conbots
CONnected through roBOTS

Technical Manual and User Manual

Central Control Software

Version: 1

December 2023

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 871803

1. DOCUMENT REVISION LOG

VERSION	REVISION	DATE	DESCRIPTION	AUTHOR
1	0	12/12/23	Technical & User Manual of CCS v1	Francesco Di Tommaso

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

2. TABLE OF CONTENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

3. TECHNICAL MANUAL

3.1. Introduction

3.1.1. Overview of the software and intended use

The Central Control Software (CCS) is an open-source platform for the integration of technical equipment in research scenarios, encompassing commercial devices, prototypes, software applications and APIs. It has been developed within the framework of the CONBOTS project, which aims at designing a new class of robots to physically couple people and facilitate learning of complex motor tasks, such as playing a musical instrument or handwriting. The project is built upon the development of four enabling technologies, each of them representing a module of the software architecture and possibly including a number of submodules. An overview of the CONBOTS modules and related submodules is listed below:

1. **Bi-directional User Interface,** including:
   1.1. Wearable Sensors
   1.2. Instrumented Objects
   1.3. Augmented Reality (AR)
2. **Robotic Haptic Devices,** including:
   2.1. Exoskeletons
   2.2. End-Effector Devices
3. **Machine Learning Algorithms,** estimating the psychophysiological state of the user
4. **Interaction Control Algorithms,** controlling haptic devices for physical human-human robot-mediated communication
5. **Subsidiary Devices,** including all external devices used for testing purposes.

The list of currently integrated modules, which will be covered in detail in Section 3.4.1, includes:

1. **Wearable Sensors**: set of devices to record and analyse motion and physiological parameters in a real-time, non-invasive manner.
   1.1. XsensDOT: light-weight portable inertial measurement units (IMU) from Xsens.
   1.2. Shimmer3 GSR+ Unit (hereafter, Shimmer): wireless galvanic skin response (GSR) sensor from Shimmer Sensing.
   1.3. Zephyr Bioharness (hereafter, Bioharness): wearable recording device for human monitoring from Medtronic's Zephyr.
2. **Machine Learning model** (hereafter, Machine Learning): novel machine learning algorithm to estimate the engagement of an individual from data collected by the set of Wearable Sensors.
3. **Instrumented Objects**: set of smart objects for real-time estimation of an individual's motor performances.
4. **AR games**: augmented reality games to provide individuals with immersive environments and increase active engagement.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

5. **Exoskeleton**: 2 actuated Degrees Of Freedom (DOF) upper-limb exoskeleton for rendering haptic feedback to individuals performing a motor task.
6. **Myro**: interactive upper-limb rehabilitation device developed by Tyromotion.
7. **Subsidiary Devices**: set of modules not originally envisioned in the overall system, but later included to facilitate experiments, testing and analysis.
    7.1. Multimedia Player: application for playing multimedia computer files. As a default multimedia player, VLC has been chosen due to its cross-platform and open-source nature.
    7.2. Multimedia Recorder: device for capturing multimedia files, such as video and audio. An external webcam is considered a good compromise between quality of recording and portability.
    7.3. Xsens MTw: IMUs from Xsens.
    7.4. Optitrack: motion capture and 3D tracking system from Optitrack.
    7.5. Metronome: digital device that produces an audible click at regular intervals.
    7.6. OpenSoundControl: system to synchronize computer and multimedia devices.

The core component of the CONBOTS platform is the *Central Control Software* (CCS) (Figure 1), that it is devoted to the management of all modules and responsible for the following high-level tasks:

1. **Synchronization.** This process is essential to guarantee a synchronized real-time functioning of the integrated modules. The CCS will manage the connection with every module and broadcast a common trigger signal to synchronously start/stop the recording of the data (*UDP Synch*).
2. **Data recording and pre-processing.** The platform is in charge of recording the data streamed by each module (*UDP Data*) and performing a pre-processing phase in those cases that require a high-level platform-mediated sharing of data among different modules.
3. **Control of physical communication.** An essential task performed by the CCS is the control of physical communication, i.e., the control of coupling stiffness in the four enabling technologies, as well as the parameters of the AR games, i.e., the difficulty level or the avatar appearance, according to the inputs received from Machine Learning algorithms about the user status.
4. **Data storage.** The CCS is responsible for the storage of raw and pre-processed data in a proper format (e.g., .txt, .csv), specific for each integrated submodule.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

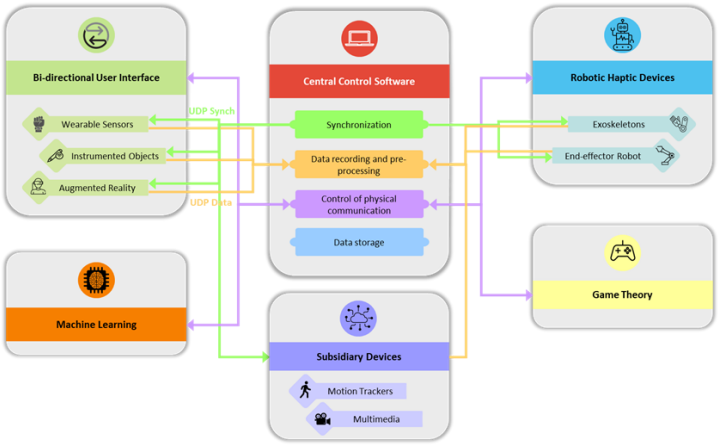

Figure 1: Overview of the high-level software architecture of the CONBOTS platform.

The application has been developed as a Windows Form Application in C# using Visual Studio Community 2022 as Integrated Development Environment (IDE), while the middleware is composed of routines in different programming languages, such as Python, NodeJs, C#, C and LabView.

The present document addresses technical aspects of the design, development and maintenance of the software that makes it directed mainly to other developers or system administrators interested in extending the functionalities of CCS or integrating it with other systems.

3.1.2. Aim of the technical manual

This document serves as a comprehensive guide to understanding, deploying, and optimizing the functionality of the CCS. The primary objectives of this manual are to provide detailed insights into the features, functionalities, and configuration options of CCS, to guide system administrators through the installation, setup and maintenance process, to facilitate developers in extending the capabilities of CCS and integrating other equipment.

The description of system requirements, installation procedure, components and modalities on how to integrate additional modules will assume that the reader has the technical background to understand the contents of the document and will focus on the open-source project rather than on the executable file.

3.2. System Requirements

3.2.1. Hardware requirements

The following requirements for the operating machine are needed:

- Processor: ARM64 or x64 processor, Quad-core or better (recommended)
- RAM: 4 GB (minimum), 8 GB (recommended)
- Disk space: minimum of 850 MB up to 210 GB of available space for the installation of Visual Studio, depending on the installed features.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

- Operating System (OS): at least Windows 10 (64-bit) is recommended.

3.2.2. Software requirements and installation

The customization of the application requires the use of Visual Studio Community 2022 as IDE for Windows Form Application and the installation of .NET Framework 4.8. The detailed procedure to install Visual Studio Community 2022 will not be covered in the present document because it has already been extensively described by the provider Microsoft here, as well as the installation of .NET Framework 4.8, which is covered here.

Some of the devices integrated in the platform have their own control routine that serves as a middleware node in the high-level architecture (Section 3.4.1) and whose execution in dedicated processes is managed directly by CCS. These APIs can be developed with different frameworks and programming languages, and can be customized from existing Software Development Kits (SDK) or developed from scratch, therefore the functioning of the CCS depends on some other requirements:

- Miniconda (*Bioharness*): Python package and environment manager, part of the Conda open-source system. The latest version of miniconda3 should be downloaded with built-in Python version (Python 3.9).
- Node.js (*XsensDOT*): open-source cross-platform JavaScript runtime environment, can be installed from here.
- Node-gyp (*XsensDOT*): cross-platform command-line tool for compiling native addon modules for Node.js. Documentation can be found here.
- Zadig (XsensDOT): Windows application to install or update generic USB drivers. Documentation and installer can be found here.

Once Python 3.9 has been installed or even if it was previously installed, the following packages must be downloaded or checked (by using the `pip list` command):

- OpenCV (*Multimedia Recorder*): Python package for computer-vision applications and capturing devices. Documentation can be found here.
- PyAudio (Multimedia Recorder): Python open-source library for recording or playing audio files. Documentation can be found here.
- NumPy (*Multimedia Recorder*): Python open-source library for manipulating multi-dimensional arrays and matrices. Documentation can be found here.
- PySerial (*Shimmer*): Python package for accessing to serial ports. Documentation can be found here.

Depending on the specific applications installed on user's computer, a Python IDE might be needed for customizing the provided APIs. A recommended suggestion could be Visual Studio Code, a cross-platform source-code editor developed by Microsoft and currently considered the most popular developer environment tool.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

3.3. Installation

The latest release of the CCS can be found in the official GitHub repository at https://github.com/fraditommaso/Central-Control-Software, both as customizable open-source project and pre-built binaries provided in the "Release" folder.

3.3.1. Windows Installer

The following steps must be completed to install the CCS on a Windows machine:

1. Download the CCS installer for Windows from the official repository.
2. Run the installer executable and follow the on-screen instructions.
3. During the installation, specify the installation directory.
4. Complete the installation process.

3.3.2. CCS Project

The entire project folder containing Visual Studio solution, dependencies and APIs can be downloaded from the repository. The folder is structured as follows:

- *AudioVideoRecorder_Python*: it contains the Python packages needed for the API `AudioVideoRecorder.py`, which controls a recording video device, such as the laptop built-in camera or an external webcam.
- *BioHarness*: it contains the dependencies for controlling Bioharness, as well as its official user manual.
- *CentralControlSoftware*: it contains the Visual Studio Community solution that can be open and customized, the installer and the executable files.
- *Media*: it contains any multimedia file that the application might need, such as the software icon and a collection of .wav files storing digital metronomes with different beats per minute.
- *OpenSoundControl*: it contains the library added in the references of the software to manage the module.
- *Optitrack*: it contains the library for remotely triggering the proprietary software Motive.
- *Shimmer*: it contains the Python API to control the module and the official user manual.
- *XsensDOT*: it contains the NodeJS project to control the devices and the official user manual.
- *XsensMTw*: it contains the firmware of a microcontroller to control the receiving station with analogue connection.

