



| | | | |
|---|--|--------------------|----------|
|  | OMNI Summit Microservice V1.X User Manual | Doc. Number | Rev. |
| | | OMNI-SM-015 | 1 |

Contents

| | |
|--|-----------|
| 1 Document Purpose | 3 |
| 2 Document Scope | 3 |
| 3 Background | 3 |
| 4 Overview of gRPC | 3 |
| 5 Overview of Protobuf files | 3 |
| 6 Setup and Installation of the OMNI Summit Microservice | 4 |
| 7 Example 1 – use in JavaScript applications | 5 |
| Overview | 5 |
| Pre-requisites | 5 |
| 1. Node.js | 6 |
| 2. Obtaining protos files | 6 |
| 3. Sense Configuration Files | 6 |
| Architecture | 7 |
| Getting Started | 7 |
| Starting the OpenMind Server | 8 |
| Running the application locally | 8 |
| 8 Example 2 – Use in Python Applications | 10 |
| Overview | 10 |
| Pre-requisites | 10 |
| Installation | 10 |
| Building the Protos | 11 |
| Starting the OpenMind Server | 11 |

| | | | |
|---|--|--------------------|----------|
|  Open Mind | OMNI Summit Microservice V1.X User Manual | Doc. Number | Rev. |
| | | OMNI-SM-015 | 1 |

| | |
|---|----|
| Running the Python Client | 11 |
| Code Walkthrough | 12 |
| 9 Interfaces | 16 |
| 10 SOUP | 16 |
| Risk Control Measures Implemented in the OMNI Summit Microservice | 16 |
| 11 Documentation | 16 |
| 12 Reference Information | 16 |
| Definitions | 16 |
| 13 Approvals | 18 |

| | | | |
|---|--|--------------------|----------|
|  Open Mind | OMNI Summit Microservice V1.X User Manual | Doc. Number | Rev. |
| | | OMNI-SM-015 | 1 |

1 Document Purpose

This document provides a series of examples for integration and use of the OMNI Summit Microservice.

2 Document Scope

This document is intended to provide information about the use and limitations of the OMNI Summit Microservice application, and guidance on how to integrate this microservice into client applications.

3 Background

Today, neural stimulation applications are written in such a way that little to no code-reuse is possible. The purpose of the OMNI Summit Microservice is to provide a hardware-agnostic layer to manage connections between the Medtronic Summit Clinician Telemetry Module (CTM) and the Summit RC+S Implantable Neural Stimulator (INS).

The OMNI Summit Microservice achieves this goal by leveraging gRPC, a remote procedure call library based on HTTP/2 protocol. OMNI Summit Microservice uses the Protobuf interface description language to define messages and their associated types.


4 Overview of gRPC

gRPC is an industry-developed, open-source high-performance Remote Procedure Call (RPC) framework that facilitates software interoperability via networked application programming interfaces. It can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking, and authentication. It is also applicable in the last mile of distributed computing to connect devices, mobile applications and browsers to backend services.

<https://grpc.io/>

5 Overview of Protobuf files

gRPC uses the Protobuf interface description language (IDL) to define services, endpoints, and messages used across interfacing software.

| | | | |
|---|--|--------------------|----------|
|  | OMNI Summit Microservice V1.X User Manual | Doc. Number | Rev. |
| | | OMNI-SM-015 | 1 |

Protocol buffers provide a language-neutral, platform-neutral, extensible mechanism for serializing structured data in a forward-compatible and backward-compatible way. It's like JSON, except it's smaller and faster, and it generates native language bindings.

Protocol buffers are a combination of the definition language (created in .proto files), the code that the proto compiler generates to interface with data, language-specific runtime libraries, and the serialization format for data that is written to a file (or sent across a network connection).


