obtained by calling IPort.Info.ValidValues.

### Integer Values

Ports with an integer value (IntegerValues type) have a minimum and maximum and their value must be within the given range.

Examples for such ports are the system sound volume or the CPU temperature.

### Flag Values

Ports with a flag value (FlagValues type) can only have two states:

* False (0)
* True (1)

Examples for such ports are the ignition or the backlight state.

### Enum Values

Ports with an enum value (EnumValues type) can have a fixed list of values which each has a label assigned. Those values are exclusive, meaning the port has exactly one value at any time.

Examples for such ports are the DVI level shifter or the RS-485 interface configuration.

### Enum Flag Values

Ports with enum flag values (EnumFlagValues type) can have multiple valid values each with a label assigned. For this to work properly, the basic values must all be 2n where n is between 0 and 31.

Currently there are no ports of this type available in the TFT system.

## Readable and Writable Ports

Each port may or may not be read and / or written. The following properties provide this information:

* IPort.Info.CanRead
* IPort.Info.CanWrite

The following matrix shows how those categories of ports are called and gives examples:

|  |  |  |
| --- | --- | --- |
|  | CanRead = true | CanRead = false |
| CanWrite = true | **Read/write:**   * Digital output * Backlight state | **Write-only:**   * System time |
| CanWrite = false | **Read-only:**   * Digital input * CPU temperature | n/a |

# Clients

GIOoM ports are always accessed using clients. Usually, an application just uses the singleton GioomClient.Instance, but when only access to remote ports is required, it is also possible to use RemoteGioomClient.

Both classes provide methods to find and access ports. The GioomClient gives also the possibility to register local ports so they can be found by other components and applications.

## Common Methods

All client implementations (subclasses of GioomClientBase) can be used to find ports. Ports can either be searched by address and name or the list of all known ports can be retrieved. When searching for ports, it is always possible to use wildcards. Like this for example “all ports on Unit ‘X’” or “all ports called ‘Y’” can be found.

Once a port is found, the provided IPortInfo can be used to open the port for reading and/or writing. A port should always be closed once it is no longer needed. Especially remote ports use resources by regularly re-registering to port value changes and the actual port value changes being sent over Medi.

## GioomClient

The GioomClient class is always accessed through the singleton property GioomClient.Instance. The object can then be used to find local and remote ports as well as register new local ports. Those ports are then announced to all clients searching for this port.

## RemoteGioomClient

Remote clients provide no additional functionality over the common methods. It is important to dispose of a remote client once it is no longer used since it registers to certain Medi messages which can lead to a considerable amount of data transferred over a period of time.

# Creating Ports

This chapter explains how a new GIOoM port is created and then registered to the GioomClient to be available to other components.

## Port Creation

Any port that you want to publish through GIOoM must implement IPort. In most cases, the default implementation SimplePort can be used. The following example assumes that SimplePort is used; other cases can be deduced from this example or found in code.

When creating a SimplePort, you need to specify its name, access rights (read/write), the type of the value as well as its initial value. The value type can be any of the supported types (see chapter 3.2). Most of them allow to specify their range or a list of supported values; it is important to note that these parameters can’t be changed after the port was created.

Examples:

ignitionPort = new SimplePort(

"Ignition", true, false, new FlagValues(), FlagValues.True);

rs485InterfacePort = new SimplePort(

            "RS485Interface",

true,

true,

EnumValues.FromEnum<Rs485Interface>(),

(int)Rs485Interface.At91);

The created SimplePort actually always allows to read and write its value (even if this was restricted through the constructor parameters. This is required so that the value can actually be changed by the component that created the port.

If the port is writable and its value is changed from the outside, the ValueChanged event can notify the creator of the port.

## Port Registration

Once an IPort implementation is created (and configured), it can be registered with GIOoM. It is important that the port will only be accessible from other components once it was registered and will stay so until it is being deregistered again.

Example for registration:

GioomClient.Instance.RegisterPort(ignitionPort);

Example for deregistration:

GioomClient.Instance.DeregisterPort(ignitionPort);

# Using Ports

There are two ways of using ports: the raw, original way is by using a GioomClientBase implementation (e.g. GioomClient.Instance) and the simplified access using PortListener. The PortListener approach allows for continuous search for a port, so that a port can be found even if it is not available at the moment when the PortListener is created.

## Finding Ports using GioomClientBase

The class GioomClientBase allows to find ports by their name and address. You can either search for all ports matching the address provided or simply open one single port by providing the exact name and address.

Example for a port search:

GioomClient.Instance.BeginFindPorts(

    address,

    timeout,

    ar => this.ShowPorts(GioomClient.Instance.EndFindPorts(ar)),

    null);

This search will go on until the given timeout is reached and then the information for all ports found during that time will be returned as IPortInfo objects. An IPortInfo can then be used to actually open the port (which is a synchronous method call):

var port = GioomClient.Instance.OpenPort(portInfo);

// do something with the port

port.Dispose();

If the address and port name are known ahead of time, it is enough to call the asynchronous OpenPort:

GioomClient.Instance.BeginOpenPort(

address,

"Ignition",

this.UsePort(GioomClient.Instance.EndOpenPort(ar)),

null);

This will search exactly once during 10 seconds for the port with the given name and address. The address can contain “\*” wildcards allowing to open the first port with the given name on any Unit matching the address.

## Finding Ports using PortListener

The disadvantage of accessing ports directly through GioomClientBase (see chapter 6.1) is that if the port doesn’t exist yet, it is up to the user to restart searching for the port. This can especially happen at start-up when one application may want to access a port that hasn’t been created by another application (or that application is not yet running at all).

PortListener will continuously poll for the given port and will update its Port and Value properties once the port is found. The Value shouldn’t be accessed until the port was found (which is when you get for the first time the ValueChanged event.

Example:

var listener = new PortListener(address, "Ignition");

listener.ValueChanged += (s, e) => this.ShowIgnition(listener.Value);

Despite its name, PortListener allows also to change the port value if it supports writing.