Finally, an additional folder "Data" is created by the CCS to store any recorded data. If the folder already exists, then CCS use it without creating a new one.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

3.4. System architecture

3.4.1. High-level design

High-level architecture

The platform has been engineered both at high level and low level with the aim of satisfying all the requirements envisioned during the initial stage of the design process. In the multi-tiered architecture presented in Figure 2, the CCS represents the core of the system which is in charge of managing the connection with each module and holding the components of the Graphical User Interface (GUI) that interacts with the user. The modules are not directly connected to the core but each one is controlled by its own API. Therefore, the communication between the core and the external layer of modules is mediated by a middleware of APIs, which acts as abstract units loosely coupled with the core, keeping the interaction limited to the transmission and reception of data in a predefined structure. This choice promotes a robust architecture that is not affected by possible low-level modifications.

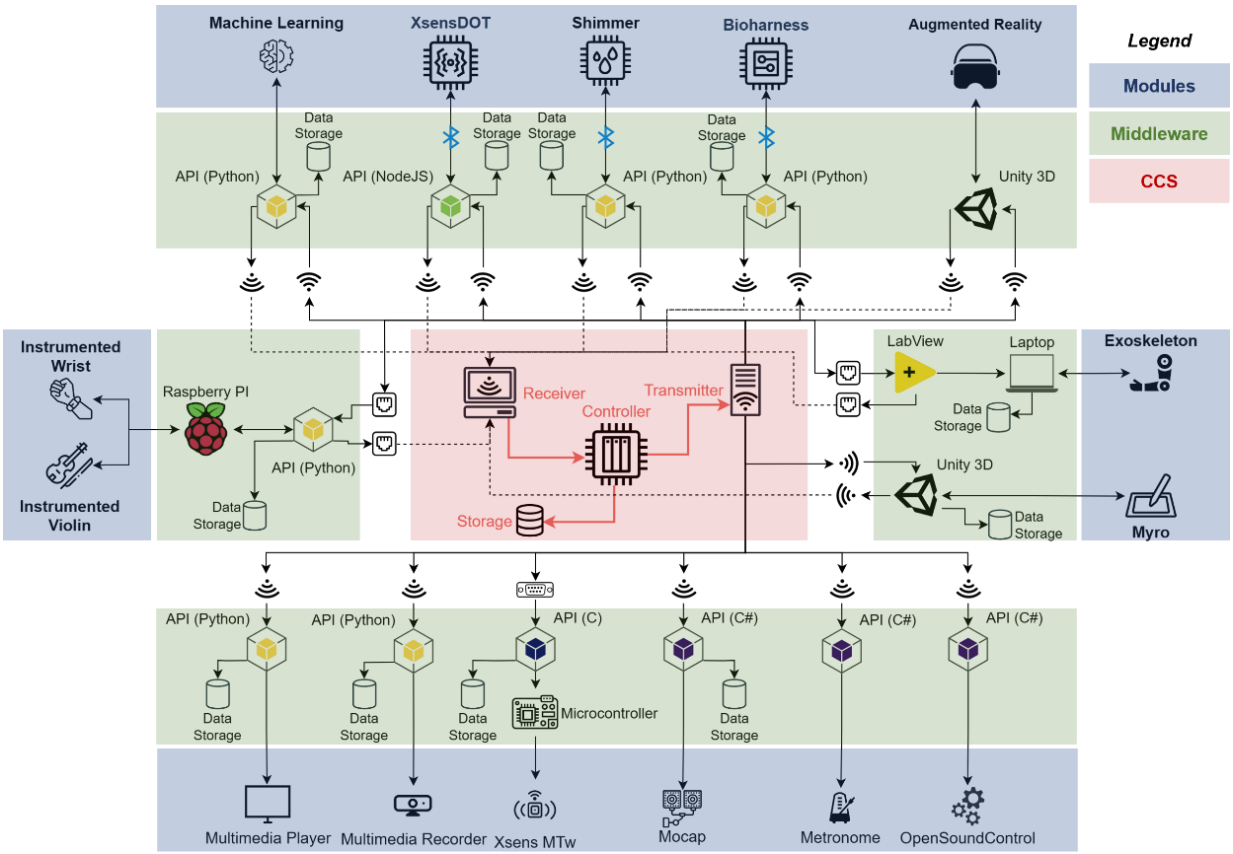

Figure 2: High-level architecture of CCS

The list of integrated devices has been presented in Section 3.1.1.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

3.4.2. Low-level design

Low-level architecture

Modularity and flexibility of the architecture are achieved by selecting an OSI-compliant communication protocol to comply with different types of modules, i.e., they can be hardware or software modules, controlled via Bluetooth, Wi-Fi, Ethernet or serial communication. Specifically, UDP protocol has been identified as the most suitable one due to its high transmission rate. It provides a secure and real-time communication and enables the definition of a standard communication model between the CCS core and a generic device: CCS serves as a common transmitter for all modules of a start/stop recording command (UDP Synch) and as a receiver of data, while the API of a generic module serves as a data transmitter (UDP Data) and command receiver. Both CCS and APIs are equipped with a transmission and reception socket, defined by an IP address and port numbers (Figure 3).

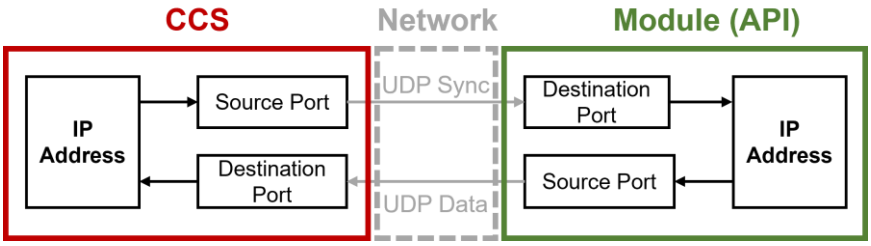

Figure 3: UDP conventional communication model.

From a programming perspective, this aspect is crucial since it allows the definition of a single interaction model between the CCS and a generic module, which then can be extended to all modules (Figure 4). In particular, this model can be translated into an object-oriented abstract class, which outlines common attributes and methods and can be used as a template by other classes, namely the child classes. The exploitation of inheritance and encapsulation leads to the definition of an easily extendable architecture, where the effort of adding a new module is substantially limited by the fact that most of the required logic has already been implemented in one abstract class (whose code is also easier to maintain).

The abstract class should possess the attributes required to establish a UDP socket for data transmission and reception, as well as specific methods to perform these operations. In particular, a synchronous transmitter and an asynchronous receiver have been implemented, with the latter enabling the optimization of data receipt through the network and preventing operational blocks by leveraging the asynchrony of the reception. These operations should be concurrent and independent, i.e., the software should have the capability of transmitting and receiving data from a module at the same time in a scalable manner, i.e., regardless of the number of modules. These essential requirements can be fulfilled with multithreading programming, which allows the concurrent execution of multiple parts of a program and the maximization of its responsiveness, throughput, and efficiency.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

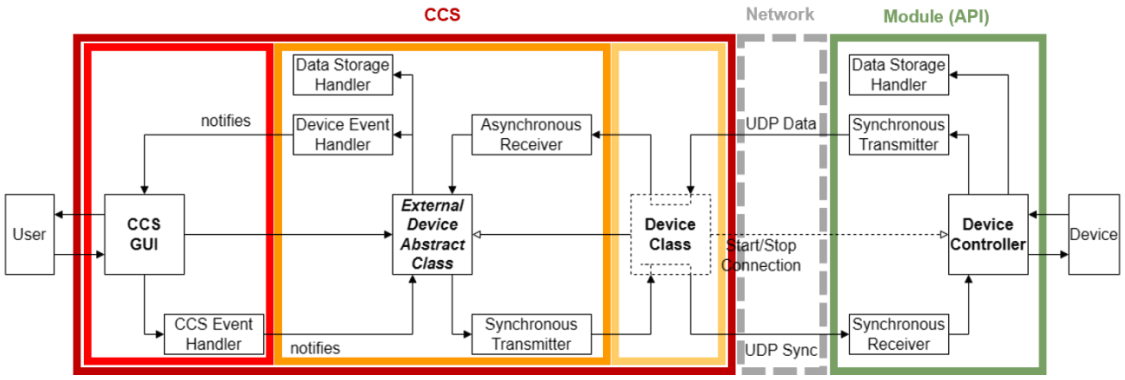

Figure 4: Low level architecture diagram.

Moreover, the abstract class should be able to exchange information with the core of the platform, which will be a class called CCS and will be used to implement the GUI, and vice versa. This mutual exchange of information should be reliable and suitable to let the components be sufficiently loosely coupled to ensure that they can be developed and modified independently. An event-driven paradigm has been exploited to fulfil these requirements, specifically, event handlers which allow a component to "notify" the occurrence of an event to any other component interested in knowing when that particular event has occurred. In particular, the abstract class needs to know when a start/stop command from the CCS has to be sent to the APIs, and, on the other hand, the CCS is interested in knowing when a new packet from the API is received by the abstract class. This paradigm is particularly helpful from the CCS side to broadcast common information to multiple modules at specific occurrences, such as the trigger that synchronizes data reception from each connected module.

The previously presented design choices significantly reduced the cost of extending the functionalities of the system. For instance, the integration of a new module will consist of implementing a new child class which will inherit most of the logic from the abstract class and will require the custom implementation of only three methods to perform the following operations: *start the connection with the module*, *interpret the packets according to a predetermined convention*, *stop the connection with the module*.

