HELICS Documentation

Philip Top, Trevor Hardy, Ryan Mast, Dheepak Krishnamurthy, An

Sep 09, 2026

CONTENTS

1 HELICS Quick Start **3**
- 1.1 Install HELICS and the Python Language Binding . . . 3
- 1.2 Confirm installation . . . 3
- 1.3 Clone in the HELICS Examples Repository . . . 3
- 1.4 Navigate to the "Fundamental Default" example . . . 3
- 1.5 Run the Fundamental Default Example . . . 4
- 1.6 Next Steps . . . 5

2 User Guide **7**
- 2.1 Orientation . . . 7
- 2.2 HELICS Installation . . . 8
- 2.3 HELICS User Tutorial . . . 30
- 2.4 Examples . . . 140
- 2.5 Support . . . 284

3 Developer Guide **287**
- 3.1 Style Guide . . . 287
- 3.2 Generating SWIG extension . . . 288
- 3.3 Run tests . . . 289
- 3.4 Generating Documentation . . . 289
- 3.5 HELICS Benchmarks . . . 289
- 3.6 Continuous integration builds . . . 291
- 3.7 CI/CD Infrastructure . . . 292
- 3.8 Porting Guide: HELICS 2 to 3 . . . 292
- 3.9 Public API . . . 294
- 3.10 RoadMap . . . 296
- 3.11 HELICS Type Conversions . . . 297

4 References **303**
- 4.1 HELICS Publications . . . 303
- 4.2 API Reference . . . 306
- 4.3 Configuration Options Reference . . . 395
- 4.4 Tools with HELICS Support . . . 426
- 4.5 Built-In HELICS Apps . . . 428

5 Quick links **455**

Index **457**

This is the documentation for the Hierarchical Engine for Large-scale Infrastructure Co-Simulation (HELICS). HELICS is an open-source cyber-physical-energy co-simulation framework for energy systems, with a strong tie to the electric power system. Although HELICS was designed to support very-large-scale (10,000,000+ federates) co-simulations with off-the-shelf power-system, communication, market, and end-use tools; it has been built to provide a general-purpose, modular, highly-scalable co-simulation framework that runs cross-platform (Linux, Windows, and Mac OS X) and supports both event driven and time series simulation. It provides users a high-performance way for multiple individual simulation model "federates" from various domains to interact during execution–exchanging data as time advances–and create a larger co-simulation "federation" able to capture rich interactions. Written in modern C++ (C++20), HELICS provides a rich set of APIs for other languages including Python, C, Java, and MATLAB, and has native support within a growing number of energy simulation tools.

Brief History: HELICS began as the core software development of the Grid Modernization Laboratory Consortium (GMLC) project on integrated Transmission-Distribution-Communication simulation (TDC, GMLC project 1.4.15) supported by the U.S. Department of Energy's Offices of Electricity Delivery and Energy Reliability (OE) and Energy Efficiency and Renewable Energy (EERE). As such, its first use cases centered around modern electric power systems and that domain continues to be one of the core use cases. However, HELICS has since expanded into use for many other domains such as Natural Gas, Water, Weather, and Transportation and new use cases appear frequently. HELICS's layered, high-performance, co-simulation framework builds on the collective experience of multiple national labs.

Motivation: Energy systems and their associated information and communication technology systems are becoming increasingly intertwined. As a result, effectively designing, analyzing, and implementing modern energy systems increasingly relies on advanced modeling that simultaneously captures both the cyber and physical domains in combined simulations. It is designed to increase scalability and portability in modeling advanced features of highly integrated power system and cyber-physical energy systems.

General citation for HELICS:

T. Hardy, B. Palmintier, P. Top, D. Krishnamurthy and J. Fuller, "HELICS: A Co-Simulation Framework for Scalable Multi-Domain Modeling and Analysis," in IEEE Access, doi: 10.1109/ACCESS.2024.3363615, available at https://ieeexplore.ieee.org/document/10424422

CHAPTER

ONE

HELICS QUICK START

If you just want to get a HELICS co-simulation running on your local machine and to see how it works for yourself, this is the place to start. This Quick Start Guide will get HELICS installed along with the Python interface and the first fundamental example. You'll then be able to run it, see your results, and take a look at the code. And, of course, if at the end of all of that you're puzzled as to what you actually did, we have a *whole User Guide* to get you up to speed.

The commands below are terminal/command-line tools available on Windows, Linux, and macOS.

1.1 Install HELICS and the Python Language Binding

```
pip install 'helics[cli]'
```

The HELICS User Guide predominantly uses Python as, in our experience, Python is the *lingua franca* of the application-oriented (vs. computer science) computing world. The above command installs the Python language bindings for HELICS (allowing you to add `import helics` to any Python script) as well as a HELICS library.

1.2 Confirm installation

`helics --version` should return something reasonable-looking, namely a version number followed by unique identifier for the release.

1.3 Clone in the HELICS Examples Repository

```
git clone https://github.com/GMLC-TDC/HELICS-Examples.git
```

OR

Download a copy of the repository (if you're not familiar with Git.)

The HELICS Examples repository contains all the examples for the User Guide (and other example content as well). Cloning or downloading this repository will give you a local copy of all those examples.

1.4 Navigate to the "Fundamental Default" example

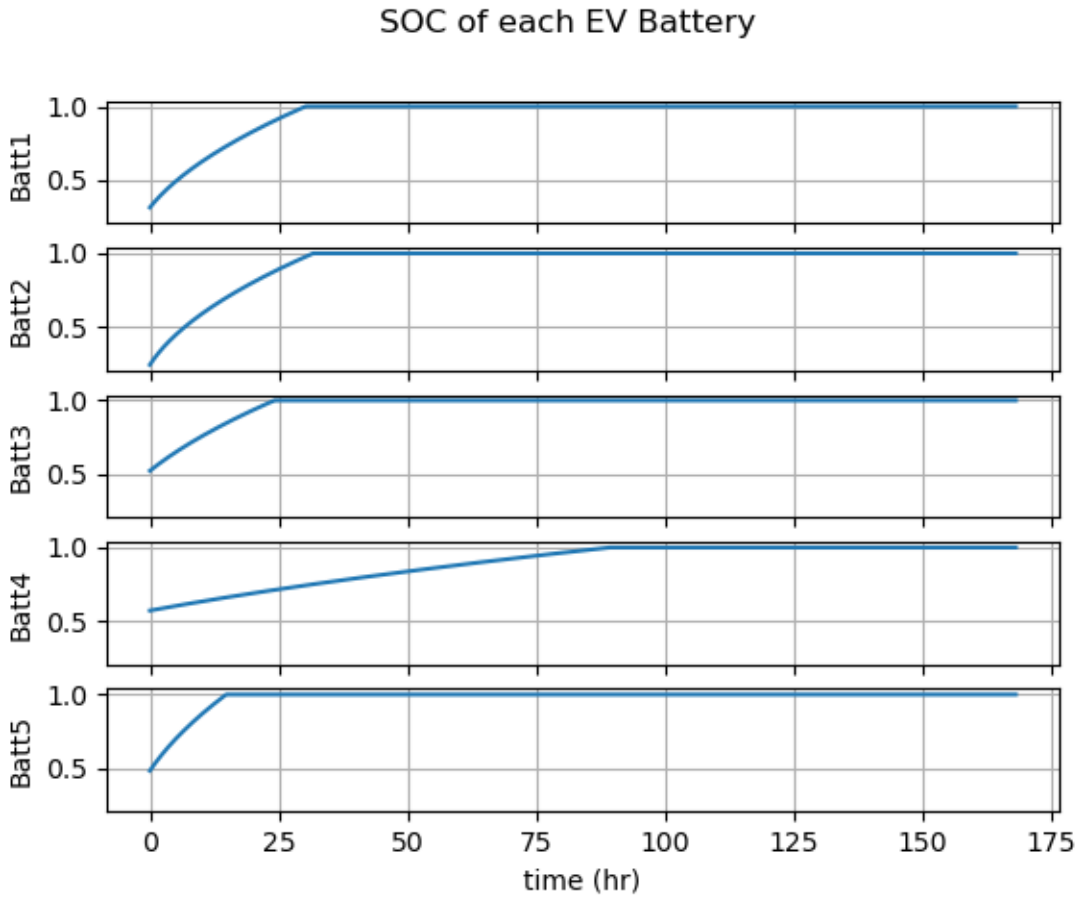
We'll be running the first example in the User Guide. From the top level of the HELICS Examples repository follow this path:

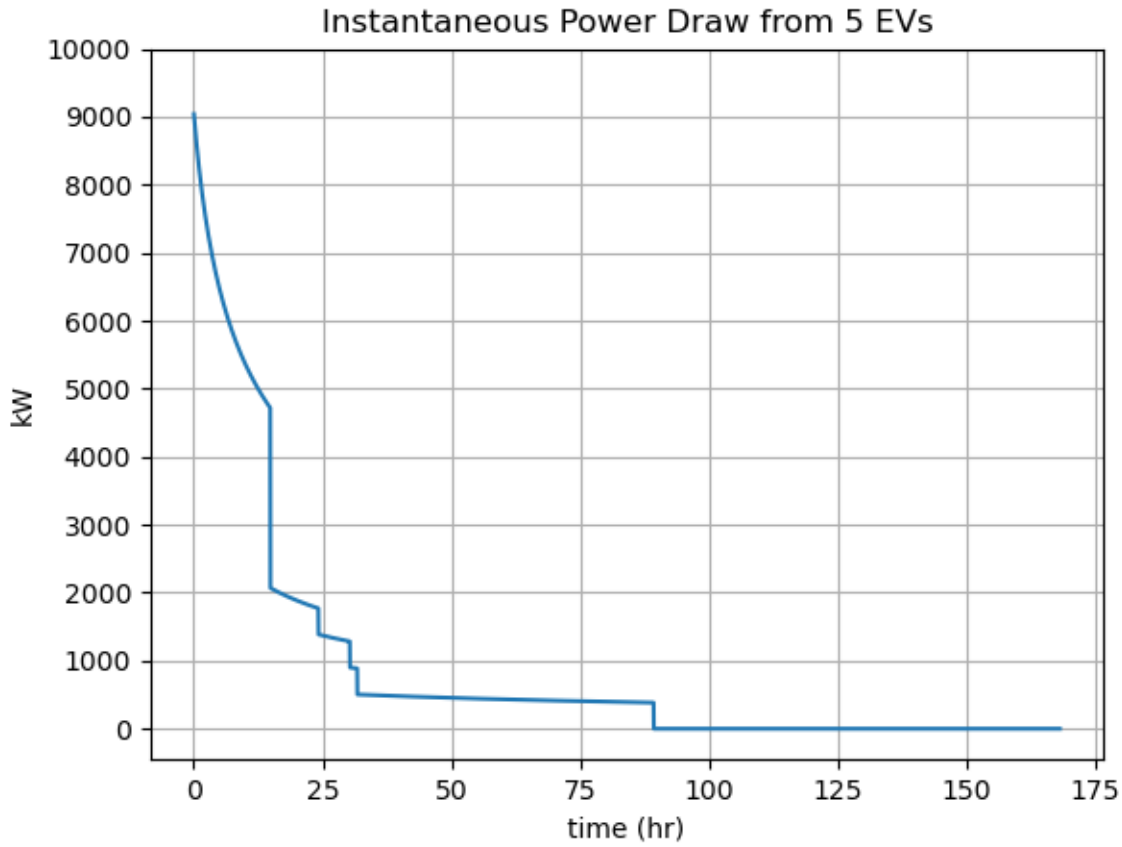
```
user_guide_examples/fundamental/fundamental_default
```

1.5 Run the Fundamental Default Example

```
helics run --path=fundamental_default_runner.json
```

The `helics run` command provides an easy way to launch a co-simulation based on the contents of the runner file ("fundamental_default_runner.json" in this case). In this case, the runner launches two Python federates created for this example, "Battery.py" (which models five EV batteries) and "Charger.py" (which models five EV chargers). You should see a few graphs that look like this:





1.6 Next Steps

- Look through the code of "Battery.py" and/or "Charger.py" to see how they work.
- If you're a little confused, look at the *documentation on the Fundamental Default example*
- If you're still a little confused, start from *the beginning of the User Guide* to better understand HELICS principles and concepts.
- If this is all making sense now, *try running another example* to better understand some of the other features of HELICS.

CHAPTER

TWO

USER GUIDE

Co-simulation is a powerful analysis technique that allows simulators of different domains to interact through the course of the simulation, typically by dynamically exchanging information that defines boundary conditions for other simulators. HELICS is a co-simulation platform that has been designed to allow integration of these simulators across a variety of computation platforms and languages. HELICS has been designed with power system simulation in mind (GridLAB-D, GridDyn, MATPOWER, OpenDSS, PSLF, InterPSS, FESTIV) but is general enough to support a wide variety of simulators and co-simulation tasks. Support for other domains is anticipated to increase over time.

2.1 Orientation

There are a number of classes of HELICS users:

- **New users** that have little to no experience with HELICS and co-simulation in general
  - Start with ***Installation***
  - Read the ***Fundamental Topics***
  - Try the ***Examples***
- **Intermediate users (Modelers)** that have run co-simulations with HELICS using simulators in which somebody else has done the simulator integration with HELICS.
  - Review ***Fundamental Topics*** as needed
  - Look over the ***Advanced Topics*** to see which features of HELICS may be most useful for your analysis.
    - *Multi-Source Inputs* (*example*)
    - *Broker Hierarchies* (*example*)
    - *HELICS Core Types* (*example*)
    - *Queries* (*example*)
    - *Simultaneous Co-simulation* (*example*)
    - *Multiple Co-simulation Orchestration* (*example*)
    - *Encrypted Communication*
- **Experienced users (Integrators)** that are incorporating a new simulator and need to know how to use specific features in the HELICS API
  - Look in the ***Configurations Options Reference*** or jump straight to the API references
    - C++
    - C++98
    - *C*

    * Python
    * Julia
    * nim

- **Developers** of HELICS who are improving HELICS functionality and contributing to the code base
  - See the ***Developer Guide***

2.2 HELICS Installation

2.2.1 Installing the Pre-Compiled Libraries

Download pre-compiled libraries from the releases page and add them to your path. Windows users should install the latest version of the Visual C++ Redistributable. The installers come with bindings for Java extensions precompiled as part of the installation. All you need to do is add the relevant folders to your User's PATH variables.

On Windows, you can visit `Control Panel -> System -> Advanced System Settings -> Environment Variables` and edit your user environment variables to add the necessary Path, JAVAPATH environment variables to the corresponding HELICS installed locations.

On MacOS or Linux, you can edit your `~/.bashrc` to add the necessary PATH, JAVAPATH environment variables to the corresponding HELICS installed locations.

Be sure to restart your CMD prompt on Windows or Terminal on your MacOS/Linux to ensure the new environment variables are in effect.

2.2.2 Install using Spack (macOS, Linux)

Install Spack (a HELICS package is included in the Spack develop branch and Spack releases after v0.14.1).

Run the following command to install HELICS (this may take a while, Spack builds all dependencies from source!):

```
spack install helics
```

To get a list of installation options, run:

```
spack info helics
```

To enable or disable options, use `+`, `-`, and `~`. For example, to build with MPI support on the command run would be:

```
spack install helics +mpi
```

2.2.3 OS Specific installation from source

Windows Installation

Windows Installers

Windows installers are available with the different releases. The release includes zip archives with static libraries containing both the Debug version and Release version for several versions of Visual Studio. There is also an installer and zip file for getting the HELICS apps and shared library along with a pre-built Java 1.8 interface. There is also an archive with just the C shared library and headers, intended for use with 3rd party interfaces.

Build Requirements

- Microsoft Visual C++ 2019 version 16.10 or newer (MS Build Tools also works)
- CMake 3.22 or newer
- git
- Boost 1.83 or newer (needs to have boost cmake configuration)
- MS-MPI v8 or newer (if MPI support is needed)

Setup for Visual Studio

Note: Keep in mind that your CMake version should be newer than the boost version and your visual studio version. If you have an older CMake, you may want an older boost version. Alternatively, you can choose to upgrade your version of CMake.

Set up your Environment

1. Install Microsoft Visual C++ 2019 version 16.10 or newer with C++20 support MSVC

2. Install Boost 1.83 or later. For CMake to detect it automatically either extract Boost to the root of your drive, or set the `BOOST_INSTALL_PATH` environment variable to the install location. The CMake will only automatically find Boost 1.83 or newer. Building with Visual Studio 2019 requires Boost 1.83 or newer and CMake 3.22 or newer.

As an (experimental) alternative for installing Boost (and ZeroMQ), you can use vcpkg. It is slower because it builds all dependencies but handles getting the right install paths to dependencies set correctly. To use it:

1. Follow the vcpkg getting started directions to install vcpkg
2. Run `cmake` using `-DCMAKE_TOOLCHAIN_FILE=[path to vcpkg]/scripts/buildsystems/vcpkg.cmake`, or by setting the environment variable `VCPKG_ROOT=[path to vcpkg]` prior to running `cmake`.

3. *Optional* Only if you need a global Install of ZeroMQ ZeroMQ. We **highly recommend skipping** this step and building HELICS via CMake with the `HELICS_ZMQ_SUBPROJECT=ON` option enabled (which is default on Windows) to automatically set up a project-only copy of ZeroMQ. The ZeroMQ Windows installer is **very** outdated and will not work with new versions of Visual Studio. The CMake generator from ZeroMQ on Windows also works and can be used to store ZMQ in another location that will need to be specified for HELICS.

4. *Optional* Install MS-MPI if you need MPI support.

5. *Optional* Install SWIG if you wish to generate the interface libraries for Java, appropriate build files are included in the repository so it shouldn't be necessary to regenerate unless the libraries are modified. For C# a SWIG install is necessary. The simplest way to install SWIG is to use chocolatey from Windows PowerShell with

```
choco install swig
```

6. Open a Developer PowerShell for Visual Studio command line and make sure *CMake* and *git* are available in the Command Prompt. This can be done with `Get-Command`. If they aren't, add them to the system PATH variable.

```
PS C:\Users\sampleUser\localrepos\HELICS> Get-Command cmake

CommandType   Name          Version    Source
-----------   ----          -------    ------
Application   cmake.exe     3.22.3.0   C:\Program Files\CMake\bin\cmake.exe


PS C:\Users\sampleUser\localrepos\HELICS> Get-Command git
```

(continues on next page)

(continued from previous page)

```
CommandType   Name          Version    Source
-----------   ----          -------    ------
Application   git.exe       2.35.1.2   C:\Program Files\Git\cmd\git.exe


PS C:\Users\sampleUser\localrepos\HELICS> Get-Command cmake-gui

CommandType   Name          Version    Source
-----------   ----          -------    ------
Application   cmake-gui.exe 3.22.3.0   C:\Program Files\CMake\bin\cmake-gui.exe
```

Getting and building from source

1. Open the Developer PowerShell VS Command prompt. Navigate to where you would like to project to be and use `git clone` to check out a copy of HELICS.

```
git clone https://github.com/GMLC-TDC/HELICS.git
```

2. Go to the checked out HELICS project folder (the default folder name is HELICS). Create a build folder and go to the build folder.

```
cd HELICS
mkdir build
cd build
```

3. Run CMake or CMake GUI. It should automatically detect where MPI is installed if the system path variables are set up correctly, otherwise you will have to set the CMake path manually. `ZMQ_LOCAL_BUILD` is set to `ON` by default so ZeroMQ will automatically be built unless the option is changed.

