Peltihamsteri project

Mari Kasanen
Leevi Liimatainen
Marina Mustonen
Juhani Sundell
Arttu Ylä-Sahra

Technical manual

Syncster 1.0.0 & Touchster 1.5

Version 0.2.0
Public
5.6.2019

University of Jyväskylä

Faculty of Information Technology

Contents

1 Development environment

In the chapter we give basic instructions on setting up development environments for Syncster and Touchster.

1.1 Setting up a development environment for Syncster

Following software are required for developing Syncster:

1. Visual Studio 2017 Community is sufficient. For improved testing facilities, it is recommended to use Visual Studio 2017 Enterprise.
2. .NET Framework version 4.6.1 (or higher) must be available on the system.

After the prerequisites have been met, a project can be opened simply by opening `Ajolabra.sln`. It consists of four projects:

- `ALGUI` is the entry point, and it includes the main user interface.
- `ALManager` is the manager who acts between ALGUI and ALBackend. It manages the backend components and provides a simplified interface to the GUI.
- `ALBackend` is the backend containing device-specific modules and abstract pipeline components.
- `ALUnitTests` contains unit tests for all other projects, separated from the actual code.

For instructions on how to connect all supported devices to Syncster, see the instruction manual of Syncster (`Syncster_instruction_manual.pdf`).

1.2 Setting up a development environment for Touchster

For Android development, you will need Visual Studio 2017 and Visual Studio Tools for Xamarin. Xamarin's versions to be used are as follows:

Xamarin	4.12.3.80,
Xamarin Designer	4.6.13,
Xamarin Templates	1.1.128, and
Xamarin.Android SDK	9.1.7.0.

In addition, you will either need an Android device for testing, or an Android emulator if you do not have such a device. Once these prerequisites have been met, open `AndroidTouch.sln`.

2 Installing Syncster and Touchster

In the chapter we outline the process of compiling a release version of Syncster and making an apk-file of Touchster.

2.1 Instructions for installing Syncster

To make a release version for a 64 bit PC with Windows 10, carry out the following steps:

1) Open `Ajolabra.sln` in Visual Studio 2017. Make sure the ALGUI project is set as Startup Project.
2) Select *Project* → *ALGUI Properties…* In the *Application* tab, you can change the *Assembly name* (currently `Syncster`), *Target framework* (currently `.NET Framework 4.6.1`) and *Output type* (currently `Windows Application`), if needed. Please let `hamsterIcon.ico` stay as the project's icon, though.
3) Select `Solution 'Ajolabra'` in *Solution Explorer*.
4) Select *Build* → *Configuration manager*.
    a. Change *Active solution configuration* to `Release`. Each project's *Configuration* should automatically change to `Release`.
    b. Change *Active solution platform* to `x64`. Each project's *Platform* should automatically change to `x64`.
    c. Click *Close*.
5) Select *Build* → *Rebuild solution*.
6) Now there should be a working release version of Syncster inside the folder `Ajolabra\ALGUI\bin\x64\Release` and it should contain the following files:
    a. `ALBackend.dll`,
    b. `ALBackend.pdb`,
    c. `ALManager.dll`,
    d. `ALManager.pdb`,
    e. `Syncster.exe`,
    f. `Syncster.exe.config`,
    g. `Syncster.pdb`, and
    h. `Syncster_instruction_manual.pdf`.
7) Copying the `Release` folder to the desired location on to a PC is enough to install Syncster.

2.2 Instructions for installing Touchster

To make a Touchster apk file and install it, carry out the following steps:

1) Open `AndroidTouch.sln` in Visual Studio 2017 (with Xamarin Tools installed).
2) Pick `Release` from *Solution Configurations*.
3) Open the project's *Properties* from *Solution Explorer*.
    a. In `Android Manifest`:
        i. Change *Application name*, if needed.
        ii. Give new *Application icon*, if needed.
        iii. Make version number higher than before (for example old being `1.2`, new could be `1.3` or `1.21`).
    b. In `Android Options`:
        i. Deactivate debugging (should already be off).
        ii. Under *Linking*, select *Select Sdk Assemblies Only* (makes app's size smaller).
4) Select *Build* → *Rebuild Solution*.
5) From *Solution Explorer* right click your project and select *Archive*. Visual Studio will show an indeterminate progress bar.
6) When Visual Studio is ready:
    a. Select the version you created.
    b. From its options, select *Distribute*.
    c. Select *Ad Hoc*.
    d. In the *Signing Identity* window:
        i. Select the *touchster* key (if you can't see it, you need to Import it).
        ii. Select *Save As*.
        iii. Save the apk file in your preferred file location. You can also rename it.

iv. After this you need to give the password of the keystore, which is `touchster`.