Class diagram

The class diagram provides a visual representation of CCS object-oriented design, showing the classes that compose the system, their relationship and interactions within the system. It is crucial to define a clear class diagram by taking into account every design choice previously decided because even before starting programming it can give a clear indication of the requirements that have been fulfilled. For instance, it is straightforward to demonstrate from the diagram shown in Figure 5 that the envisioned system has a modular and extensible architecture, because of the presence of abstract classes that facilitate the integration of additional modules by exploiting the advantages of inheritance. Moreover, the use of multiple instances of the same module is easily manageable by simply instantiating multiple objects from the same class.

Two main components can be identified in CCS class diagram:

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

1. *CCS*: it is the class that holds all the components needed to create the GUI presented to the user. It has several attributes, some of which defined *a-priori* and some other changing according to the interaction of the user with the system, and several methods, which are mainly managed by event handlers bind to the interactive components.
2. *ExternalDevice*: it is the abstract class that holds all the components needed to manage the connection, synchronization and data reception from the integrated modules. It has several properties and methods, some of which are overridden and some other inherited by the child classes. In particular, the only methods that need to be overridden are the ones to start and stop the connection with the module (*StartConnection* and *StopConnection*), and to properly interpret the packets received from the module (*UnpackArrayData*).
3. *Device*: every single module included in the architecture has its own child class which inherits properties and methods from ExternalDevice class. Some of the methods are overridden and customized to the specific modules.
4. *DeviceProperties*: it is the class to encapsulate the properties of a child class that depend on the specific instantiation.
5. *DataHandler*: it is the class that handles the visualization on real-time charts of incoming data.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

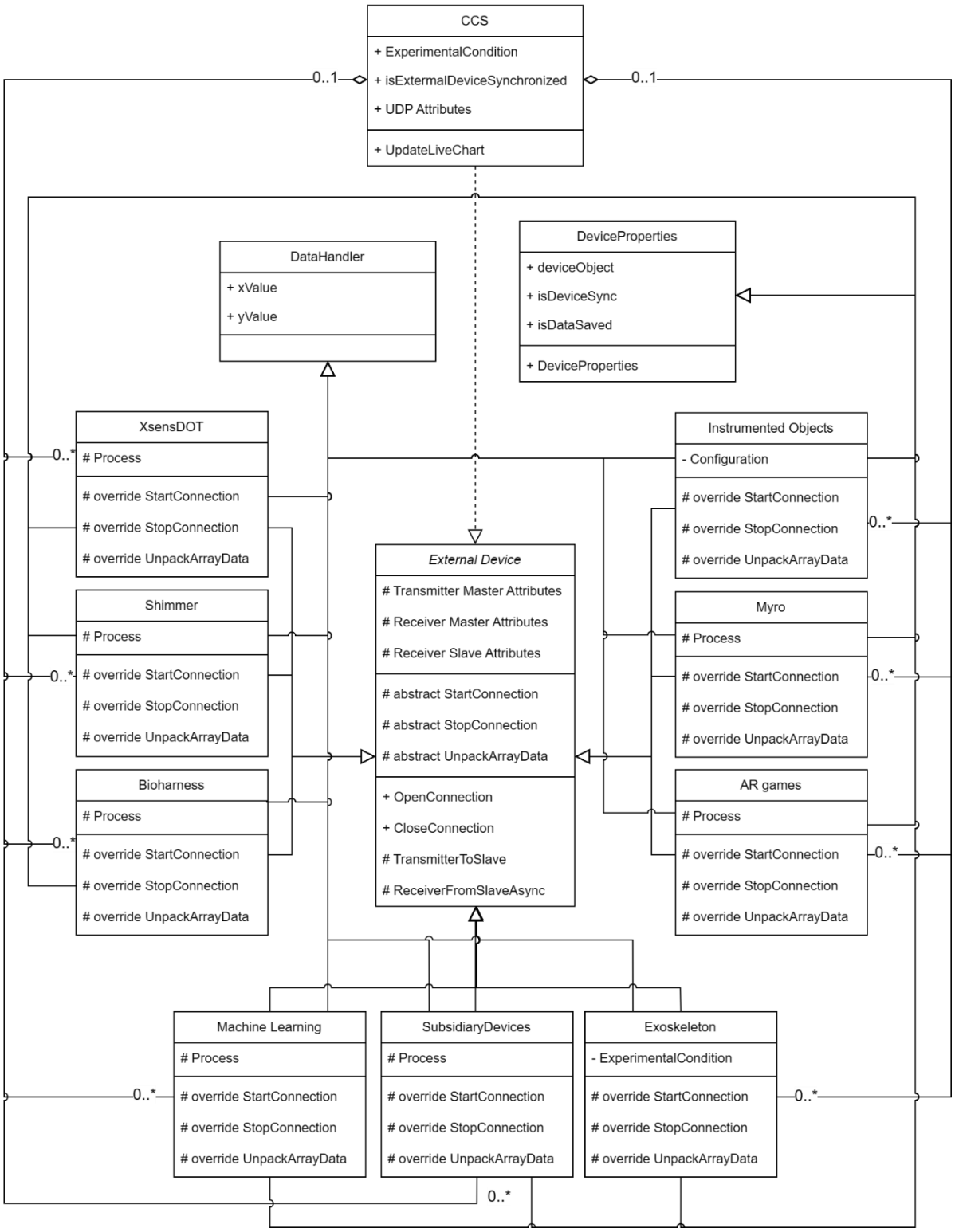

Figure 5: Class Diagram of the CCS.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

3.5. Components and modules

3.5.1. CCS class

The CCS class serves as the core component of the application, encapsulating the GUI and providing the necessary functionality for interacting with integrated modules (Figure 6). This class is essential in managing data collection, connections with external modules, and facilitating user-friendly controls.

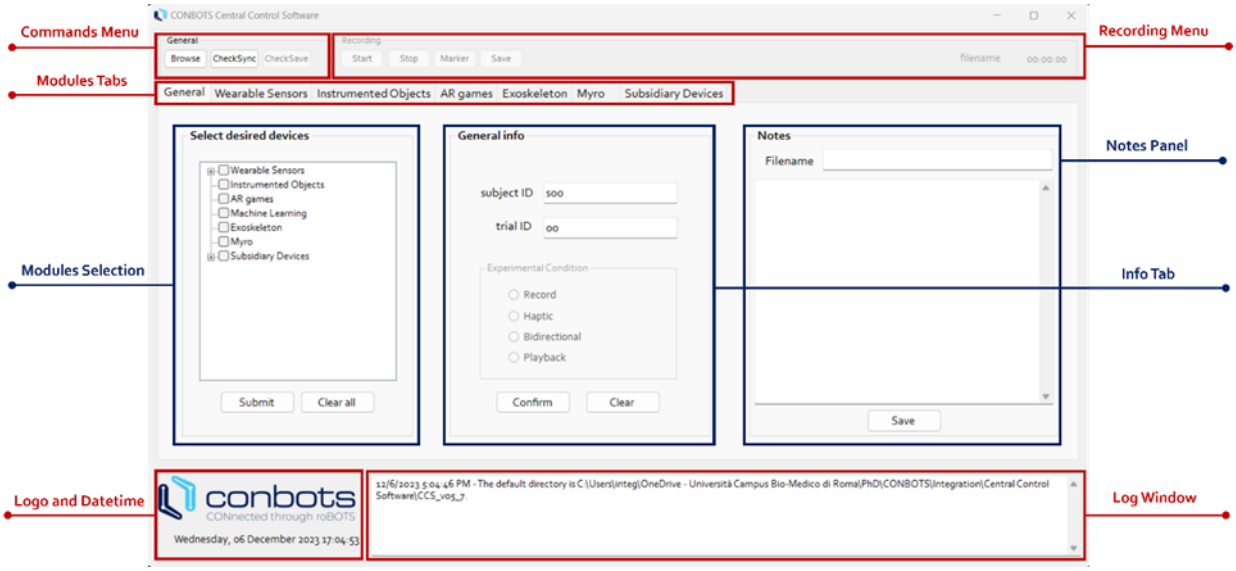

Figure 6: Overview of the CCS GUI prompted to users.

The class is structured as follows:

1. Definition Properties and accessory classes, such as *DeviceProperties* and *DataHandler*.
2. Definition of the constructor of the class, which contains the initialization of the necessary components.
3. Definition of methods to manage high-level components and operations.
4. Definition of methods bind to specific GUI components
   - 4.1. *Main Menu*
     - 4.1.1. *Commands Menu*
     - 4.1.2. *Recording Menu*
   - 4.2. *Modules Tab*
     - 4.2.1. *Tab General*
     - 4.2.2. *Tab Wearable Sensors*
     - 4.2.3. *Tab Instrumented Objects*
     - 4.2.4. *Tab AR games*
     - 4.2.5. *Tab Exoskeleton*
     - 4.2.6. *Tab Myro*
     - 4.2.7. *Tab Subsidiary Devices*

The CCS class follows an event-driven paradigm, utilizing event handlers to efficiently communicate with other components. Besides the event handlers bind to GUI components, another essential event-

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

handler is implemented, namely *NotificationRaised*, which handles the notification of significant event from the CCS to subscribers, i.e., classes derived from *ExternalDevice* abstract class, such as start and stop trigger events, high-level mediated data, markers, names of files or folders.

3.5.2. External Device abstract class

The *ExternalDevice* class serves as an abstract blueprint for generic external modules integrated into the system and implements the logic required to communicate with each module API. By encapsulating common properties and methods, this class streamlines the process of extending the functionalities of the platform with new modules. The class has several attributes, some of which are valid for any child class and some other are specific, as well as several methods, some to be inherited as they are and some other to be overridden by the child class. In particular, the class is structured as follows:

1. Definition of properties.
2. Definition of the constructors of the class, with the initializations of the necessary components, such as UDP properties, endpoints and threads. Specifically, UDP properties must be defined in the constructors.
3. Definition of overridden methods
   3.1. Method to start connection (*StartConnection*)
   3.2. Method to stop connection (*StopConnection*)
   3.3. Method to interpret received data (*UnpackArrayData*)
4. Definition of inherited methods.

