BRAUDE
College of Engineering, karmiel

TESTING SYSTEM MAINTENANCE GUIDE

BRAUDE
College of Engineering, karmiel

1. Maintenance Guide 2
1.1. Testing Station App 2
1.1.1. Installing and Running the Application 2
1.2. Tested OBC App 3
1.2.1. Installing and Running the Application 3
1.3. Error Detection and Debugging 4
1.3.1. Log Files 4
1.3.2. Local Simulation 4
1.4. Implementing Changes 4
1.4.1. Additional Communication Types 4
1.4.2. Additional Algorithms Variations 5
1.4.3. Additional Test Creation Options 5
1.4.4. Testing Different Algorithms Types 5
1.4.5. Additional GUI Features 5

1. Maintenance Guide

The Tested OBC application, the on-board-computer side and the Testing Station application were successfully built and tested on the following platforms:

- Windows 10/11 (x64)
- macOS (Intel-based, Apple Silicon builds are also possible with minor adjustments)
- Raspberry Pi 5 (running a standard 64-bit Linux distribution such as Raspberry Pi OS 64-bit)

The Tested OBC and Testing Station applications rely on:

- C++17 or newer
- CMake (3.15+) on Linux/macOS
- Visual Studio 2022 on Windows (or MSVC from the command line)
- Optionally: Command-line builds for Raspberry Pi 5
- Optionally: CLion for remote development on Raspberry Pi 5

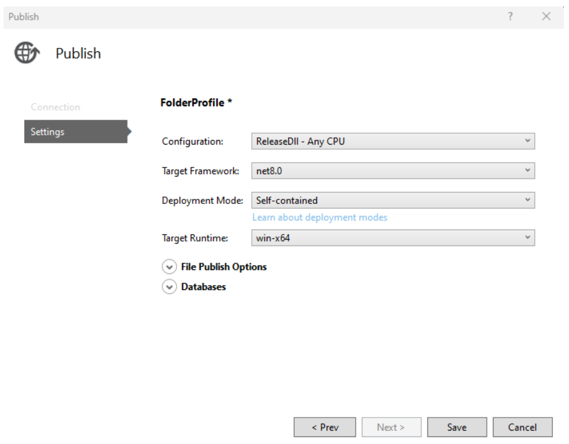
Below are the detailed instructions for building and installing on each environment.

1.1. Testing Station App

1.1.1. Installing and Running the Application

The Testing Station on Windows can be run immediately using the precompiled Blazor executable found in the ReleasedVersions folder. Double-clicking the executable will launch the GUI in your default browser, displaying real-time status indicators for on-board computer connectivity. To rebuild the application, you can use Visual Studio and Publish the blazor project with the existing settings for ReleaseDll – ANY CPU, while choosing the deployment mode as "Self-contained" and Target Runtime as "win-x64". Once compiled, ensure that the TestingStationAppSettings.ini file is in the same folder as the executable so that TCP or UDP ports, IP addresses, and logging parameters are correctly configured.In addition, if there is no Logger folder present, create a new folder named Logger in the same location where the INI file and the executable are extracted. All log files and results files will be saved in this Logger folder.

Image 1: Visual Studio for Publish the blazor project with the existing settings for ReleaseDll – ANY CPU



FolderProfile *

Configuration: ReleaseDll - Any CPU

Target Framework: net8.0

Deployment Mode: Self-contained

Learn about deployment modes

Target Runtime: win-x64

File Publish Options

Databases

1.2. Tested OBC App

1.2.1. Installing and Running the Application

1.2.1.1. Running on Windows

On Windows, the Tested OBC application is distributed as a precompiled .exe located in the ReleasedVersions folder. Simply place it alongside the TestedObcAppSettings.ini configuration file, then double click the .exe or run it from the command line. If the configuration file specifies TCP, the OBC app will attempt to connect to your Testing Station's IP and port, for UDP, ensure ports are open as needed. To compile the application for windows you can either compile a released version via Visual Studio or use the CMakeLists file located at Code\TestedOBCApp\TestedOBCAppCMake` and build a new windows version. After compilation, copy the resulting .exe and .ini files together to run in a single folder. In addition, if there is no Logger folder present in that directory, create a new folder named Logger in the same location where the INI and EXE files are located. All log and result files will be stored in this Logger folder.