7) Connect your phone to your PC and copy your new apk file in it.
8) Find the apk file from the directory of your phone and tap to install it.

If there are problems when installing the apk file, you can try the following methods:

- Play Protect may be stopping the install process. To deactivate it, go to the settings of Google Play Store and select *Play Protect*. Deactivating Play Protect may allow you to install Touchster.
- If you have an older version of Touchster on your phone, a new version might not be installed. This occurred once when different development versions were tested. To install a new version, first remove the old version and then try installing again.

3 Adding modules to Syncster

In the chapter we outline steps that are required to take in order to add a new device module to Syncster. We assume the reader wants to implement a fictional module for a device called `Example`.

These instructions are only applicable for modules that receive data in real time. Syncster currently has very little underlying infrastructure to support components that import data after recording has concluded, so adding such components requires more extensive development. `Importer` and `ETFileReader` provide an example implementation of such a component.

Syncster device modules, by pattern, need at least two discrete classes: a reader and a wrapper. `DSWrapper` shows an example of how to use more than two classes, and a more complicated structure. However, in the following we share a basic knowledge of a reader class and a wrapper class, followed by instructions on how to bind the module to `Manager` and `UI`.

3.1 ExampleReader

Each module has an `AbstractPipelineComponent` implementation, a so-called reader. Each pipeline component operates in its own thread, looping requesting data until stopped with a cancellation token.

In detail, a reader has the following responsibilities:

- It delivers data using `IngestNewData()`, by returning a new row of data at arbitrary intervals.
- An implementation should be able to react to a `CancellationToken` activating, and stopping gracefully when requested.
- A single row consists of arbitrarily named keys and `IDataElements`. These keys should remain more or less consistent over time, and need not exist on every row. No key is allowed to have a null value. Do note that if you utilize certain subcomponents, they may have their own requirements. See `CSVRawLineSplitter` for example.
- It implements necessary preparation and shutdown steps where required. Some components may need to do certain discrete actions before they are started or stopped completely. This is implemented with `NeedsPreparation`, `PrepareInternal` and `ShutdownInternal`. Do note that the implementation should also be somewhat resilient to exceptional stops. Even if the component stops to a fault, `ShutdownInternal` will still be called.
- It manages its own subcomponents, starting, stopping and handling exceptions and messages where necessary. Not all components are required to contain subcomponents, but if they exist, it is required that the enclosing component monitors and gracefully handles any situations that may occur. Examples of this pattern are found from `DSReader` and `ETReader`.
- It provides log messages using `OnLogMessage` and `OnDebugMessage`. The former should be only used for messages that the average user would find important

for them, whilst the latter can be used for more diagnostic, debugging-oriented messages.

- (Optional) It implements `PauseInternal` and `UnpauseInternal`. Originally, the components supported a separate pause state, in which the components would otherwise be running, but their data would be discarded. Intention was that the components could do certain optimizations with the knowledge that data is discarded, but this was never widely used. As such, it is declared obsolete, and in general is not a required part of a component, but it is good to be aware of the feature.
- (Optional) It implements IDisposable. This is not strictly required from components, but may be useful to implement if the component uses any resources that should be manually released.

There are multiple approaches for almost all of the responsibilities mentioned above. It is recommended to study existing components and from there, select the approaches best fitting your needs.

3.2 ExampleWrapper

Each device will also need its own `IModuleWrapper` implementation. More precise technical definitions are located in `IModuleWrapper.cs`, but rough responsibilities are as follows:

- It implements `Connect()` and `Disconnect()`, which respectively start and stop device components upon request. The device components are connected before an actual recording starts.
- It implements `StartRecording()` and `StopRecording()`. These typically add and remove the wrapper's internal `BlockingCollection` to/from the appropriate component output queue, but other patterns may be possible as well.