The design of the *ExternalDevice* class embraces an object-oriented approach, leveraging inheritance and encapsulation. This allows for the creation of child classes, each representing a specific type of external module, which can inherit the common attributes and methods from this abstract class. To integrate a new external module, developers need only to implement a new child class inheriting from the *ExternalDevice* class. The child class requires customization of three key methods: starting the connection, interpreting packets, and stopping the connection. This design choice significantly reduces the complexity and effort required when extending the system. Moreover, the implementation of multithreading programming within this class ensures concurrent and independent data transmission and reception. This capability, essential for scalability, allows the system to handle multiple modules simultaneously without compromising responsiveness.

The class is also built upon an event-driven paradigm, not only because it subscribes to the events notified by CCS class but also because it implements another event handler, namely *DataFromExternalDevice*, to manage the notification to CCS class of every packet received from the module's API.

3.5.3. Device child class

Encapsulating the variable properties in an abstract class facilitates the implementation of a versatile model that is largely independent on the type of module, with the inherited methods being defined only for the abstract class. The flexibility lies in the fact that only three methods are specific for each module and have to be overridden. Specifically, the methods to unpack incoming packets are strictly

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

dependent on the convention agreed with the transmitting module on the exact structure of the transmitted packets, while the methods to start and stop the connection can be classified according to the type of API.

Overridden methods for a child class with a local API

A generic module, whether it is an hardware device or a software application, that is controlled by an API, that runs on the same machine as the CCS, is considered to have a "*local API*". A local API has an endpoint in the network that exploits the same local IP address as CCS but simply has a different port and can be run directly by CCS in a separate process. Therefore, when establishing the connection with the module, i.e., with its API, the CCS must be able to start the routine on a separate process and this requires specific logic.

Overridden methods for a child class with a remote API

On the other hand, a module whose API is run on a separate machine is considered a "*remote API*" and only requires the definition of the endpoints in the network, significantly reducing the programming load. A crucial aspect is the definition of the IP address, which has to be manually set to avoid potential errors in the code due to dynamic IP address which might vary unpredictably.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

3.6. Integration of additional modules

3.6.1. Customization of the GUI

The integration of an additional module requires both the creation of specific GUI components and the customization of CCS class with the implementation of the required logic.

The first customizable component is the *Modules Selection* panel, where users can select the desired modules and developers should include any additional one by modifying the list of nodes (Figure 7). The list can be customized by entering both a single node and a new category that holds multiple nodes.

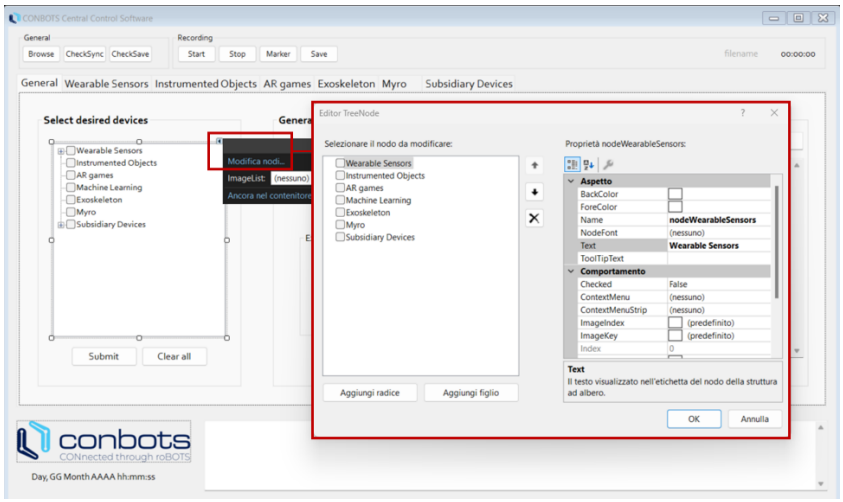

Figure 7: Customization of *Modules Selection* Panel.

The subsequent step is the creation of all the components related to the newly entered module, i.e., a dedicated tab in *Modules Tabs* or a panel in another tab, buttons to trigger specific events, labels to show information to the user or chart areas to hold real-time visualization of data. To add a new tab, first select the tab control in the GUI, then inspect its properties and select *TabPages* (Figure 8); this will prompt an editor where new tabs can be implemented and customized.

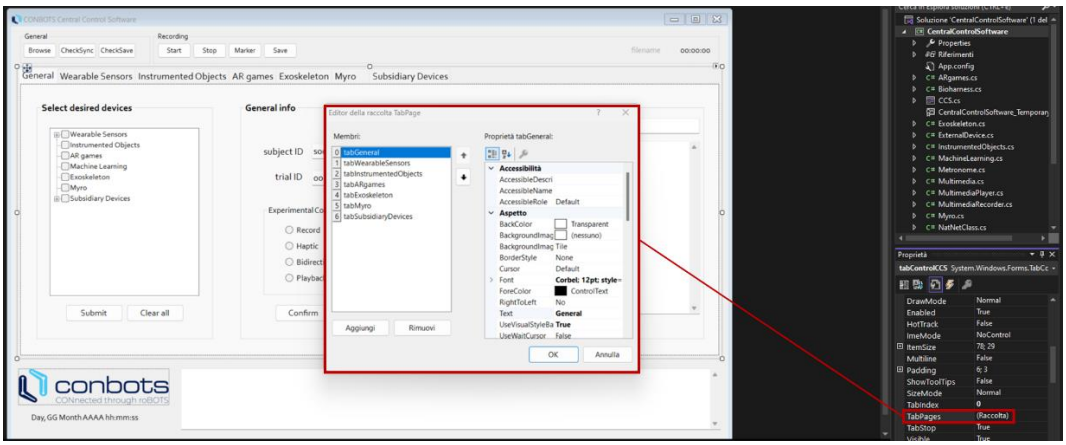

Figure 8: Adding a new tab to *Modules* Tabs.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

The class is engineered to facilitate the process by significantly reducing the programming load on the developers and limiting the coding of additional logic to few easily-identifiable sections denoted with warning "CUSTOMIZE". To this aim, the first customizable section of the code is delimited in the region "*Instantiation of objects from child classes (CUSTOMIZE)*", where each module is associated with a specific object instantiated from the corresponding child class, an object to hold the properties of the module and an object to manage the data within the GUI components, as shown in Figure 9.

```
#region Instantiation of objects from child classes (CUSTOMIZE)
// If a new class of objects is included, please instantiate the object here and then modify
// the method buttonConfirmDevices_Click.
public XsensDOT xsensDot;
public Bioharness bioharness;
public Shimmer shimmer;
public InstrumentedObjects instrumentedObjects;
public MachineLearning machineLearning;
public ARgames arGames;
public Exoskeleton exoskeleton;
public Myro myro;
public Serial serialPort;
public XsensMTw xsensMTw;
public Optitrack optitrack;
public OscSynch oscSynch;
public Metronome metronome;
public MultimediaRecorder multimediaRecorder;
public MultimediaPlayer multimediaPlayer;

public DataHandler shimmerDataHandler;
public DataHandler bioharnessDataHandler;
public DataHandler xsensDOTDataHandler;
public DataHandler instrumentedObjectsDataHandler;
public DataHandler machineLearningDataHandler;
public DataHandler arGamesDataHandler;
public DataHandler exoskeletonDataHandler;
public DataHandler myroDataHandler;

public DeviceProperties xsensDotProperties;
public DeviceProperties bioharnessProperties;
public DeviceProperties shimmerProperties;
public DeviceProperties machineLearningProperties;
public DeviceProperties instrumentedObjectsProperties;
public DeviceProperties arGamesProperties;
public DeviceProperties exoskeletonProperties;
public DeviceProperties myroProperties;
public DeviceProperties xsensMtwProperties;
public DeviceProperties optitrackProperties;
public DeviceProperties multimediaRecorderProperties;
public DeviceProperties multimediaPlayerProperties;
public DeviceProperties metronomeProperties;
public DeviceProperties oscSynchProperties;
```

Figure 9: Instantiation of objects from child classes

The second section that requires active developer's intervention related to *Modules Selection* panel, where users can select the desired modules. From a programming point of view, the list shown to the users is a tree view that is implemented in the region of the class entitled "*Tab General: Tree View Manager (CUSTOMIZE)*". Within this region, two methods have to be customized (Figure 10):

- `buttonConfirmDevices_Click(object sender, EventArgs e)`
- `buttonClearAllSelected_Click(object sender, EventArgs e)`

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

The former implements a switch statement nested in a foreach loop to instantiate objects from every module selected in the tree view, while the latter uses the same structure to clear every object previously created and release every associated resources.

```
private void buttonConfirmDevices_Click(object sender, EventArgs e)
{
    foreach (string checkedNode in treeViewCheckedNodes)
    {
        bool deviceFound = false;
        try
        {
            switch (checkedNode.Substring(checkedNode.LastIndexOf('\\') + 1))
            {
                case "XsensDOT":
                    if (xsensDot == null)
                    {
                        xsensDot = new XsensDOT(transmitterIP, 50111, transmitterIP, 50114, transmitterIP, 50112);

                        // Custom dictionary
                        xsensDotProperties = new DeviceProperties(xsensDot, false);
                        selectedDevices.Add("XsensDOT", xsensDotProperties);

                        deviceFound = true;
                        groupBoxXsensDot.Enabled = true;

                        xsensDOTDataHandler = new DataHandler();
                    }
                    break;
                case "Bioharness":
                    if (bioharness == null)
                    {
                        bioharness = new Bioharness(transmitterIP, 50131, transmitterIP, 50134, transmitterIP, 50132);

                        bioharnessProperties = new DeviceProperties(bioharness, false);
                        selectedDevices.Add("Bioharness", bioharnessProperties);

                        deviceFound = true;
                        groupBoxBioharness.Enabled = true;

                        bioharnessDataHandler = new DataHandler();
                    }
                    break;
```

Figure 10: Integration of additional modules within CCS tree view manager.