Make sure to set `CMAKE_INSTALL_PREFIX` to the path of the install folder.

If you need CMake to use a generator for an IDE or build system other than the default (ex: Ninja instead of a Visual Studio project), the `-G` option can be used to specify one of the generators listed by `CMake --help`. If you are using a Visual Studio generator, such as Visual Studio 2019, and need to select an architecture other than the default (ex: building a 32-bit target on a 64-bit host or vice versa), the `-A` option can be used to specify a target platform name. For example, for a 32-bit x86 build with Visual Studio 2019 on a 64-bit copy of Windows, you would use the cmake options `-G "Visual Studio 16 2019" -A Win32`. Similarly, `-A x64` can be used to build for an x64 processor.

Information on CMake usage and cross-compiling for different target architectures can be found in the CMake documentation at https://cmake.org/cmake/help/latest/manual/cmake-generators.7.html, and is recommended as a source of information on CMake as it will be more up-to-date on the latest version of CMake than this guide.

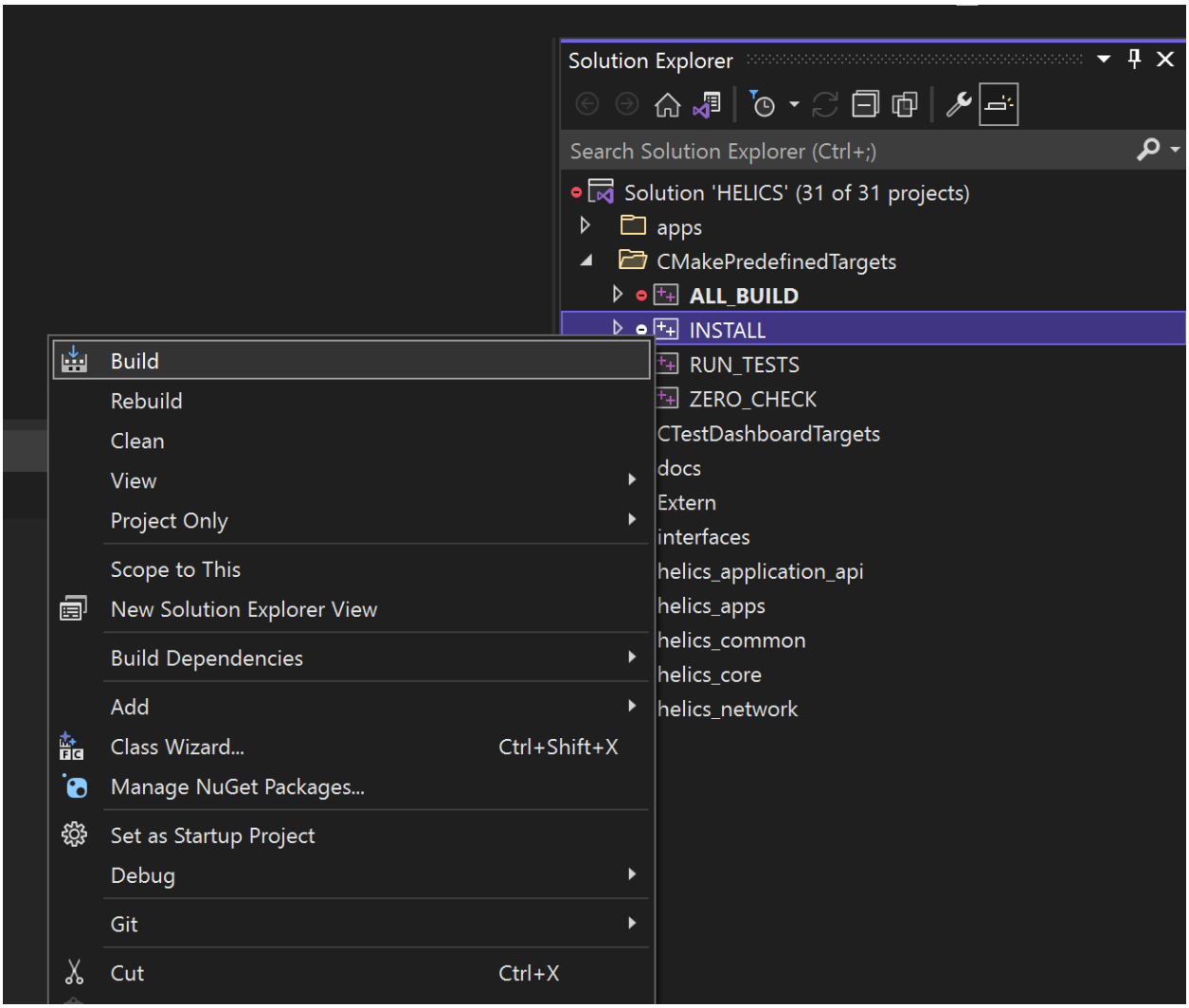
To avoid problems when building later, the target architecture and Visual Studio version should match the version of the Boost libraries you are using.

If you installed Boost into the root of the C or D drives with the default location (or the `BOOST_INSTALL_PATH` environment variable has been set), CMake should automatically detect their location. Otherwise the location will need to be manually given to CMake through `Boost_INCLUDE_DIR`. This should be used if the Boost version is new and not known to HELICS or you want to be explicit about which boost folder to use. NOTE: CMake 3.14 and later separate the architecture into a separate field for the generator

A basic call with `cmake` using Visual Studio 2022 on a 64bit Windows machine and installing to a folder called `install` inside the repository would be:

```
    cmake --install-prefix 'C:\Users\sampleUser\localrepos\HELICS - Copy\install' -G
↪"Visual Studio 17 2022" ..
```

4. Open the Visual Studio solution generated by CMake (his can be done from the command prompt with `start HELICS.sln`). In the *Solution Explorer* Under `Solution 'HELICS'\CMakePredefinedTargets`, right-click on `INSTALL` and select `Build`:



Alternatively, in the MSBuild command prompt, run the command `msbuild HELICS.sln` from the build folder to compile the entire solution.

"HELICS.sln" can be replaced with the name of one of the projects to build only that part of HELICS.

If the build was successful there should be a "bin" folder inside the "install" folder with `helics.dll` inside (or `helicsd.dll` if Debug mode).

5. *Optional* If interfacing with PYHELICS (assuming already installed via `pip install helics`) the `PYHELICS_INSTALL` environment variable needs to be set to the path of the "install" folder, and "install\bin" (the folder with "helics.dll") needs to be added to the system path. This can be done via the environment variable GUI in windows or temporarily via the command line[1]

[1] Especially if you plan on regularly switching between versions of HELICS temporarily setting `PYHELICS_INSTALL` might not be such a bad idea.

In Cmd:

```
set PYHELICS_INSTALL=C:\path\to\HELICS\install
set PATH=%PATH%;%PYHELICS_INSTALL%\bin
```

In Powershell:

```
$env:PYHELICS_INSTALL = "C:\path\to\HELICS\install"
$env:Path += ";$env:PYHELICS_INSTALL\bin"
```

To verify PYHELICS is pointing to the right version try:

```
helics --version
```

Testing

A quick test is to double check the versions of the HELICS player and recorder (located in the 'build/src/helics/apps/player/Debug' folder):

```
> cd C:/Path/To/build/src/helics/apps/Debug

> helics_player.exe --version
x.x.x 20XX-XX-XX

> helics_recorder.exe --version
x.x.x 20XX-XX-XX
```

there may be additional build information if a non tagged version is built.

MSYS2

MSYS2 provides a Linux like terminal environment on your Windows system. MSYS2 can be installed from here. Once MSYS2 has been installed start up msys2.exe. Follow first time updates as described on the MSYS2 website.