Protobuf comes with a variety of tools to generate both client and server code for 10 supported programming languages (with more languages supported through the open-source community)

<https://developers.google.com/protocol-buffers/docs/overview>

6 Setup and Installation of the OMNI Summit Microservice

Setup steps for Windows 10 computers

1. Make sure you have a research agreement with Medtronic to gain access to their Summit RDK. The DLLs provided therein are required for the OMNI Summit Microservice to build and run.
NOTE: the Medtronic Summit DLL's are NOT provided by Openmind and are confidential.
2. Download the latest OMNI Summit Device Microservice release from:
<https://github.com/openmind-consortium/OmniSummitDeviceService/releases>
3. Run the OmniSummitMicroservice.msi windows installer file found in the downloaded zip above. The installer will create an OpenMind folder in C:/Program Files (x86) with the executable, as well as desktop and start menu shortcuts.
4. In the Summit RDK, navigate to DLLs/AnyCPU/ where you should see the Medtronic-provided DLLs.
5. Copy the following files from the DLLs/AnyCPU/ folder:
 - Medtronic.NeuroStim.Olympus.dll
 - Medtronic.SummitAPI.dll

| | | | |
|---|--|--------------------|----------|
|  | OMNI Summit Microservice V1.X User Manual | Doc. Number | Rev. |
| | | OMNI-SM-015 | 1 |

- NLog.dll
- wclCommon.dll
- Medtronic.TelemetryM.dll
- wclBluetoothFramework.dll
- wclCommunication.dll

These files are necessary to run the OmniSummitMicroservice.

6. Paste the DLLs copied above into the same folder as the OmniSummitDeviceService.exe executable, which should have this filepath:

C:/Program Files (x86)/OpenMind/OmniSummitMicroservice

7. Running the OmniSummitMicroservice executable will open a terminal window and launch the server. You may now connect to it with a client application.

7 Example 1 – use in JavaScript applications


Overview

The OMNI Reference Client is a desktop application built using JavaScript Electron.js Library to interface with OMNI compatible device services. The OMNI client provides a way, through use of the OMNI Summit Microservice, to find and connect to up to two Medtronic Summit RC+S systems, configure data streams, record data, and troubleshoot connections.

The order of operations is as follows:

1. Connect to the OpenMind Server
2. Find and connect to one or two bridges
3. Find and connect to one or two devices
4. Configure sensing on the Summit RC+S
5. Stream time domain data from the Summit RC+S

Pre-requisites

| | | | |
|---|--|--------------------|----------|
|  | OMNI Summit Microservice V1.X User Manual | Doc. Number | Rev. |
| | | OMNI-SM-015 | 1 |

1. Node.js

First, install node js using the instructions on <https://nodejs.org/en/>

2. Obtaining protos files

Github

If the repository was cloned from github, git already linked the OmniProtos files as a submodule. The original repository for OmniProtos can be found here (<https://github.com/openmind-consortium/OmniProtos.git>). To obtain those files, first clone the OmniReferenceClient repository:

```
git clone https://github.com/openmind-consortium/OmniReferenceClient.git
```

Then, move to the cloned directory and initialize the submodule:

```
git submodule init
```

and update the submodule:


```
git submodule update
```

Zip file

If the repository was obtained using a zip file. create a new folder in the home directory of the repository called protos and copy the .protos files from the protobuf deliverable into that folder.

3. Sense Configuration Files

In the packaged version of the app, the sensing config files will not be bundled with the app and needs to be copied into this directory on windows: /AppData/Roaming/omniconfig. Examples of these files can be found in the config folder of this repository:

| | | | |
|---|--|--------------------|----------|
|  | OMNI Summit Microservice V1.X User Manual | Doc. Number | Rev. |
| | | OMNI-SM-015 | 1 |

- config.json - main config file, the name field of the left and right objects need to be updated with the serial number of both the CTM and the INS: "//summit/bridge//device/".
- senseLeft_config.json - sensing config for the left INS, make sure that this file is in the same directory as config.json.
- senseRight_config.json - sensing config for the right INS, make sure that this file is in the same directory as config.json.

Architecture

OMNI client is built using electron. The application is broken into three separate processes: renderer, main and preload. The renderer process handles the user interface. The main process interfaces with the backend via gRPC. To do this, a library called grpc-js is installed with the dependencies. In the main process, bridgeClient and deviceClient are called when needed to interface with the backend. For example, deviceClient.ListDevices function is called to retrieve device information from the backend.