After this region, the class holds several other regions, one for each tab created in the GUI; thus, the developers should create their own region and implement there the logic required to manage the components that the user is interacting with and whose events are triggered.

If the GUI has been equipped with real-time charts for data received by the additional module, the developer must also customize the section of the class dedicated to updating the chart areas, namely "*Real-time charts of data from objects (CUSTOMIZE)*". Here, the delegate `ExternalDevice_DataFromExternalDevice(object sender, ExternalDevice.DataFromExternalDeviceEventArgs e)` will be called anytime a new event is notified from a device object, i.e., anytime a new packet is received.

Finally, the class implements the method `CCS_FormClosing(object sender, FormClosingEventArgs e)`, which is triggered when the user is closing the GUI and that serves as a backup method to release all the resources that may still be used by some of the objects.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

3.6.2. Creation of a device child class

After implementing the required logic inside CCS class, developers should create the child class of the integrated module by declaring its specific attributes, defining its constructor and customizing the overridden methods. Specifically, the constructor must comply with one of the possible constructors defined in the parent class and assign some of the overridden attributes, such as:

- `externalDeviceName`: the name of the new module.
- `columnHeadersExternalDevice`: the sequence of variable names to interpret data received in a packet.
- `ackMessageSynchronization`: the acknowledgement message sent by the API to confirm the synchronization of the module with CCS.
- `triggerStart`: the specific character or string used to notify to the API the start of recording
- `triggerEnd`: the specific character or string used to notify the API the stop of recording.

An example of constructor for the child class *Bioharness* is shown in Figure 11.

```
// CONSTUCTOR ...
1 riferimento
public Bioharness(string _transmitterIpMaster, int _transmitterPortMaster, string _receiverIpMaster, int _receiverPortMaster,
    string _receiverIpSlave, int _receiverPortSlave) : base(_transmitterIpMaster, _transmitterPortMaster,
        _receiverIpMaster, _receiverPortMaster, _receiverIpSlave, _receiverPortSlave)
{
    externalDeviceName = "Bioharness";
    columnHeadersExternalDevice = string.Join("\t", "Trigger", "Timestamp",
        "Heart Rate", "Respiratory Rate", "ECG", "Breathing", "\n");
    ackMessageSynchronization = "sync done";
    triggerStart = "1";
    triggerEnd = "0";
}
```

Figure 11: Example of child class constructor for Bioharness module.

The overridden methods are strictly dependent on the type of integrated module and whether its API runs locally or remotely and on the structure of packets streamed.

3.7. Troubleshooting

3.7.1. Common issues and solutions

In Table **1** the list of common issues is reported together with the corresponding solutions.

Table 1. Common issues and solutions.

	Issue	Solution
1	Python APIs return error on a package not included.	Install the missing package by referring to official documentation.
2	UDP port already in use.	Check the list of used ports/endpoints and change port number.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

3	Device class with a remote API not receiving data.	Check network configurations, firewall settings, and ensure that the routine on the remote machine is actively streaming data.
4	Device class with a local API not receiving data.	Refer to error messages, logs, or consult the Python script documentation.

If issues not included in Table **1** are found, developers are kindly asked to submit a report to github repository to allow effective maintenance and further improvements of the platform.

4. USER MANUAL

4.1. Introduction

4.1.1. Overview of the software and intended audience

The software is designed to facilitate the integration in a single platform of several modules, such as commercial devices, custom devices and prototypes, control routines or APIs. It has been developed in Visual Studio Community 2022, which provides programmers with an Integrated Development Environment (IDE) ideal for exploiting built-in libraries and testing systematically the application.

The CCS is mainly thought to address specific issues of daily research activities, which requires the use of technologies that need a high-level mediation to exchange information with each other. This is especially true with commercial devices, marketed from different manufacturers and as such often not easily usable in combination with other equipment. The software is designed to simplify complex tasks and empower users with efficient solutions with its user-friendly interface that does not require specific programming skills and can be easily used by a wide range of users, both with technical and non-technical backgrounds, such as engineers and scientific researchers or clinicians and teachers, respectively.

4.1.2. Aim of the user manual

The primary purpose of this user manual is to provide guidance on how to navigate and operate the Central Control Software (CCS). This document will serve as a reference to the users to comprehend the features of the system and troubleshoot common issues.

4.2. Installation instructions

4.2.1. Installation

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

The application can be run both with Visual Studio Community IDE and with the executable file, which has to be installed first. The application installer is located in the project folder and can be found with the following path:

CentralControlSoftware > *CentralControlSoftware* > *bin* > *Release* > *CentralControlSoftware.application*

The same folder also contains the executable file *CentralControlSoftware.exe*, which can be run after the installation is completed the first time.

4.2.2. Requirements

The following requirements for the operating machine are needed:

- Processor: ARM64 or x64 processor, Quad-core or better (recommended).
- RAM: 4 GB (minimum), 8 GB (recommended).
- Disk space: minimum of 850 MB up to 210 GB of available space for the installation of Visual Studio, depending on the features installed.
- Operating System (OS): at least Windows 10 (64-bit) is recommended.

The system is built on .NET Framework 4.8 and its system requirements can be found here.

Depending on the modules that are integrated in the system, additional hardware or software requirements might be needed. Some modules included in the version of software described in the present document are controlled by specific APIs which have the following dependencies:

- Python 3.9
  - cv2 (pip install opencv-python) (*AudioVideoRecorder.py*)
  - numpy (*AudioVideoRecorder.py*)
  - pyaudio (pip install PyAudio) (*AudioVideoRecorder.py*)
  - pyserial (*shimmer_API_python.py*)
- VLC (MultimediaPlayer)
  - Download VLC
  - Enable cmd control (View > Add Interface > Console)
- Miniconda (Bioharness)
  - Install miniconda3 with built-in Python version (Python 3.9)
- USB Bluetooth Dongle (XsensDOT)

4.2.3. Preliminary connections

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The latest version of the software integrates some devices that need a Bluetooth connection to operate and the first time they are used a pairing procedure is needed.

XsensDOT

The inertial measurement units (IMU) Xsens DOT (Xsens) were originally commercialized by the manufacturer with a web server API that can scan, connect and manage measurements and recordings with Xsens DOT on Windows, macOS and Raspberry Pi. The API is developed in Node.js and is run by the CCS in a dedicated process, but this requires that it is correctly installed and the full installation procedure can be found here. However, the API has been customized to be integrated within the CCS and the modified version is provided together with the software, therefore it is not needed to perform the full procedure but only the following steps from command line:

- Install Node.js
- Install node-gyp
- Install Zadig to setup WinUSB driver and convert USB dongle.

After the installation procedure has been performed, the only operation that will be required every time XsensDOT will be used is to plug in the USB dongle into the laptop.

Bioharness

The device must be paired with the laptop during the first Bluetooth connection. To this aim, the device must be turned on, then connected to, by accessing to Bluetooth settings:

Start button > Settings > Bluetooth and devices > Add new device > Bluetooth

Once the device is selected and the pairing requested, the user will be asked to enter a PIN code (by default: 1234). The pairing procedure has to be operated only the first time, then it is sufficient to turn the device on and let the CCS manage the connection through module's API. In particular, Bioharness MAC address is needed to establish the connection. It can be found in Devices and Printer with the following path:

Start button > Settings > Bluetooth and devices > Devices > More devices and printer settings

Device properties has to be inspected by right-clicking on its tag and selecting Properties; in Bluetooth tab a list of properties is listed, among which an address in the form *a0:b1:c2:d3:e4:f5*. This address must be entered in the corresponding space in the CCS *Tab Wearable Sensors – Bioharness* (Figure 12).

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

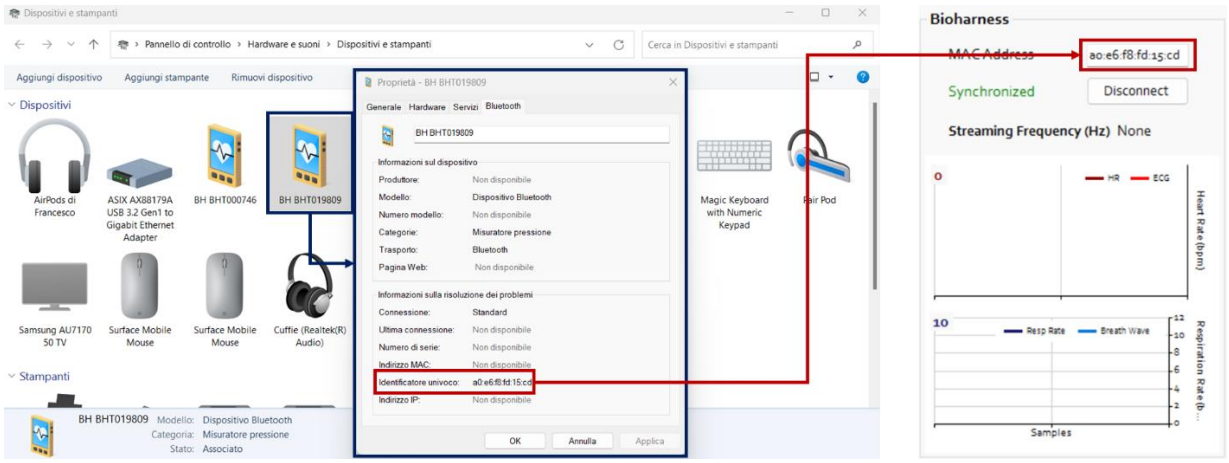

Figure 12: MAC address of the Bioharness

Shimmer

The device must be paired with the laptop during the first Bluetooth connection. To this aim, the device must be turned on, then connected to, by accessing the Bluetooth settings:

Start button > Settings > Bluetooth and devices > Add new device > Bluetooth

Once the devices is selected and the pairing requested, the user will be asked to enter a PIN code (by default: 1234). The pairing procedure has to be operated only the first time, then it is sufficient to turn the device on and let the CCS manage the connection through module's API. In particular, Shimmer COM port number is needed to establish the connection. It can be found in Devices and Printer with the following path:

Start button > Settings > Bluetooth and devices > Devices > More devices and printer settings

Device properties has to be inspected by right-clicking on its tag and selecting Properties; in Services tab a list of properties is listed, among which a *Serial Port COMx*. The number of the port *x* must be entered in the corresponding space in CCS *Tab Wearable Sensors – Shimmer* (Figure 13).