Using pacman package manager

HELICS is available on the Mingw-32 and Mingw-64 environments through the MSYS2 repositories. From the MINGW64 shell

```
$ pacman -Sy mingw64/mingw-w64-x86_64-helics
:: Synchronizing package databases...
 mingw32               453.3 KiB  2.86 MiB/s 00:00 [####################] 100%
 mingw32.sig           119.0   B  0.00   B/s 00:00 [####################] 100%
 mingw64               456.0 KiB  2.77 MiB/s 00:00 [####################] 100%
 mingw64.sig           119.0   B  0.00   B/s 00:00 [####################] 100%
 msys                  185.9 KiB  1804 KiB/s 00:00 [####################] 100%
 msys.sig              119.0   B  0.00   B/s 00:00 [####################] 100%
resolving dependencies...
looking for conflicting packages...

Packages (8) mingw-w64-x86_64-gcc-libs-9.2.0-2  mingw-w64-x86_64-gmp-6.2.0-1
             mingw-w64-x86_64-libsodium-1.0.18-1
             mingw-w64-x86_64-libwinpthread-git-8.0.0.5574.33e5a2ac-1
             mingw-w64-x86_64-mpc-1.1.0-1  mingw-w64-x86_64-mpfr-4.0.2-2
```

(continues on next page)

(continued from previous page)

```
             mingw-w64-x86_64-zeromq-4.3.2-1  mingw-w64-x86_64-helics-2.4.0-1

Total Download Size:    9.17 MiB
Total Installed Size:  65.78 MiB

:: Proceed with installation? [Y/n] y
```

you will be asked to proceed with installation, answering `y` will install HELICS and the required dependencies.

```
$ helics_broker --version
2.4.0 (2020-02-16)
```

The helics apps and libraries are now installed, and can be updated when HELICS gets an update. For the MINGw32 use

```
$ pacman -Sy mingw32/mingw-w64-i686-helics
```

if you are installing both the 32 and 64 bit versions or just want a simpler command to type

```
$ pacboy -Sy helics
:: Synchronizing package databases...
```

if the python interface is needed on MSYS2 it can be installed through pip but requires some setup first.

```
$export CMAKE_GENERATOR="MSYS Makefiles"
$pip install helics
```

This will install the HELICS python extension in the correct location. The pacman package should be installed first

Building HELICS From Source on Windows with MSYS2

After MSYS2 has been successfully updated Some packages need to be installed in order to configure and build HELICS. The following packages need to be installed:

- base-devel
- mingw-w64-x86_64-toolchain
- git
- mingw-w64-x86_64-CMake
- mingw-w64-x86_64-boost
- mingw-w64-x86_64-qt6 (only if you want to be able to run cmake-gui which this guide recommends.)
- mingw-w64-x86_64-zeromq

All packages can be installed by typing the following:

```
$ pacman -Sy base-devel mingw-w64-x86_64-toolchain git mingw-w64-x86_64-CMake mingw-w64-
↪x86_64-boost mingw-w64-x86_64-qt5 mingw-w64-x86_64-zeromq
```

For base-devel and mingw-w64-x86_64-toolchain you may have to hit enter for installing all packages that are part of the group package. The qt5 package is quite large, if you are only using it once it might be faster to use ccmake which is a text based interface to CMake. After all the packages have been installed has been done /mingw64/bin must be in the PATH environment variable. If it isn't then it must be added. Please note that this is only necessary if you

are compiling in MSYS2 shell. If you are compiling in the MSYS2 MINGW-64bit shell then /mingw64/bin will be automatically added to the PATH environment variable. If not

```
$ export PATH=$PATH:/mingw64/bin
```

Download HELICS Source Code

Now that the MSYS2 environment has been setup and all prerequisite packages have been installed the source code can be compiled and installed. The HELICS source code can be cloned from GitHub by performing the following:

```
$ git clone https://github.com/GMLC-TDC/HELICS.git
```

git will clone the source code into a folder in the current working directory called HELICS. This path will be referred to by this guide as HELICS_ROOT_DIR.

Compiling HELICS From Source

Change directories to HELICS_ROOT_DIR. Create a directory called helics-build. This can be accomplished by using the mkdir command. cd into this directory. Now type the following:

```
$ cmake-gui ../
```

If this fails that is because mingw-w64-x86_64-qt5 was not installed. If you did install it the CMake gui window should pop up. click the Advanced check box next to the search bar. Then click Configure. A window will pop up asking you to specify the generator for this project. Select "MSYS Makefiles" from the dropdown menu. The native compilers can be used and will most likely default to gcc. The compilers can also be specified manually. Select Finish; once the configure process completes finished several variables will show up highlighted in red. Since this is the first time setup the Boost and ZeroMQ library. Below are the following CMake variables that could to be verified.

- `HELICS_ENABLE_CXX_SHARED_LIB` should be checked if you are using HELICS with GridLAB-D, GridLAB-D dynamically links with the shared c++ library of HELICS, the default is off so you would need to change it

For others the advanced checkbox can be selected to see some other variables

- Boost_INCLUDE_DIR C:/msys64/mingw64/include
- Boost_LIBRARY_DIR_DEBUG/RELEASE C:/msys64/mingw64/bin
- CMake_INSTALL_PREFIX /usr/local or location of your choice
- ZeroMQ_INCLUDE_DIR C:/msys64/mingw64/include
- ZeroMQ_INSTALL_PATH C:/msys64/mingw64
- ZeroMQ_LIBRARY C:/msys64/mingw64/bin/libzmq.dll.a
- ZeroMQ_ROOT_DIR C:/msys64/mingw64

Once these CMake variables have been correctly verified click Configure if anything was changed. Once that is complete click Generate then once that is complete the CMake-gui can be closed.

Back in the MSYS2 command window[make sure you are in the build directory] type:

```
$ make -j x
```

where x is the number of threads you can give the make process to speed up the build. Then once that is complete type: `make -j` will just use the number of cores you have available

```
$ make install
```

unless you changed the value of CMake_INSTALL_PREFIX everything the default install location /usr/local/helics_2_1_0. This install path will be referred to as HELICS_INSTALL for the sections related to GridLab-D. If you want to build Gridlab-d on Windows with HELICS see Building with HELICS. Please use branch feature/1179 to build with HELICS 2.1 or later instead of the branch listed.

Compiling with clang

Clang does not work to compile on MSYS2 at this time. It has in the past but there are various issues with the clang standard library on MSYS yet so this will be updated if the situation changes. It is getting closer as of (1/30/2020) Mostly it compiles when linked with Libc++ and libc++abi, but there seems to be some missing functions as of yet, so cannot be used other than for some warning checks.

For building with clang using libc++, CMake 3.18+ must be used.

Building with mingw

HELICS can also be built with the standalone MinGW

- We assume you have MinGW installed or know how to install it.
- Boost; you can use the Windows installer for Boost installed in the default location
- Run CMake to configure and generate build files, using "MinGW Makefiles" as the generator,
- Run mingw32-make -j to build

Building with cygwin

Cygwin is another UNIX like environment on Windows. It has some peculiarities. HELICS will only build on the 32 bit version due to incompatibilities with ASIO and the 64 bit build. But it does build on the 32 bit versions completely and on the 64 bit version if `HELICS_DISABLE_ASIO=ON` is set

- required packages include CMake, libboost-devel, make, gcc, g++, libzmq(if using zmq)
- use the unix makefiles generator

Mac Installation

Requirements

- C++20 compiler
- CMake 3.22 or higher
- XCode 14 or higher
- git
- Boost 1.83 or newer
- ZeroMQ 4.2 or newer (if ZeroMQ support is needed)
- MPI-2 implementation (if MPI support is needed)

Useful Resources

Some basics on using the macOS Terminal (or any Unix/Linux shell) will be useful to fully understand this guide. Articles and tutorials you may find useful include:

- How to add a new path to PATH

- Getting to Understand Linux Shell(s)
- Paths - where's my command
- Unix/Linux for Beginners
- Settling into Unix.

Setup

Note: Keep in mind that your cmake version should be newer than the boost version. If you have an older cmake, you may want an older boost version. Alternatively, you can choose to upgrade your version of cmake.

To set up your environment:

1. (if needed) Install git on your system for easy access to the HELICS source. Download from git-scm. This installs the command line which is described here. GUI's interfaces such as SourceTree are another option.
2. (if desired) Many required libraries are easiest installed using the homebrew package manager. These directions assume this approach, so unless you prefer to track these libraries and dependencies down yourself, install it if you don't have it yet. As an alternative package manager, you can use vcpkg – it is slower because it builds all dependencies for source, but instead of following step below you could either run `cmake` using `-DCMAKE_TOOLCHAIN_FILE=[path to vcpkg]/scripts/buildsystems/vcpkg.cmake` as shown in the vcpkg getting started instructions, or by setting the environment variable `VCPKG_ROOT=[path to vcpkg]` prior to running `cmake`.
3. (if needed) Setup a command-line compile environment

   a) Install a C++20 compiler. e.g. `clang` from the Xcode command line tools. These can be installed from the command line in Terminal by typing `xcode-select --install` and following the on-screen prompts. b) Install cmake with `brew install cmake`. Alternately, a DMG file is available for cmake from their website.
4. Install most dependencies using homebrew.

   ```
   brew install boost
   brew install zeromq
   brew install cmake
   ```
5. Make sure *cmake* and *git* are available in the Command Prompt with `which cmake` and `which git` If they aren't, add them to the system PATH variable.

Getting and building from source:

1. Use `git clone` to check out a copy of HELICS.
2. Create a build folder. Run cmake and give it the path that HELICS was checked out into.

   ```
   git clone https://github.com/GMLC-TDC/HELICS
   cd HELICS
   mkdir build
   cd build
   ```

Compile and Install

There are a number of different options and approaches at this point depending on your needs, in particular with respect to programming language support.

Note: For any of these options, if you want to install in a custom location, you can add the following CMake argument: `-DCMAKE_INSTALL_PREFIX=/path/to/install/folder/`. There are also many other options, and you can check them out by running `ccmake .` in the `build` folder.