The preload acts as a security layer between the main and renderer process. The config.forge section of package.json contains the actual configuration for all of the processes.


The user interface is built using React. Communication between the renderer process and the main process uses the Electron Context Bridge to mitigate security risks exposed by using Electron's ipcRenderer directly in the renderer process.

The OMNI device service is a stateless backend (with the exception of connection state management). Similarly, the main process has no state. The main process acts as a passthrough from the renderer to the OMNI device service. All states are managed by the React application.

The entrypoint for the renderer process is /src/renderer/index.tsx, the entrypoint for preload is /src/preload/index.ts and the entrypoint for the main process is /src/main/index.ts.

Getting Started

| | |
|--|--------------|
| Document is for reference only – not for as-is submission to FDA | Page 7 of 18 |
|--|--------------|

| | | | |
|---|--|--------------------|----------|
|  | OMNI Summit Microservice V1.X User Manual | Doc. Number | Rev. |
| | | OMNI-SM-015 | 1 |

Starting the OpenMind Server

Open the OpenMind Server project in VisualStudio. Build and run it. A terminal window will appear showing that the server is running on localhost:50051.

Running the application locally

First, clone repository if not already done so:

```
git clone https://github.com/openmind-consortium/OmniReferenceClient.git
```

Make sure that the OmniProtos and sense configuration files are set up as explained above. Next, navigate in the terminal to the source folder and install the node dependencies:

```
npm install
```

Next, start the development server:

```
npm start
```

Electron forge does not support hot-reloading. To reload the OMNI client after you've made changes to the source type rs and hit enter from the terminal where you started the development server.


To package the application run:

```
npm run make
```

To lint the code run:

```
npm run lint
```

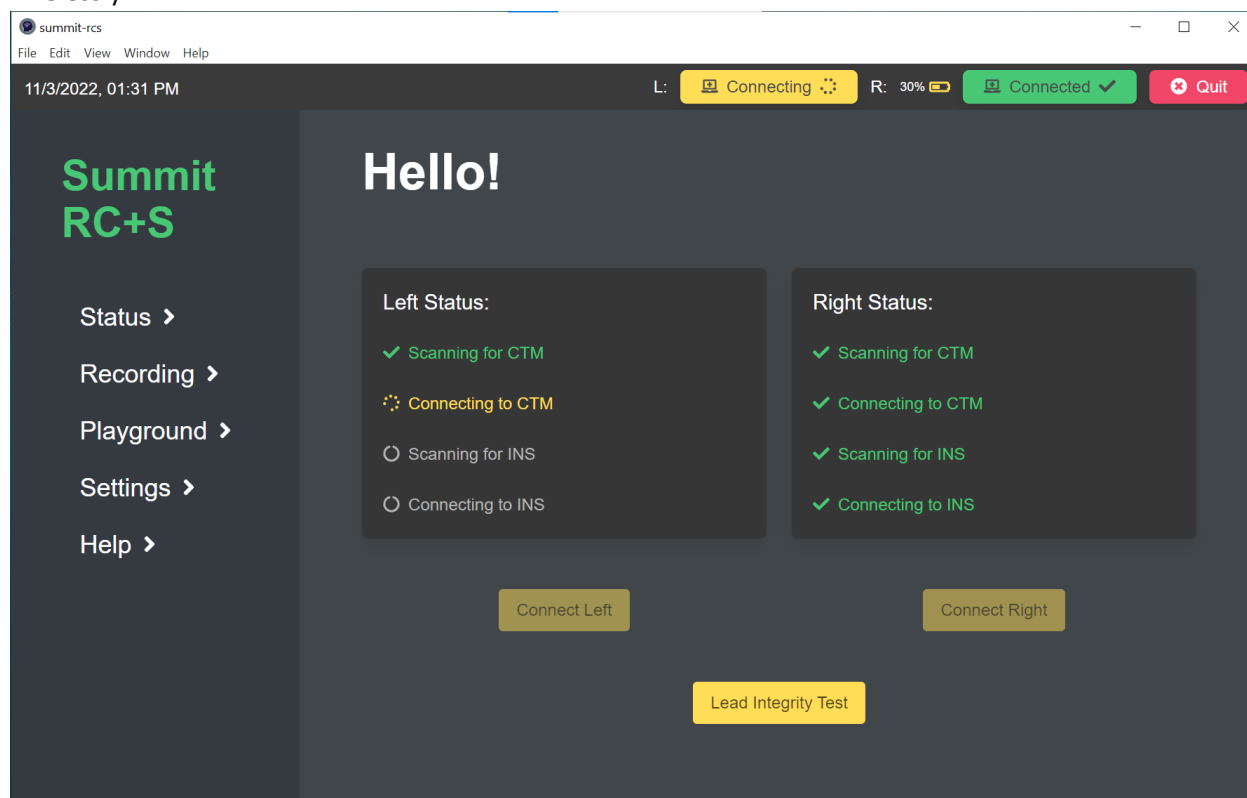
To have the linter fix errors, run:


| | | | |
|---|--|--------------------|----------|
|  Open Mind | OMNI Summit Microservice V1.X User Manual | Doc. Number | Rev. |
| | | OMNI-SM-015 | 1 |

```
npm run lint -- --fix
```

To edit the application, use either github to clone the repository (<https://github.com/openmind-consortium/OmniReferenceClient>) or open the zip file. In the protos folder, copy the google protobuf files included in the package. Make sure dependencies needed to run javascript and Electron are installed in the computer, including but not limited to node.js, and open a command prompt at the root directory.

Note: As per the Medtronic Summit RDK Manual, the first time that you connect to a CTM it must have a wired connection to the base computer. Successive connections to the same CTM can then be made wirelessly.



| | | | |
|---|--|--------------------|----------|
|  | OMNI Summit Microservice V1.X User Manual | Doc. Number | Rev. |
| | | OMNI-SM-015 | 1 |

The streamed data is saved at the location - C:/ProgramData/Medtronic ORCA/SummitData/OMNI Summit Device Service

Refer to the Summit User Manual for the format of the data.

8 Example 2 – Use in Python Applications

Overview

The OpenMind Server provides a gRPC interface for interacting with the Medtronic API. This allows for custom applications to be built on any operating system using any of the gRPC supported programming languages. These examples show how to stream time domain data from the Summit RC+S using Python.

The order of operations is as follows:

6. Connect to the OpenMind Server
7. Find and connect to a bridge
8. Find and connect to a device
9. Configure sensing on the Summit RC+S
10. Stream time domain data from the Summit RC+S

Pre-requisites

This demo streams data from the Summit RC+S and prints time series data packets to the terminal, therefore a Summit RC+S and accompanying CTM are required.


The OpenMind Server is used to broadcast data from the INS that will be picked up by this code. Follow instructions on the [OpenMind Server GitHub](#) and compile the project using Visual Studio.

This sample will run on any major operating system (Linux, Windows, Mac) as long as Python3 is installed. To install the required Python packages, please run: pip install grpcio

Installation

Run the following command to download the [GitHub](#) repository

```
git clone https://github.com/openmind-consortium/Omni-Python-Examples.git
```

| | | | |
|---|--|--------------------|----------|
|  | OMNI Summit Microservice V1.X User Manual | Doc. Number | Rev. |
| | | OMNI-SM-015 | 1 |

Notice the submodule [OmniProtos](#). This contains the gRPC protobuf files that will help this Python client communicate with the OpenMind Server (which is written in C#).

There are two main files in the repository: client.py and build_protos.sh. client.py is the Python script containing the code for this example. build_protos.sh is a Bash script to build the protobuf files found in the OmniProtos submodule.

Building the Protos

This example depends on gRPC protobuf files, which need to be built before the code is run. build_protos.sh is a Bash script that can be used to build the protobuf files on a Linux machine (or on Windows Subsystem for Linux [WSL]). To run it on those systems use the command:

```
./build.protos.sh
```

For other operating systems, make a directory at path/to/github/repo/protos and run the following Python command as described in the [gRPC Python documentation](#):

```
python -m grpc.tools.protoc -IomniProtos --python_out=./protos
--grpc_python_out=./protos OmniProtos/*.proto
```

This protos folder is imported as a Python module into client.py. To make the protos directory a proper Python module, create an empty file inside called `__init__.py`.