- It implements `DiscardData()`, which is expected to empty the contents of captured data and reset other state where appropriate.
- It implements `IObservable<PipelineStatus>`. This is commonly delegated to the outermost device component, as it implements this same interface.
- It implements `IPipelineEvents`. This is also commonly delegated to the device component, as this essentially requires log and heartbeat events.
- It implements `IDataCollector`. This contains the following methods and properties:
  - `ModuleName` is a human-readable name of the module which must be usable as a part of a file name. It is used as a prefix for the module's data columns in the final output file, so a ModuleName should be recognizable and brief (current modules have two to three letter long ModuleNames).
  - `CollectedData` shall return a copy of the collected data so far.
  - `UnprefixedStickyKeys` declares certain sticky keys (that print out on every row in the output file), if the module has them.
  - `SynchronizationCanonicalTimeDeterminer` is a canonical time determiner. A time determiner takes a single row, and returns a `DETimestamp` object signifying the canonical time for that row. Canonical time in the context of module wrappers has been declared to be the absolute time in relation to the event creating that row.
- (Optional) It implements `IDisposable`. If a module is `IDisposable`, `Dispose()` will be called when the module is about to go permanently out of scope, allowing the module to release any manually releaseable resources it holds.

3.3 Binding to Manager

In order to make your module usable from the `UI`, it first needs to be bound to the `Manager`. To do this, follow these instructions:

- Add support to `ModuleFactory`, allowing the module to be instantiated on demand. This means adding the constructor of the module to the switch-case of the `GetModuleWrapper()` method and providing the needed settings as parameters.
- Each module should have a unique name. If modules have duplicate names, only one of them will be active during recording.
- If the data collected by the module contains keys that can be filtered by the user, store the keys that are not selected to the `_blacklistedKeys` variable when Manager's `ConnectAsync()` method is called.
- (Optional) Add a new comparer to `ColumnHeaderComparer` to determine the order of columns in the final output file. The comparer should inherit the `AbstractColumnComparer` base class and override its `ColumnOrder` property. For more information, see how other comparers (for example `ETComparer` or `EEGComparer`) have been implemented. If no comparer is defined, columns will be arranged in alphabetical order.

It should also be noted, that the `Manager` assumes that all modules execute their actions, such as connecting and disconnecting, without blocking or taking too much time. Any single module that behaves differently causes indefinite waiting times, preventing further interactions with other modules as well.

3.4 Binding to UI

In order to show the settings and state of your module to the user, follow these instructions:

- Create appropriate UI elements in `MainWindow.xaml` for the new component, and dependency properties for them in `MainWindow.xaml.cs`. Default values of dependency properties have been used as a convenient way of providing information to the user.

- The component needs its own device tab inside the *tabctrl_devices* tab control. Please take a look at the other device tabs for guidance.
- Inside the *Record* tab's GroupBox with the header *Device names, heartbeats, and statuses*, create a new grid row for your module. It should consist of a `border`, two `labels`, and an `ellipse`. See the other rows for example. The latter label and the ellipse should be properly named, because they will be used in the state visualization of your module.
- Edit `SerializableSettings` so that the component properties are appropriately bindable to the UI and can be passed onwards.
- Add the information of the component to `CreateSettings()` and `ShowSettings()` to ensure data flow for both saving and loading settings.
- If the component has selectable data (like DS and EEG), you should implement a blacklist method in `SerializableSettings` that is called from Manager's `ConnectAsync()`. DS and EEG have very different implementations of blacklisting which are hopefully useful as guides.
- In `MainWindow.xaml.cs`, you should also listen to your module's events. Assuming your module is properly connected to `Manager`, this is already taken care of in MainWindow's constructor.
- In `MainWindow.xaml.cs` inside the code region *Backend event handlers*, add your module's information to the following methods: `GetStatusEllipse`, `GetStatusBrush`, and `_manager_HeartbeatUpdate`, in order to visualize `ExampleReader`'s state and heartbeat.