Keep in mind running HELICS commands like `helics_app` will not work from just any old random folder with a custom install folder. You will either need to run them from inside the `bin` subfolder of your custom install, or provide a more complete path to the command. To run HELICS commands from any folder, you must add the `bin` subfolder of your custom install to the `PATH` environment variable. See the first link in the *Useful Resources* section for details.

Basic Install (without language bindings)

Run the following:

```
cmake ../
ccmake . # optional, to change install path or other configuration settings
make
make install
```

Building HELICS using gcc

Firstly, you'll need gcc. You can `brew install gcc`. Depending on the version of gcc you'll need to modify the following instructions slightly.

First you will need to build boost using gcc from source. Download the latest version of boost from the boost.org website. In the following example we are doing to use boost v1.84.0 Keep in mind that your cmake version should be newer than the boost version, so if you have an older cmake you may want an older boost version. Alternatively, you can choose to upgrade your version of cmake as well.

Unzip the folder `boost_1_84_0` to any location, for example Downloads.

```
$ cd ~/Downloads/boost_1_84_0
$ ./bootstrap.sh --prefix=/ --prefix=$HOME/local/boost-gcc-1.84.0
```

Open `project-config.jam` and changes the lines as follows:

```
# Compiler configuration. This definition will be used unless
# you already have defined some toolsets in your user-config.jam
# file.
# if ! darwin in [ feature.values <toolset> ]
# {
    # using darwin ;
# }

# project : default-build <toolset>darwin ;

using gcc : 8.2 : /usr/local/bin/g++-8 ;
```

```
$ ./b2
$ ./b2 install
$ # OR
$ ./bjam cxxflags='-fPIC' cflags='-fPIC' -a link=static install # For static linking
```

This will install boost in the `~/local/boost-gcc-1.84.0` folder

Next, you will need to build HELICS and tell it what the `BOOST_ROOT` is.

```
$ cmake -DCMAKE_INSTALL_PREFIX="$HOME/local/helics-gcc-X.X.X/" -DBOOST_ROOT="$HOME/local/
↪boost-gcc-1.84.0" -DCMAKE_C_COMPILER=/usr/local/Cellar/gcc/8.2.0/bin/gcc-8 -DCMAKE_CXX_
```

(continues on next page)

(continued from previous page)

```
↪COMPILER=/usr/local/Cellar/gcc/8.2.0/bin/g++-8 ../
$ make clean; make -j 4; make install
```

Testing HELICS

Basic test (without language bindings)

A quick test is to double check the versions of the HELICS player and recorder:

```
cd /path/to/helics_install/bin

$ helics_player --version
x.x.x (20XX-XX-XX)

$ helics_recorder --version
x.x.x (20XX-XX-XX)
```

Testing HELICS with MATLAB support

To run the MATLAB HELICS extension, one would have to load the `helicsSharedLib` in the MATLAB file. This is run by the `helicsStartup` function in the generated MATLAB files. You can test this by opening MATLAB from the terminal or using the icon.

```
/Applications/MATLAB_R2017b.app/bin/matlab -nodesktop -nosplash -nojvm
```

and running

```
>> helicsStartup
```

Note: See Helics issue #763, if your installation doesn't point the dylib to the correct location.

You can run the following in two separate windows to test an example from the following repository:

```
git clone https://github.com/GMLC-TDC/HELICS-examples
```

Run the following in one MATLAB instance

```
matlab -nodesktop -nosplash
cd ~/GitRepos/GMLC-TDC/HELICS-examples/matlab
pireceiver
```

Run the following in a separate MATLAB instance.

```
matlab -nodesktop -nosplash
cd ~/GitRepos/GMLC-TDC/HELICS-examples/matlab
pisender
```

Linux Installations

Ubuntu Installation

Requirements

- Ubuntu 22.04 or newer

- C++20 compiler (GCC 11 or clang 15 or higher)
- CMake 3.22 or newer
- git
- Boost 1.83 or newer
- ZeroMQ 4.2 or newer (generally recommended but technically not essential)
- MPI-2 implementation (if MPI support is needed)

Setup

Note: Keep in mind that your CMake version should be newer than the boost version. If you have an older CMake, you may want an older boost version. Alternatively, you can choose to upgrade your version of CMake.

To set up your environment:

1. Install dependencies using apt-get.

   ```
   $ sudo apt install libboost-dev
   $ sudo apt install libzmq5-dev
   $ sudo apt install git
   $ sudo apt install cmake-curses-gui #includes ccmake gui
   ```

   As an alternative, you can use vcpkg – it is slower because it builds all dependencies from source but could have newer versions of dependencies than apt-get. To use it, follow the vcpkg getting started directions to install vcpkg and then run `cmake` using `-DCMAKE_TOOLCHAIN_FILE=[path to vcpkg]/scripts/buildsystems/vcpkg.cmake`, or by setting the environment variable `VCPKG_ROOT=[path to vcpkg]` prior to running `cmake`.

2. Make sure *CMake* and *git* are available in the Command Prompt. If they aren't, add them to the system PATH variable. This should be covered by installing them as above.

   ```
   $ which git
   /usr/local/git

   $ which cmake
   /usr/local/cmake

   $ which ccmake
   /usr/local/ccmake
   ```

3. Checkout the source code and build from source:

Notes for Ubuntu

```
git clone https://github.com/GMLC-TDC/HELICS
cd HELICS
mkdir build
cd build
cmake ../
# the options can be modified by altering the CMakeCache.txt file or by using the ccmake␣
↪command to edit them
# the cmake GUI will also work to graphically edit the configuration options.
ccmake . #optional, invokes the cmake GUI to edit options
make
make install
```

Testing

A quick test is to double check the versions of the HELICS player and recorder:

```
cd /path/to/helics_install/bin

$ helics_player --version
3.x.x (20XX-XX-XX)

$ helics_recorder --version
3.x.x (20XX-XX-XX)
```

To run a full co-simulation go to the "examples/comboFederate1" folder and run the "run3.sh". This will produce four output files: "broker.out", "fed1.out", "fed2.out", and "fed3.out". Opening "fed1.out" should show it sending messages to fed2 and receiving messages from fed3.

```
[2022-01-19 11:53:05.308] [console] [info] fed1 (0)::registering PUB fed1/pub
[2022-01-19 11:53:05.308] [console] [info] fed1 (0)::registering Input
entering init State
[2022-01-19 11:53:05.310] [console] [info] fed1 (131072)[t=-9223372036.
↪854776]::Registration Complete
[2022-01-19 11:53:05.311] [console] [debug] fed1 (131072)[t=-9223372036.
↪854776]::Granting Initialization
[2022-01-19 11:53:05.311] [console] [debug] fed1 (131072)[t=-9223372036.854776]::Granted␣
↪Time=-9223372036.854776
entered init State
[2022-01-19 11:53:05.312] [console] [debug] fed1 (131072)[t=-1000000]::Granting Execution
[2022-01-19 11:53:05.312] [console] [debug] fed1 (131072)[t=0]::Granted Time=0
entered exec State
message sent from fed1 to fed2/endpoint at time 1
[2022-01-19 11:53:05.313] [console] [debug] fed1 (131072)[t=1e-09]::Granted Time=1e-09
processed time 1e-09
received message from fed3/endpoint at 0 ::message sent from fed3 to fed1/endpoint at␣
↪time 1
received updated value of 1 at 1e-09s from fed2/pub
message sent from fed1 to fed2/endpoint at time 2
[2022-01-19 11:53:05.315] [console] [debug] fed1 (131072)[t=2e-09]::Granted Time=2e-09
processed time 2e-09
received message from fed3/endpoint at 1e-09 ::message sent from fed3 to fed1/endpoint␣
↪at time 2
received updated value of 2 at 2e-09s from fed2/pub
message sent from fed1 to fed2/endpoint at time 3

...
```

Docker

Requirements

- Docker 17.05 or higher

Dockerfile

This `Dockerfile` will build and install HELICS in Ubuntu 24.04 with Python support.

```
FROM ubuntu:24.04 as builder

WORKDIR /root/develop

RUN apt update && apt install -y \
  libzmq5-dev python3-dev \
  libboost-all-dev \
  build-essential swig cmake git

RUN git clone --recurse-submodules \
  https://github.com/GMLC-TDC/HELICS.git helics

WORKDIR /root/develop/helics

RUN cmake \
  -DCMAKE_INSTALL_PREFIX=/helics \
  -DCMAKE_BUILD_TYPE=Release \
  -B build

RUN cmake --build build -j -t install


FROM ubuntu:24.04

COPY --from=builder /helics /usr/local/

ENV PYTHONPATH /usr/local/python

# Python must be installed after the PYTHONPATH is set above for it to
# recognize and import libhelicsSharedLib.so.
RUN apt update && apt install -y --no-install-recommends \
  libboost-filesystem1.83.0 libboost-program-options1.83.0 \
  libboost-test1.83.0 libzmq5 pip python3-dev

RUN pip install helics

CMD ["python3", "-c", "import helics; print(helics.helicsGetVersion())"]
```

Build

To build the Docker image, run the following from the directory containing the `Dockerfile`:

```
$ docker build -t helics .
```

Run

To run the Docker image as a container, run the following:

```
$ docker run -it --rm helics
```

Doing so should print the version and exit.

HELICS with language bindings support

HELICS with Python

`pip install helics` should work for most use cases.

For developers and special use cases, the HELICS Python module code can be found in the PyHELICS repository.

HELICS with Java

To install HELICS with Java support, you will need to add `HELICS_BUILD_JAVA_INTERFACE=ON`.