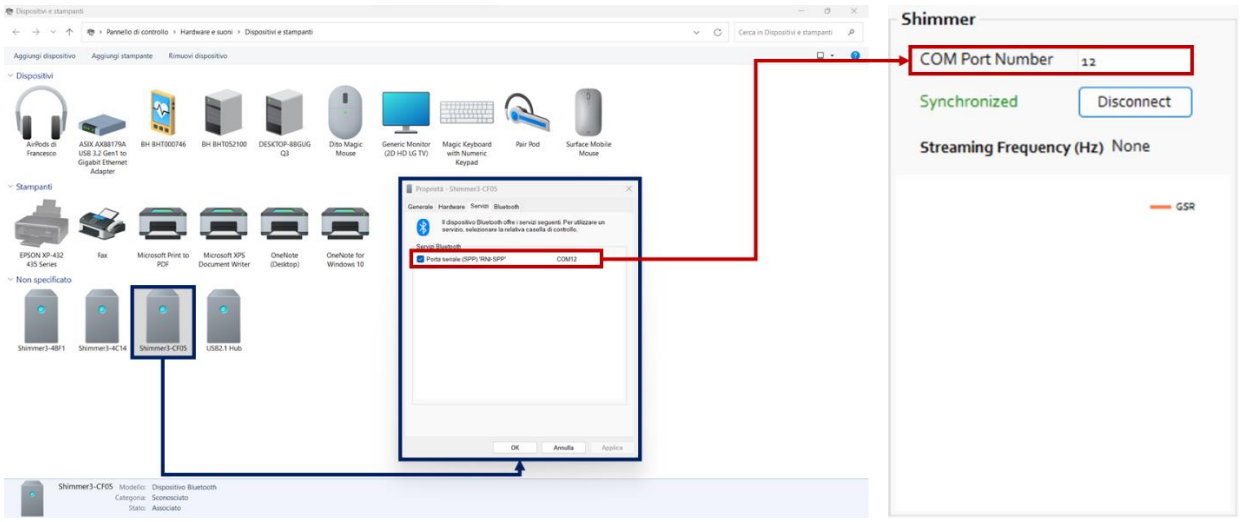

Figure 13: Shimmer serial COM port number

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

4.3. Graphical User Interface

4.3.1. Overview of the main interface

Main Menu

The *Main Menu* presented to the users upon launching the application (Figure 14) serves as a central hub from which they can navigate to different sections and holds the GUI principal components:

- *Commands Menu*
- *Recording Menu*
- *Modules Tabs*
- *Logo and Datetime*
- *Log Window*

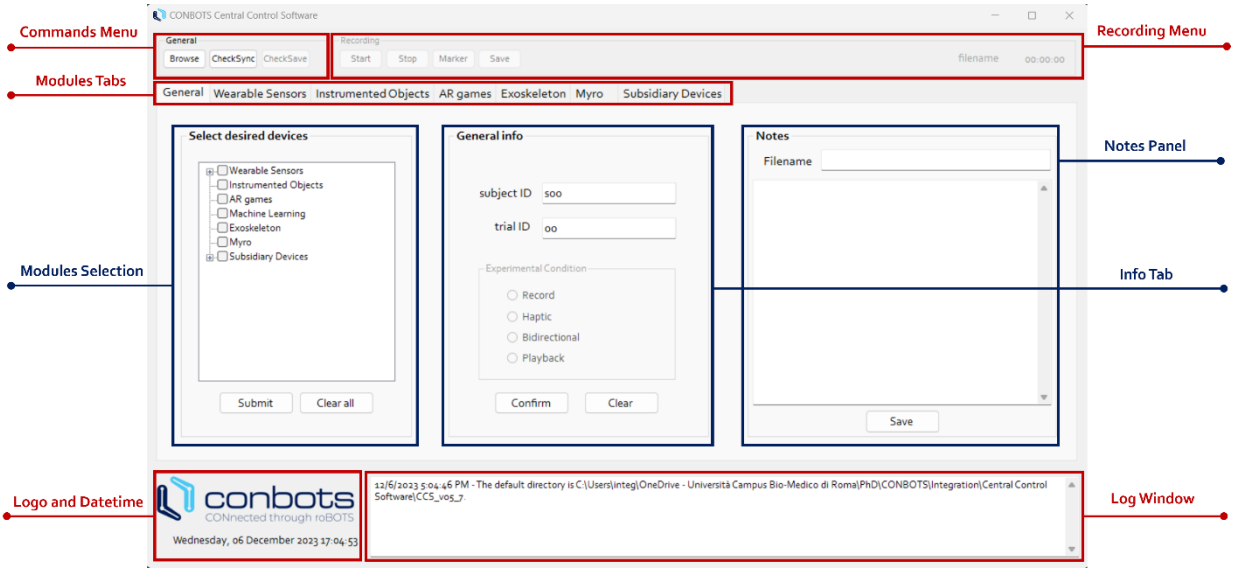

Figure 14: *Main Menu* of the CCS and its principal components. The sections highlighted in red are elements of the *Main Menu*, while the sections highlighted in blue are specific for *Tab General*, hence will not be visible when navigating in another Tabs.

The *Commands Menu* panel gathers the commands that the user is allowed to control and is always visible during the functioning of the system, to guarantee a real-time control of the platform. However, it is initially disabled because the user is not allowed to start or stop a recording if at least one module has not been previously selected, connected to and synchronized. The button *Browse* enables the user to select the current working directory, i.e., the directory in which the software will look for libraries and APIs and will automatically save the files generated at the end of each recording for every device. The button *CheckSync* can be pressed to check that all selected modules are correctly synchronized and it is particularly useful before starting recording when the user wants to verify that the modules are synchronized but does not want to navigate through each of the *Modules Tabs* to verify this information individually. When the button is pressed, a message box will appear showing the list of the modules that were originally selected and a Boolean value showing if each of the modules is synchronized (*True*) or not (*False*), as depicted in Figure 15.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

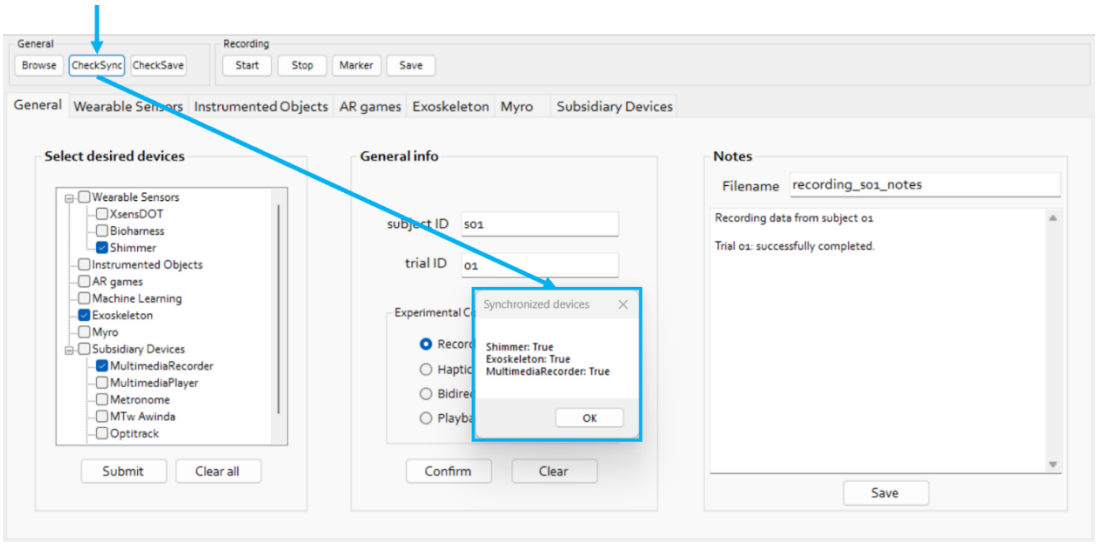

Figure 15: Button *CheckSync* and visualization of the list of synchronized modules.

Similarly, *CheckSave* button can be pressed when the user has already stopped the recording and wants to check whether the data recorded by each of the connected and synchronized modules were correctly stored by the CCS. Pressing the button will prompt a message box that shows the list of connected modules and a nullable Boolean value that confirms if data were saved (*True*) or not (*False*), or if the module was not enabled to save data (*null*) (Figure 16).