1.2.1.2. Running on MacOS

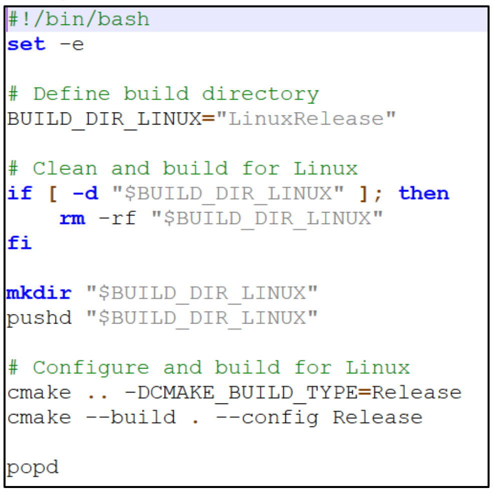
To run the Tested OBC on macOS, clone or copy the TestedOBCApp source folder onto your system. From a terminal, navigate to the root of the CMake project and create a build directory (for example, mkdir macOSRelease && cd macOSRelease). Run cmake .. -DCMAKE_BUILD_TYPE=Release, then cmake --build . to compile the application. Alternatively, you can open the project in CLion, select a Release configuration, and let the IDE handle the CMake workflow automatically. When the build completes, place the generated binary (TestedOBCApp) in the same directory as the TestedObcAppSettings.ini file.If there is no Logger folder present in that directory, create a new folder named Logger in the same location where the INI file is located. All log and result files will be stored in this Logger folder. Finally, Launch the app by calling ./TestedOBCApp from a terminal, and it will attempt to connect to the configured Testing Station or perform a local simulation depending on the .ini settings.

1.2.1.3. Running on Linux (Raspberry Pi 5)

On Linux, you can build the Tested OBC application using the provided `build_Linux.sh` script or use the `CMakeLists` file to build the project from the command line. If using the script, mark it executable (chmod +x build_linux.sh) and run ./build_linux.sh to generate a Release build in a dedicated folder, such as LinuxRelease. Alternatively, you can use CLion Remote Development to configure a remote toolchain for the Raspberry Pi 5, build the project, and deploy it directly on the board. Place the TestedObcAppSettings.ini file in the same folder, adjusting IP addresses and ports to match your Testing Station or local simulation preference. If there is no Logger folder present in that directory, create a new folder named Logger in the same location where the INI file is located, all log files and result files will be stored in this Logger folder. Finally, start the application by executing

./TestedOBCApp from the terminal. The console output and logs will confirm successful initialization, communication attempts, or any errors encountered.

Image 2: The build Linux script, simply building with CMake in release.



```
#!/bin/bash
set -e

# Define build directory
BUILD_DIR_LINUX="LinuxRelease"

# Clean and build for Linux
if [ -d "$BUILD_DIR_LINUX" ]; then
    rm -rf "$BUILD_DIR_LINUX"
fi

mkdir "$BUILD_DIR_LINUX"
pushd "$BUILD_DIR_LINUX"

# Configure and build for Linux
cmake .. -DCMAKE_BUILD_TYPE=Release
cmake --build . --config Release

popd
```

1.3. Error Detection and Debugging

1.3.1. Log Files

Our application has a few logging mechanisms, all log outputs should be created in the Logger folder. The first one is an Event Logger, logging system event like starting the system, configuration loaded, receiving messages and starting tests and errors, both the Testing Station and Tested OBC application use this logger. Additionally, both applications have a Result Logger, saving tests results, but with somewhat different format and implementation. All log files are created when the application starts and their names contain the creation time and date, creating a new file for every system run. The event logger can be used to identify problems with the system, like connection or communication errors.

1.3.2. Local Simulation

The local simulation is a great debugging option, with an asynchronized run and no dependencies and a single threaded application debugging is as easy as it gets. The only problem that can't be identified in the local simulation is a specific communication problem. You can run the local simulation mode in Visual Studio or CLion by making sure that the configuration INI file (found under the main filter in VS) is set to local simulation. For the Tested OBC App there are additional local simulation related parameters in there, like what algorithms to test and more.

1.4. Implementing Changes

We collected a few possible future needed updates and how to implement them easily.

1.4.1. Additional Communication Types

Although we think that the TCP communication should suffice, you might want to run the application on a very specific satellite's computer with limited access to communication and connection, and might need to use a different communication type (serial communication for example). We designed our application to allow adding such changes easily. All you need to do in order to add a new communication type is to implement the ICommChannel interface and add the new channel to the applications and the creation to the Factories.

The comm managers on both applications doesn't assume any protocol or the correctness of the input and can still work and identify errors even with a less reliable communication.

1.4.2. Additional Algorithms Variations

Adding a new algorithm can be done by adding the algorithm to the algorithms Enum (for test request and so on) and implementing the creation of the object in the Tested OBC App Factory and Test Manager. Adding a new root finding algorithm for **CATCH [5]** is the same, updating the relevant Enum and factory, while implementing the root finder interface.

1.4.3. Additional Test Creation Options

Adding additional options for the user can be quite easy, for example setting a set of tests at once instead of manually or running over a set of inputs and creating a test for each one. All we need to do is add an appropriate function to the Lab and Lab Wrapper classes and create the appropriate GUI.

1.4.4. Testing Different Algorithms Types

You can use our application as a base and add different tests, for example testing other required algorithms for satellites. To do so might be more complicated, we recommend adding a test type to the test request message header allowing you to parse the data required for your test by adding code instead of changing the full application.

1.4.5. Additional GUI Features

For creating additional features and make them available in the GUI level, for example adding an option for generating graph form tests results, you need to implement the feature in your code then add it to the Lab and Lab Wrapper to make it available in the GUI environment, there you simply import it in the LabInterop class and its available to use withing the GUI managers.