HELICS with MATLAB

The Matlab interface to HELICS is found in the matHELICS repository.

HELICS with Simulink

A Simulink interface to HELICS is found in the simHELICS repository. This is a minimalistic interface that is built using Simulink S-functions.

HELICS with Octave

The Octave interface to HELICS is found in the matHELICS repository. It is currently built with the Matlab Interface.

Notes

The octave interface requires a newer octave 8.3 and above that supports building with mex files.

HELICS with C Sharp

C# is supported through SWIG. This requires swig being installed and generating the CMake for HELICS with HELICS_BUILD_CSHARP_INTERFACE=ON. If in Visual studio this will generate the appropriate files for C# usage.

A suitable version of swig can be installed on macOS, Windows, or Linux using:

```
pip install swig
```

Depending on your build environment on Windows, using Chocolatey to install swig might make it easier for the HELICS build scripts to automatically locate swig:

```
choco install swig
```

Linking with the HELICS Library

Once HELICS is built or installed it needs to be integrated into a project.

Language bindings

If the project is in Python, Matlab, Java, C#, Octave, or Julia, the language binding specific to that language is the best bet.

C based project

The C based shared library is the way to go. Either link with libhelics.lib/so/dylib and add the include helics.h or link the CMake target HELICS::helics.

C++98 or C++03 or C++11 of C++14

Then linking with the C shared library and using the C++98 header only wrapper is the most appropriate choice. The CMake target HELICS::helicsCpp98 can be used if using CMake.

C++17

If you are using C++17 and can install or generate the C++ shared library and make sure they are built with identical compilations configurations it is possible to link with the C++ shared library generated by HELICS. This can provide a richer interface and potentially slightly faster interaction. This can be done in a couple ways. If the project is CMake based then HELICS will install a configuration file that generates the targets for the HELICS::helicscpp library with the appropriate information for linking, this is the simplest approach. Unlike the C shared library, the C++ shared library requires the importing program to have the same compilation libraries since the library does not export std library symbols, or other symbols defined solely in header files, so those must have the same interpretation inside and out of the library for this to work. If some of the extensions available in the helics::apps are needed then the HELICS::helicscpp-apps target is also appropriate.

If you are not using CMake then link against the libhelicscpp.lib/so/dylib as appropriate for the operating system and include the headers directory. Also advisable (though not strictly necessary is to define HELICS_SHARED_LIBRARY as part of the compilation before including some headers). The apps library is `libhelicscpp-apps.lib/so/dylib` and the corresponding dll/so/dylib should be installed on the system path, or in the same directory for Windows, or added to the RPATH of the binary you're compiling on other (non-Windows) operating systems.

If you are using CMake and building HELICS as part of the project and want to use static libraries or use the apps library as a static library then HELICS supports building as a CMake subproject and linking the targets HELICS::apps or HELICS::application-api.

Troubleshooting shared library errors on Windows

If you encounter an error along the lines of `DLL load failed: The specified module could not be found` when attempting to use the C shared library, it is likely a required system dependency is missing. You can determine which DLL it is unable to find using a tool like https://github.com/lucasg/Dependencies to show which dependencies were not found when attempting to open the helics C shared library DLL. It is fine if it shows it can't find `WS2_32.dll`, but all other DLLs should be found. The most likely to be missing is `vcruntime140_1.dll`, which can be fixed by downloading the latest Visual C++ Redistributable from https://support.microsoft.com/en-us/help/2977003/the-latest-supported-visual-c-downloads and installing it.

Summary

- For Python(2,3), Java, Matlab, C#, Octave, Julia - use the defined interface
- For C use the C shared library
- For most C++ use the C++98 Wrapper to the C shared library
- For C++17 in common build configurations use the C shared library or the C++ shared library from the installers or build it yourself to ensure library compatibility.
- For specific C++ applications desiring static builds and using CMake, use HELICS as a subproject in CMake (the main use is some helics extensions, but other applications can use it as well)
- For specific C++ applications with particular build considerations or flags use HELICS as a subproject and link the static or shared C++ Libraries.

- For other languages - work with swig or use the foreign language capabilities of the language to import the C shared library.

HELICS CMake options

Main Options

- `CMake_INSTALL_PREFIX`: CMake variable listing where to install the files
- `HELICS_BUILD_APP_LIBRARY` : [Default=ON] Tell HELICS to build the helics apps shared library
- `HELICS_BUILD_APP_EXECUTABLES` : [Default=ON] Build some executables associated with the apps
- `HELICS_BUILD_BENCHMARKS` : [Default=OFF] Build some timing benchmarks associated with HELICS
- `HELICS_BUILD_CXX_SHARED_LIB` : [Default=OFF] Build C++ shared libraries of the Application API C++ interface to HELICS and if `HELICS_BUILD_APP_LIBRARY` is also enabled another C++ shared library with the APP library
- `HELICS_BUILD_EXAMPLES` : [Default=OFF] Build a few select examples using HELICS, this is mostly for testing purposes. The main examples repo is here
- `HELICS_BUILD_TESTS` : [Default=OFF] Build the HELICS unit and system test executables.
- `HELICS_ENABLE_LOGGING` : [Default=ON] Enable debug and higher levels of logging, if this is turned off that capability is completely removed from HELICS
- `HELICS_ENABLE_PACKAGE_BUILD` : [Default=OFF] Enable the generation of some installer packages for HELICS
- `HELICS_GENERATE_DOXYGEN_DOC` : [Default=OFF] Generate doxygen documentation for HELICS
- `HELICS_WITH_CMAKE_PACKAGE` : [Default=ON] Generate a `HELICSConfig.cmake` file on install for loading into other libraries
- `HELICS_BUILD_JAVA_INTERACE` : [Default=OFF] Build the HELICS Java Interface
- `HELICS_BUILD_CSHARP_INTERACE` : [Default=OFF] Build the HELICS C# Interface
- `CMAKE_CXX_STANDARD` : Specify the C++ standard to use in building. HELICS requires C++20 or higher and uses C++20 if no standard is specified.
- `HELICS_INSTALL` :[Default=ON] If set to off HELICS will not generate any install instructions

NOTE: All HELICS options are prefixed with HELICS_ to separate them from other libraries so HELICS can be used cleanly as a subproject.

Advanced Options

There are several different additional options available to configure HELICS for particular situations, most of which are not needed for general use and the default options should suffice.

HELICS Configuration options

These options effect the configuration of HELICS itself and how/what gets built into the HELICS core libraries

- `HELICS_ENABLE_ZMQ_CORE` : [Default=ON] Enable the HELICS ZeroMQ related core types
- `HELICS_ENABLE_TCP_CORE` : [Default=ON] Enable the HELICS TCP related core types
- `HELICS_ENABLE_UDP_CORE` : [Default=ON] Enable the HELICS UDP core type
- `HELICS_ENABLE_IPC_CORE` : [Default=ON] Enable the HELICS interprocess shared memory related core types

- HELICS_ENABLE_TEST_CORE : [Default=OFF] Enable the HELICS in process core type with some additional features for tests, required and enabled if the HELICS_BUILD_TESTS option is enabled
- HELICS_ENABLE_INPROC_CORE : [Default=ON] Enable the HELICS in process core type, required if HELICS_BUILD_BENCHMARKS is on
- HELICS_ENABLE_MPI_CORE : [Default=OFF] Enable the HELICS Message Passing Interface (MPI) related core types, most commonly used for High Performance Computing applications (HPC)

HELICS logging Options

- HELICS_ENABLE_TRACE_LOGGING : [Default=ON] Enable trace level of logging inside HELICS, if this is turned off that capability is completely removed from HELICS
- HELICS_ENABLE_DEBUG_LOGGING : [Default=ON] Enable debug levels of logging inside HELICS, if this is turned off that capability is completely removed from HELICS

Build configuration Options

Options effect the connection of libraries used in HELICS and how they are linked.

- HELICS_DISABLE_BOOST : [Default=OFF] Completely turn off searching and inclusion of boost libraries. This will disable the IPC core, disable the webserver and few other features, possibly more in the future.
- HELICS_DISABLE_WEBSERVER : [Default=OFF] Disable building the webserver part of the helics_broker_server and helics_broker. The webserver requires Boost to be enabled and HELICS_DISABLE_BOOST will take precedence.
- HELICS_DISABLE_ASIO : [Default=OFF] Completely turn off inclusion of ASIO libraries. This will disable all TCP and UDP cores, disable real time mode for HELICS, and disable all timeout features for the Library so **use with caution**.
- HELICS_ENABLE_SUBMODULE_UPDATE : [Default=ON] Enable CMake to automatically download the submodules and update them if necessary
- HELICS_ENABLE_ERROR_ON_WARNING :[Default=OFF] Turns on Werror or equivalent, probably not useful for normal activity, There isn't many warnings but left in to allow the possibility
- HELICS_ENABLE_EXTRA_COMPILER_WARNINGS : [Default=ON] Turn on higher levels of warnings in the compilers, can be turned off if you didn't need or want the warning checks.
- STATIC_STANDARD_LIB: [Default=""] link the standard library as a static library for no additional C++ system dependencies (recognized values are default, static, and dynamic, anything else is treated the same as default)
- HELICS_ENABLE_SWIG: [Default=OFF] Conditional option if HELICS_BUILD_MATLAB_INTERACE or HELICS_BUILD_JAVA_INTERACE is selected and no other option that requires swig is used. This enables swig usage in cases where it would not otherwise be necessary.
- HELICS_ENABLE_GIT_HOOKS: install a git hook to check clang format before a push
- Boost_NO_BOOST_CMAKE: [Default=OFF] This is an option related to the Boost find module, but is occasionally needed if a specific version of boost is desired and there is a system copy of BoostConfig.cmake. So if an incorrect version of boost is being found even when BOOST_ROOT is being specified this option might need to be set to ON.
- HELICS_DISABLE_C_SHARED_LIB: [Default=OFF] Turns off building of the HELICS C shared library. May be used for building apps that only use the (modern) C++ shared library. Using the C++98 wrapper requires the C shared library.