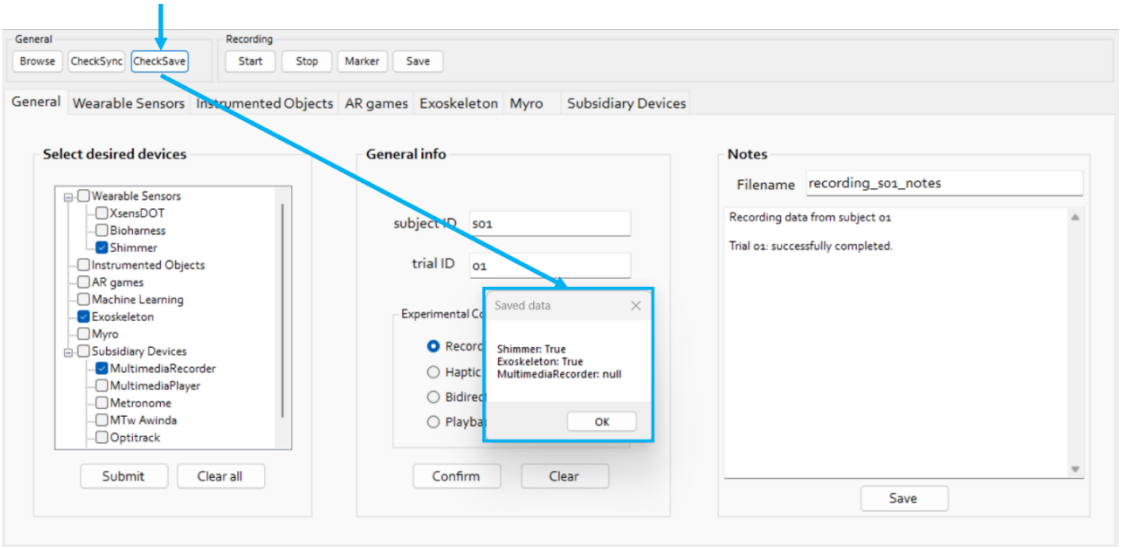

Figure 16: Button *CheckSave* and visualization of data saving confirmation list.

The upper-right corner of the *Main Menu* holds the *Recording Menu*, a panel dedicated to the management of the recordings. The buttons *Start* and *Stop* allow to send to the connected modules a start and stop recording command, respectively. The former will only be enabled once all the selected modules are connected and synchronized and the latter will automatically save the collected data in a dedicated .txt file for each module. The button *Marker* allows to set a temporal flag to mark the occurrence sample of a relevant event with an increasing counter. Finally, the button *Save* (Figure 17) can be pressed when the recording has been stopped to manually save recorded data, in case the

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

message prompted by *CheckSave* showed that data from a specific module has not been saved. Pressing this button will only save this data and will not overwrite what was originally saved from other modules.

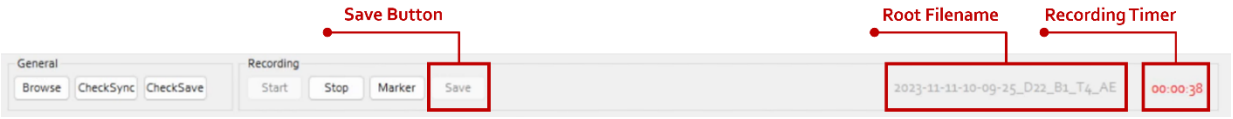

Figure 17: Additional options embedded in the *Recording Menu*.

Whether data saving is executed automatically by the CCS or manually by the user, the system will manage data storage in such a way that every module used during a recording will have its own file, whose name will consist of a common segment representing the timestamp of the recording start, the subject ID, and the trial ID entered in *Tab General*, in addition to the name of the corresponding module. The user can observe the common segment, namely *Root Filename*, regardless of the active tab. A *Recording Timer* is also included in the interface for real-time monitoring of the recording duration (Figure 17), beginning when the *Start* button is clicked and stopping when the *Stop* button is clicked.

The *Main Menu* holds the *Modules Tabs*, a series of tabs dedicated to different modules or class of modules. The user can always navigate through each of these tabs, but only the options corresponding to the selected modules will be enabled and can be modified. Besides the specific properties of the modules, each of them has a dedicated panel with a real-time changing label that specifies its status with the following descriptors:

- *Not connected*: the device has not been connected yet
- *Not synchronized*: the device is connected but not synchronized
- *Synchronizing*: the device is connected and synchronizing
- *Synchronized*: the device is connected and synchronized

Another common functionality is the button *Connect/Disconnect*, which allows the user to connect or disconnect to the corresponding module, respectively. On the other hand, some modules might require additional information to finalize the connection, such as a COM port number, and be equipped with real-time charts to show streamed data.

The default tab prompted to the user is *Tab General* and it holds the following three sections:

- *Modules Selection*: this panel allows the user to inspect the list of all possible modules and select the ones intended for use. Some modules are grouped in classes, such as XsensDOT, Shimmer and Bioharness (Wearable Sensors class), and they can be selected simultaneously, by checking the box with the name of the class, or individually, by unfolding the class menu and selecting the desired ones. In general, any combination of modules can be selected by checking the corresponding boxes. Once all desired modules have been selected, the user can confirm the choice by pressing the button *Submit*, which will automatically enable the sections dedicated to each of the selected modules. On the other hand, if the user wants to clear the boxes and repeat the selection, the button *Clear all* can be pressed.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

- *Info Tab*: this panel allows the user to enter information related to the experimental session, such as the names or IDs of the subject and the experimental trial (). For some of the available devices, it also possible to specify the experimental condition among some options (Record, Haptic, Bidirectional, Playback). Indeed some modules, such as the Exoskeleton, can be used in different modalities according to the specific experimental protocol carried out and the selection of the corresponding radio button will trigger the module accordingly. In particular, the available options are:
    - *Record*: one or two exoskeletons (E1 or E1 and E2) are used in transparent mode to record trajectories.
    - *Haptic*: two exoskeletons are used, one in transparent mode (E1) and one in haptic mode (E2) with the reference trajectory represented by E1 recorded trajectory.
    - *Bidirectional*: two exoskeletons are used (E1 and E2) in haptic mode with the reference trajectories estimated from the relative error between E1 and E2.
    - *Playback*: one exoskeleton (E1) is used in haptic mode with pre-recorded trajectories.

  Some other modules, such as Multimedia Recorder or Multimedia Player, can be used only in specific conditions: for instance, Multimedia Recorder will be selected any time a video has to be recorded (*Record*, *Haptic*, *Bidirectional*), while Multimedia Player will be included whenever a reference video has to be shown to the user (*Playback*).
  Finally, entered options can be confirmed by pressing the button *Confirm* or reset by pressing the button *Clear all*.
- *Notes Panel*: this panel provides a notepad to keep track of noteworthy events occurring during an experimental session and to save the notes in a .txt file with the filename entered in the corresponding section by pressing the button *Save* (Figure 18c). If the button is not pressed during the recording session, the CCS will automatically save the notes in a dedicated file, with the specified filename or, if not specified, with a default filename.

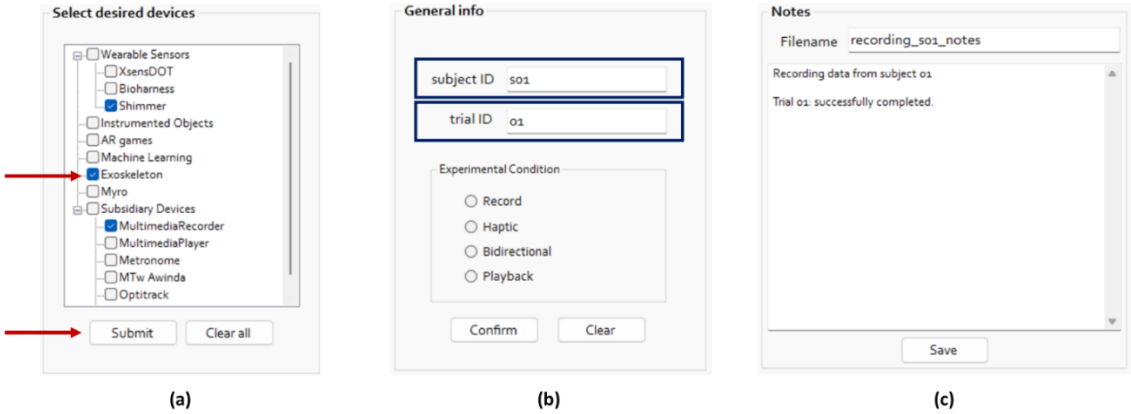

Figure 18: Tab General.

Finally, *Logo and Datetime* section holds the logo of the project and the current date and time, while *Log Window* shows the logs generated by the software during its execution.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

Modules Tabs

Tab Wearable Sensors

The *Wearable Sensors* tab (Figure 19) holds the control panels of the wearable sensors, i.e., XsensDOT, Bioharness and Shimmer. Moreover, an additional Machine Learning panel has been added to show the machine-learning-based classification of participant's emotional state according to kinematic and physiological data collected by the set of wearable sensors.