Starting the OpenMind Server


Open the OpenMind Server project in VisualStudio. Build and run it. A terminal window will appear showing that the server is running on localhost:50051.

Running the Python Client

Description of what the python code is doing

Now it's time to run client.py. There is one optional flag for an ip address of the OpenMind Server. If not provided, the default location is localhost. However, if the server is running on a different IP address, it can be specified using the flag as follows:

```
python client.py --ip <ip_address>
```

| | | | |
|---|--|--------------------|----------|
|  | OMNI Summit Microservice V1.X User Manual | Doc. Number | Rev. |
| | | OMNI-SM-015 | 1 |

Now the application is running. Terminal output will print as the bridge and device are connected, and as sensing is configured, finally resulting in the time domain packet data being printed in a continuous stream. The printing will stop when the program is stopped.

Code Walkthrough

This code can be easiest to read starting at the bottom and working backwards. Let's start at the very end of the run.py file with the entrypoint:

```
if __name__ == '__main__':
    logging.basicConfig()
    parser = argparse.ArgumentParser(description='Parsing input
commands')
    parser.add_argument('--ip', type=str, nargs='?',
default='localhost',
    help='The IP address where the OpenMind Server is running')
    args = parser.parse_args()
    run(args.ip)
```


This section is where we parse the optional argument for an IP address, which will change the default ip address from localhost, to a user specified address. That parameter is then fed into the `run()` function, which is the main function for the rest of the example.

The `run()` function starts by opening a gRPC connection to the Omni server using the specified IP address and the default port for the server.

```
def run(ip_addr):
    with grpc.insecure_channel(ip_addr+':50051') as channel:
```

Next, it creates two stubs, one for the bridge and one for the device. These stubs are the objects that are used to call functions on the server, and return the data back to the client. They will do most of the heavy lifting for carrying out operations between the server and our client example.

```
# Initialize stubs
```

| | | | |
|---|--|--------------------|----------|
|  | OMNI Summit Microservice V1.X User Manual | Doc. Number | Rev. |
| | | OMNI-SM-015 | 1 |

```
bridge_stub = bridge_pb2_grpc.BridgeManagerServiceStub(channel)
device_stub = device_pb2_grpc.DeviceManagerServiceStub(channel)
```

To better understand how these stubs will work, let's look at the next lines inside the `run()` function.

```
# Look for bridges
bridges = find_bridges(bridge_stub)
```

The first step of this code example is to use the bridge stub to find any bridges that are available on the network. This is accomplished with the helper function `find_bridges()` which takes as input the bridge stub. Looking inside the `find_bridges()` function to see how it works:

```
# Looks for any bridges connected to the host machine
# The Bridge ID must (partially) match partial_uri
def find_bridges(bridge_stub, bridge_id=''):
    partial_uri = '//summit/bridge/' + bridge_id
```

The Summit API uses the notion of Partial URIs to find bridges. For bridges, the URIs always start with `//summit/bridge/` and end with a unique ID for each bridge. When searching for bridges, if the ID is already known, it can be provided as an argument to the function as the variable `bridge_id`. This will narrow the search and make sure only that one bridge is found. However, to search for all bridges, `bridge_id` can be left as an empty string.

```
# Create a request to search for all connected bridges
# who's ID matches partial_uri
print('Looking for bridges')
query_request = bridge_pb2.QueryBridgesRequest(query=partial_uri)
```

| | | | |
|---|--|--------------------|----------|
|  | OMNI Summit Microservice V1.X User Manual | Doc. Number | Rev. |
| | | OMNI-SM-015 | 1 |

Next, the function prints that it is looking for bridges and creates the official query request using the `bridge_pb2` object, which is one of the files generated from the protobuf compilation from an earlier step.

```
# Receive a response from the server. This contains the list of
# bridges that were found
response = bridge_stub.ListBridges(query_request)
```

Then it uses the `bridge_stub` to send the request to the `ListBridges()` function, which will return a response object from the Omni server containing any information about the available bridges.

```
# Iterate through the list of bridges
for b in response.bridges:
    print('Found bridge:', b.name)
```