ZeroMQ related Options

- HELICS_USE_SYSTEM_ZEROMQ_ONLY: [Default=OFF] Only find Zeromq through the system libraries, never attempt a local build.
- HELICS_USE_ZMQ_STATIC_LIBRARY: [Default=OFF (unless only libzmq-static found)] Build and link Zeromq using a static library. (NOTE: This has licensing implications if the resulting binary is distributed)
- HELICS_ZMQ_SUBPROJECT: [Default=ON (MSVC) OFF(otherwise)] Allow ZeroMQ to be built as a subproject if a system library is not found
- HELICS_ZMQ_FORCE_SUBPROJECT: [Default=OFF] Force ZMQ to be built and linked as a subproject.
- ZeroMQ_INSTALL_PATH: Can be used to specify a path to ZeroMQ for inclusion.

Options related to helics tests and CI configurations

- HELICS_TEST_CODE_COVERAGE :[Default=OFF] Turn on code coverage testing, enables additional linkage and options inside HELICS for coverage testing, mainly useful inside the CI or for testing.
- HELICS_ENABLE_SUBPROJECT_TESTS: [Default=OFF] Turn on some additional tests for using HELICS as a subproject, mainly used in some of the CI testing to make sure HELICS works as a subproject.
- HELICS_ENABLE_CLANG_TOOLS: [Default=OFF] Enables some helper targets for using clang-tidy and clang-format.

Options related to using external/vendored libraries

These options are for making HELICS look for an external installation or system copy of libraries instead of using the vendored copy included with HELICS. The search process for locating the external copy of libraries follows the typical CMake Config Mode Search Procedure. Most users should leave these options OFF.

When enabling these options, if the static library variant is found it *must* have been compiled using the option CMAKE_POSITION_INDEPENDENT_CODE=ON or *you will encounter linking errors*.

Furthermore, there are no guarantees that HELICS will compile using arbitrary versions of these libraries other than the exact version that is included as a vendored library in the HELICS repository. To get the current commit hashes for the libraries included in the HELICS repository, the command `git submodule status` can be run using a git clone of the HELICS repository with your desired branch checked out (*beware that if there is a + before the hash* it means you *must* run `git submodule update` and *re-run the status command* to get the correct commit hash for the submodule the currently checked out branch). The GitHub website will also show the current commit hash for submodules when you use a browser to view the contents of the ThirdParty folder; for example, https://github.com/GMLC-TDC/HELICS/tree/main/ThirdParty will show the current commit hashes used for the various library submodules on the main HELICS branch.

- HELICS_USE_EXTERNAL_FMT : Use an external copy of the fmt library.
- HELICS_USE_EXTERNAL_SPDLOG : Use an external copy of the spdlog library. Note that an external spdlog library may itself depend on an external fmt library, resulting in weird, confusing errors if it is incompabile with the internal fmt library used by HELICS.
- HELICS_USE_EXTERNAL_UNITS : Use an external copy of the LLNL units library.

Hidden Options

There are a few options in the CMake system that are not visible in the GUI they mainly deal with particular situations related to release, testing, benchmarks, and code generation and should not be normally used. They are all default off unless otherwise noted.

- HELICS_SWIG_GENERATE_INTERFACE_FILES_ONLY : Use swig to generate the interface files for the different languages but don't compile them.
- HELICS_OVERWRITE_INTERFACE_FILES : Instruct CMake to take the generated files, and overwrite the existing interface files for the given language, only applies to Matlab and Java. This is used in the generation of the interface files for releases and the git repo. It is only active is HELICS_SWIG_GENERATE_INTERFACE_FILES_ONLY is enabled.
- HELICS_DISABLE_SYSTEM_CALL_TESTS : There are a few test that execute system calls, which could be problematic to compile or execute on certain platforms. This option removes those tests from compilation.
- INSTALL_SYSTEM_LIBRARIES : Install system libraries with the installation, mainly useful for making a complete installer package with all needed libraries included.
- HELICS_INSTALL_PACKAGE_TESTS : Set the find_package tests to only look for HELICS in the system install paths, and enable the package-config-tests
- HELICS_DISABLE_GIT_OPERATIONS : will turn off any of the helper tools that require git, this is useful in a couple cases for building packages and other situations where updates shouldn't be checked and no modifications should be made.
- HELICS_SKIP_ZMQ_INSTALL: This is only relevant if ZMQ is built as part of the compilation process, but it skips the installation of zmq as part of HELICS install in that case.
- HELICS_BENCHMARK_SHIFT_FACTOR: For running the benchmarks this shift factor can be used to scale the number of federates used for the benchmark tests. If used it is required to be a number and is power of 2 shift from nominal values. For example for a small system a shift factor of -1 or -2 might be appropriate for the benchmarks not to take too long. The default for systems with 4 or fewer cores is -1 and 0 for larger compute systems. For small 2 core systems a value of -2 might be appropriate. For some very large systems a bigger value might be able to be used.
- HELICS_HIDE_CMAKE_VARIABLES: When using HELICS as a subproject in a CMake build this option can be enabled to hide all HELICS related variables in CMake so they won't show up in the CMake GUI.
- WIN32_WINNT: On some systems like msys or cygwin some libraries need to know the version of windows and it is not automatically detected so this variable can override the default which is set up for windows 10.

The following are a few things that could be useful to know before starting out.

Firstly, you can follow HELICS development on our GitHub page. HELICS is open-source. The development team uses `git` for version control, and GitHub to host the code publicly. The latest HELICS development work will be on the `main` branch. Tagged releases also occur on the `main` branch. If you clone the HELICS repository, you will be placed in the `main` branch by default.

To switch to a tagged release, you can type the following:

```
git checkout v3.7.0
```

You will not need a full understanding of how `git` works for installing HELICS, but if you are interested you can find a good `git` resource in this page.

Secondly, HELICS is a modern C++ cross-platform software application. One challenge while maintaining the same codebase across multiple operating systems is that we have to ensure that everything installs correctly everywhere. The development team uses `CMake` to build HELICS. `CMake` is a cross-platform tool designed to build, test and package software. Having the latest version of `CMake` can make the build process much smoother. `CMake` reads certain files (CMakeLists.txt) from the HELICS repository, and creates a bunch of build files. These build files specify how different files depend on each other and when these build files are run, HELICS is built. The exact instructions to run on each operating system are given in the individual installation instructions, but one important thing to remember is that these build files are essentially temporary files. If you have an issue building HELICS, once you make a change (installing/removing/adding anything), you probably need to delete these temporary files and re-generate them. We've

found in practice that you don't have to do this too often, but it can save hours of frustration if you are already aware that this needs to be done.

Another valuable piece of information about `CMake` is that almost every "OPTION" is configurable while you generate the build files. This means you can pass it configurations settings as a key value pair by adding `-D{NAME_OF_OPTION}={VALUE_OF_OPTION}` to the `cmake` command line interface. For example, to build the Java extension all you need to do is pass in `-DHELICS_BUILD_JAVA_INTERFACE=ON`. You can also run `ccmake .` in the build folder, to get a command line interactive prompt to change configuration settings. On Windows, you can use the cmake GUI to do the same. Again, there are more instructions in the individual installation pages but a useful trick to know if something isn't documented or a slightly more advanced feature is required. Available CMake options for HELICS are documented *here*.

2.2.4 Package Manager

Install using pip (Windows, macOS, Linux, other)

Install Python with pip. Upgrade pip to a recent version using `python -m pip install --upgrade`.

If you're on a supported version of Windows, macOS, or Linux (see the HELICS PyPI page for details) you can then use `pip` to install the HELICS Python interface and helics-apps.

```
pip install 'helics'
```

or

```
pip install 'helics[cli]'
```

The second version with the "[cli]" additionally installs additional tools that provides an easy method for launching co-simulations that is used in the *HELICS User Guide Examples* and is recommended.

If you are on an unsupported OS or CPU architecture, you may need to install a copy of HELICS first. Depending on your OS, there could be a copy in the package manager, or you may need to build HELICS from source. From there, you can use `pip install helics` as above. The source distributions section of the PyPI page has some additional useful information on this process.

Install using Spack (macOS, Linux)

Install Spack (a HELICS package is included in the Spack develop branch and Spack releases after v0.14.1).

Run the following command to install HELICS (this may take a while, Spack builds all dependencies from source!):

```
spack install helics
```

To get a list of installation options, run:

```
spack info helics
```

To enable or disable options, use +, -, and ~. For example, to build with MPI support on the command run would be:

```
spack install helics +mpi
```

Troubleshooting shared library errors on Windows

If you encounter an error along the lines of `DLL load failed: The specified module could not be found` when attempting to use the C shared library installed by a package manager, it is likely a required system dependency is missing. You can determine which DLL it is unable to find using a tool like https://github.com/lucasg/Dependencies to see what dependency is missing for the helics C shared library DLL. It is fine if it shows it can't find `WS2_32.dll`, but all other DLLs should be found. The most likely to be missing is `vcruntime140_1.dll`, which can be

fixed by downloading the latest Visual C++ Redistributable from https://support.microsoft.com/en-us/help/2977003/the-latest-supported-visual-c-downloads and installing it.