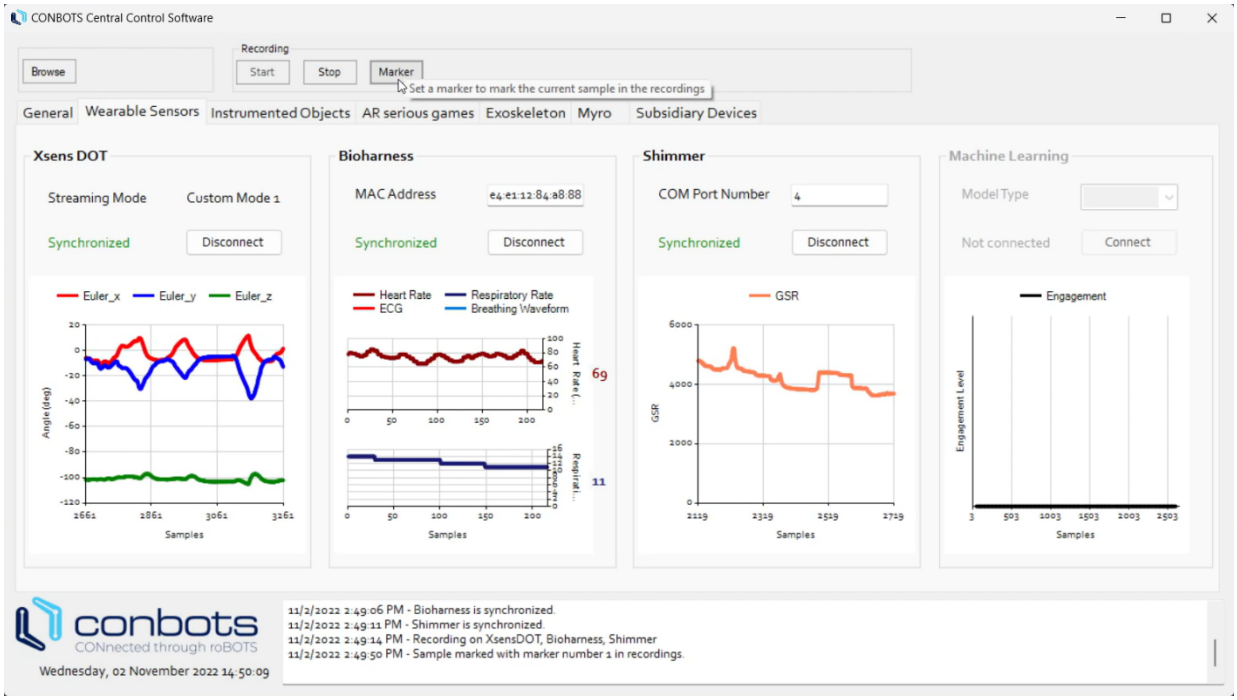

Figure 19: Tab *Wearable Sensors*.

Each panel has a label that specifies the real-time status of the corresponding sensor, as well as the *Connect/Disconnect* button. In particular, the *Xsens DOT* panel shows the streaming mode that the user should select in the Xsens DOT API and a real-time chart shows the incoming kinematic data recorded by the sensors (Euler angles along x, y and z directions). The *Bioharness* panel requests the user to enter the MAC address of the device and shows in a two-plots live chart the measured heart rate/ECG and respiratory rate/breathing waveform. The *Shimmer* panel has an input box for the COM port number to which the device is connected and a chart showing GSR data. Finally, the *Machine Learning* panel has a combo box identified by the label "*Model Type*" that allows the user to select the specific model to use to determine subject's status from the wearable sensors data among the following options:

- Global Model
- Motion Model
- Biometric Model
- Wrist Motion Model

The real-time estimation of user's status is shown in the graph included in the panel, where 0 corresponds to user disengaged and 1 to user engaged.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

Tab Instrumented Objects

The tab *Instrumented Objects* is equipped with a status panel that allows to check the real-time status of the connection with the smart objects and to connect or disconnect them; moreover, a live chart shows the incoming measured data, such as the acceleration along x, y and z axes of the violin (Figure 20).

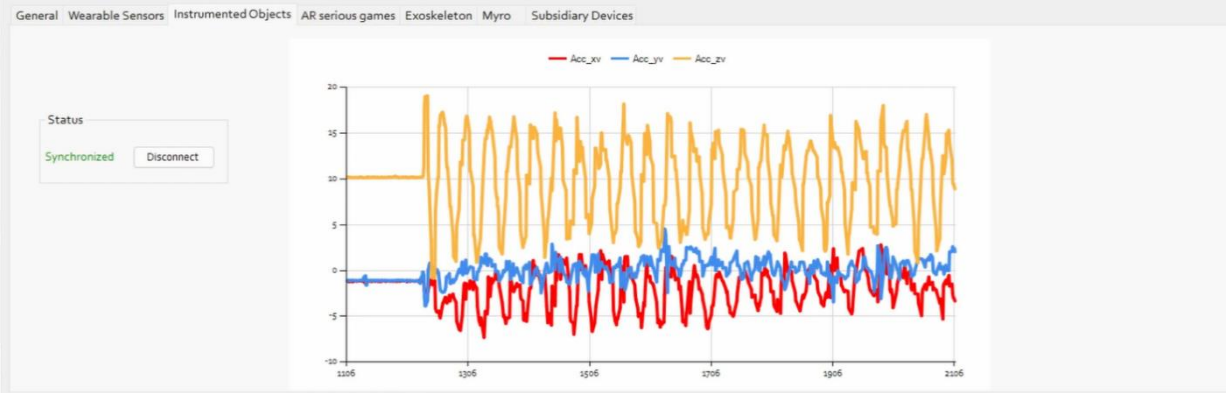

Figure 20: Tab *Instrumented Objects*.

Tab AR games

The *AR games* tab currently includes a status panel that gathers a live-status label and a *Connect/Disconnect* button (Figure 21).

Figure 21: Tab *AR games*.

Tab Exoskeleton

The tab *Exoskeleton* provides the user with an interface to manage and check the status of the exoskeleton or exoskeletons connected, and it is composed of two elements: *Exoskeleton Status Panel* and *Exoskeleton Chart Area* (Figure 22). The former reports the selected experimental condition

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

("Bidirectional"), the real-time estimated streaming frequency, the current status of the exoskeletons ("Connected") and the current functioning mode of the devices ("Haptic"), while the latter shows received data from one or two exoskeletons according to the selected experimental conditions. In particular, the chart area is divided into a 2x2 matrix where the first row shows the kinematic data and the second row shows the kinetic data for both joints; on the other hand, the first column refers to the shoulder modules, while the second column refers to the elbow modules. This arrangement allows a real-time comparison of the behaviour of both exoskeletons for each joint in terms of measured joint angles and measured or desired torque. In particular, for both exoskeletons the following data are visible in the charts:

- *Shoulder Measured Active*: active DoF from the shoulder (internal/external rotation)
- *Shoulder Measured Passive*: passive DoF from the shoulder (flexion/extension)
- *Elbow Measured Active*: active DoF from the elbow (flexion/extension)
- *Shoulder Measured*: measured torque from the shoulder module
- *Shoulder Desired*: desired torque from the shoulder module
- *Elbow Measured*: measured torque from the elbow module
- *Elbow Desired*: desired torque from the elbow module

Figure 22: *Tab Exoskeleton*.

Finally, the platform is already prepared to allow the user to manually change the impedance of the exoskeletons and to set it to a predefined low or high value, by pressing the buttons *Low* and *High*, respectively.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

Tab Myro

The tab *Myro* currently includes a status panel that gathers a live-status label and a *Connect/Disconnect* button (Figure 23).

Figure 23: *Tab Myro*.

Tab Subsidiary Devices

The tab *Subsidiary Devices* is dedicated to the management of those additional devices that have been used during the experiments and were needed to be synchronized with other integrated modules (Figure 24).

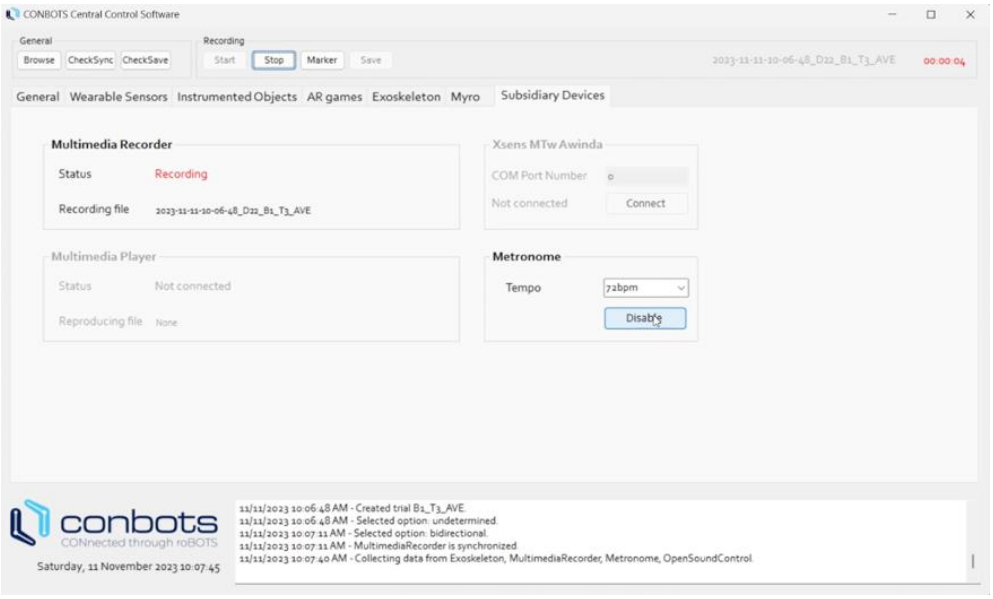

Figure 24: Tab *Subsidiary Devices*.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)

The panels dedicated to the Multimedia Recorder and Multimedia Player show the status of these devices as well as the names of the files that are currently being recorded (Multimedia Recorder) or reproduced (Multimedia Player). If Multimedia Recorder is selected and a valid experimental condition is set (Record, Haptic or Bidirectional), the software will start a process running the API that controls the recording device. The API connects by default to the built-in webcam of the laptop, but can be customized to connect to any external webcam by simply specifying the name of the device. In any case, a window will be prompted to the user showing the frames captured by the recording device. On the other hand, if the user selects Multimedia Player and the experimental condition Playback, a dialog window will appear asking the user to select the multimedia file that has to be reproduced.

The panels dedicated to Xsens MTw Awinda sensors allows the user to enter the serial COM port number to which the Awinda station is connected and to connect to the module by pressing the button *Connect*.

The panel dedicated to Metronome holds a combo box with different tempi (60, 72, 76, 80, 86, 100, 120 bpm) and an *Enable/Disable* button to manually activate or deactivate the metronome. Once a desired tempo has been selected, the metronome automatically starts when the button *Start* recording is pressed.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N. 723386

The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s)