A for loop can be used to open the response object and print the bridges that were found.

```
# Return the bridges as a list


return response.bridges
```

Lastly, this list is returned back to the `run()` function.

```
# Look for bridges
bridges = find_bridges(bridge_stub)

# Look through each bridge that was found
for bridge in bridges:
```

Back inside the `run()` function, a for loop can again be used to loop through each bridge and attempt to connect and stream data from them.

| | | | |
|---|--|--------------------|----------|
|  | OMNI Summit Microservice V1.X User Manual | Doc. Number | Rev. |
| | | OMNI-SM-015 | 1 |

```

# Attempt to connect the each bridge
bridge_connection_status = connect_to_bridge(bridge_stub,
                                             bridge)


# If the bridge connected
if bridge_connection_status == 1:
    # Search for devices on that bridge
    devices = find_devices(device_stub, bridge)

    # Look through each device that was found
    for device in devices:
        # Attempt to connect to each device
        device_connection_status =
            connect_to_device(device_stub, device)

        # If the device connected
        if device_connection_status == 1 or
           device_connection_status>4:
            # Configure sensing
            configure_sensing(device_stub, device)
            # Stream data from that device
            stream_data(device_stub, bridge, device)

```

For each bridge, the same sequence of events occurs. Using the `connect_to_bridge()` helper function, a connection is attempted to the bridge. If `bridge_connection_status` returns the value 1, meaning the connection was accepted, then it attempts to find devices connected to that bridge. Similarly to how the bridges come back as a list, so do the devices. These get cycled through in a for loop as well, and a connection is attempted to each device. If a connection is established, then sensing is configured and data is streamed and printed to the terminal.

| | | | |
|---|--|--------------------|----------|
|  | OMNI Summit Microservice V1.X User Manual | Doc. Number | Rev. |
| | | OMNI-SM-015 | 1 |

9 Interfaces

OMNI Summit Microservice is a network only service. All interactions between the client and the OMNI Summit Microservice occur over gRPC, and thus HTTP/2, via network ports. All gRPC services, endpoints, and messages are defined in the Protobuf interface description files.

Please see the Interface Control Document found locally as index.html in the Documentation/Interface Control Document folder in the downloaded package.

10 SOUP

OMNI Summit Microservice considers the Summit RC+S DLLs software of unknown provenance.

Risk Control Measures Implemented in the OMNI Summit Microservice

Developers of the client application are responsible for conducting risk analysis and appropriately mitigating identified risks. Risks may be identified that can be partially or fully mitigated by features implemented in the OMNI Summit Microservice. Risk control measures implemented in the OMNI Summit Microservice are identified in the Software Requirements Specification (SRS) with an asterisk (*).

The client application's risk analysis documentation can apply these risk controls through citation of the OMNI Summit Microservice SRS document number and relevant requirement identifier(s) directly in the using application's risk analysis document. Verification of these risk controls is accomplished through software verification of the OMNI Summit Microservice on each released version of the microservice.

11 Documentation

Documentation for the OMNI Summit Microservice implementation is autogenerated by Doxygen and is represented by the Interface Control Document (ICD).


12 Reference Information

Definitions




| Term | Definition |
|------|------------|
|------|------------|

| | | | |
|---|--|--------------------|----------|
|  Open Mind | OMNI Summit Microservice V1.X User Manual | Doc. Number | Rev. |
| | | OMNI-SM-015 | 1 |

| | |
|------|--|
| gRPC | gRPC Remote Procedure Call framework (grpc.io) |
|------|--|

| | | | |
|---|--|--------------------|----------|
|  Open Mind | OMNI Summit Microservice V1.X User Manual | Doc. Number | Rev. |
| | | OMNI-SM-015 | 1 |

13 Approvals

| Approver Role | Signature and Date |
|----------------|---|
| Project Lead | DocuSigned by:  11/13/2022 1:38 PM EST |
| Lead Developer | DocuSigned by:  11/13/2022 11:31 PM PST |
| Author | DocuSigned by:  11/13/2022 1:57 PM EST |