2.2.5 HELICS Installation Methods

The first step to using HELICS is to install it. Since there are several ways to do this, the flow chart below provides some insight into what approach is likely to be the easiest depending upon a number of factors, most predominantly the programming language bindings that you intend to use. Below the flow chart are links to more complete instructions for each method. Note that you'll need an internet connection for this process, as we'll be downloading HELICS from the internet.

As of HELICS v3, the only supported language bindings that are included with the core HELICS library downloads are C and C++98, in addition to the C++20 application API when building from source. If you end up needing to build from source AND use one of the supported language bindings you'll need to follow the instructions for installing HELICS for said language. This would also be the case if you were needing to run a co-simulation that used tools that provided their HELICS implementation in a variety of languages. Generally speaking, as long as all supported languages are on similar versions, each one can use its own installed version of HELICS without any trouble. The supported languages also have ways of being pointed towards a specific HELICS installation (rather than the one they install for themselves) if that is preferred or necessary for a particular use case.

Build from source

Build from source

Python via pip install

```
pip install helics
```

matHELICS install

Installation instructions are available in the matHELICS repository README

Julia install

```
pkg> add HELICS
```

pip install

`pip install 'helics[cli]'` (Includes the optional but recommended helics_cli tool.)

jHELICS

Build from source with the *CMAKE option* `HELICS_BUILD_JAVA_INTERFACE=ON`

nimble install

```
nimble install https://github.com/GMLC-TDC/helics.nim#head
```

C# install

Build from source with the *CMAKE option* `HELICS_BUILD_CSHARP_INTERFACE=ON`

Download pre-compiled

Download the pre-compiled libraries and add them to your system path

spack install

spack install helics

2.2.6 Running an Example

The *Quick Start guide* walks through the steps of running the first Python-based User Guide example and serves as a good way to test your (Python) installation.

2.2.7 HELICS runner

Previously a separate executable, `helics_cli` was used to provide functionality to launch a HELICS-based co-simulation by calling a JSON configuration such as

```
helics run --path=<path to HELICS runner JSON>
```

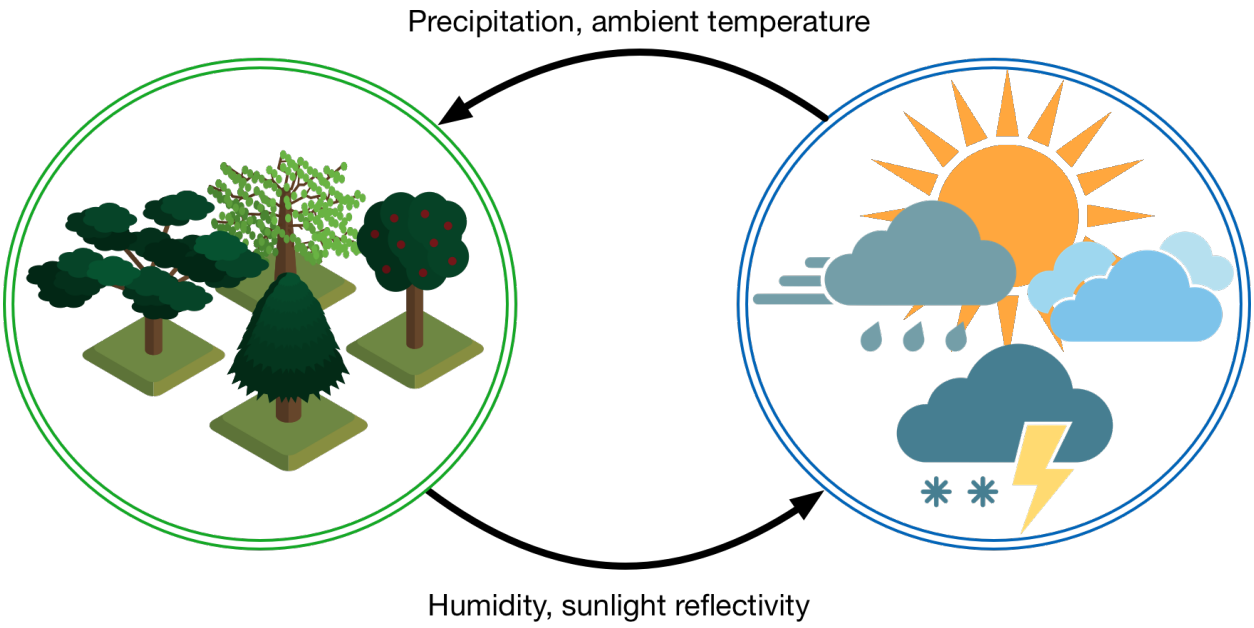
This functionality still exists but has been moved to the PyHELICS code base and the `helics_cli` repository has been deprecated. Thus, it is recommended that all users install PyHELICS (via `pip install 'helics[cli]'` as described above) to gain the runner and web interface functionality.

2.3 HELICS User Tutorial

2.3.1 Co-Simulation Overview

Co-simulation is an analysis technique that allows simulators from different domains to interact with each other, typically by exchanging values through the course of the simulation that define other simulators' boundary conditions. For example, you may have a model and simulator that is able to describe atmospheric conditions and weather as they progress through time. You may have just found out about another well-established model and simulator that describes the growth of vegetation over large geographic areas. In reality, these two systems clearly interact with the precipitation and temperatures impacting the growth of vegetation and the amount of vegetation impacting air temperature and moisture content. These models, though, may treat this interaction as a boundary condition such that the atmospheric simulator assumes a fixed rate of sunlight reflection from the ground and the vegetation simulator uses a time-series of historical precipitation values.

Co-simulation allows the coupling of existing models and simulators by breaking down these boundary condition barriers and calculating results that are more realistic and dynamic. In this case, while the atmospheric simulator is able to calculate the current weather using the latest values of surface reflectivity and humidity contributions from the vegetation, the vegetation simulator is able to use the latest values from the atmospheric simulator to determine the moisture content of the soil and the ambient temperature.



It's easy to imagine this hypothetical model being extended with other simulators throughout the ecosphere, incorporating models describing the behavior of the ocean, wildlife, even human agriculture. Rather than trying to build a single simulator that models all of this functionality, co-simulation allows us to use existing simulators that have longer-term investment and validation behind them; that experience is leveraged into building simulations of complex systems of systems.

Co-simulation also has a built-in parallelism that allows it to be used to scale up to very large models. Particularly for systems with limited interactions between portions of the models, the value exchanges that take place during co-simulation are limited, allowing the individual instances of the simulators to run independently. Using our previous example, it is easy to imagine that there is limited to no interaction between vegetation in Oregon and vegetation in Texas (except through the atmosphere). Simulations of the vegetations in each of these states can run independently due to this lack of interaction, each providing values to the atmospheric model that indirectly couples them together. When able to be constructed in such a manner, the limitations on the size and complexity of the co-simulation become limited more by computing resources than by the capabilities of the individual simulators.

This being said, coupling these simulators together effectively can be summarized in the execution of two very related functions:

1. Maintaining synchronization of all the simulator instances running
2. Facilitating the data transfer between them

Maintaining synchronization is required when working in a heterogeneous simulation environment where each simulator's concept of time is different and the computation time required by each simulator instance can vary widely. Without the regulation of the universal co-simulation time, individual simulator instances could easily run ahead of the rest, simulating days and weeks ahead of the others.

And when one simulator instance is simulating one week ahead of the others, it becomes impossible for the results from that simulation to affect the rest of the simulator instances; the values it passes and receives are literally from a different point in time than the rest and are effectively meaningless. Without the synchronization of time, there is no way for the simulator instances to affect each other and without this interaction, the results of the co-simulation are meaningless.

HELICS, at its core, has been designed to do these two functions as efficiently, quickly, and comprehensively as possible to support a wide variety of simulators and models. Simulators with HELICS support have been modified in such a way so as to allow HELICS (really, a HELICS core object but we'll get to that in the *section on timing*) to advance the time of the simulator and provide it new values for specific variables that it is interested in. Making this modification

to the simulator is not necessarily difficult but it must be done in such a way as to allow for these two fundamental functions to be executed.

Even after a simulator has been modified to support HELICS, users of that simulator who are building co-simulations have the task of constructing or designing a co-simulation to perform the particular analysis they desire. Many simulators have thousands of potential values they can provide to other simulators and generally, most of these values will not be needed by other simulators. It is the role of those designing the co-simulation to correctly configure each of the simulator instances such that the appropriate values are sent and received by the simulator instances and are mapped to the appropriate parameters inside each simulator. For a small number of simulators with a small number of values, this is not necessarily difficult but as the size of the co-simulation grows, this task becomes more challenging.

Thinking about HELICS co-simulation in general and about the nature of the simulators that have been or need to be integrated with HELICS, there are a few questions to consider. Though these questions have the most immediate impact as you begin planning how to integrate a particular simulator, they are also useful when thinking about how simulators in a federation need to be made to interact properly.

1. **What is the nature of the code-base being integrated?** Is this open-source code that can be fully modified? Is it a simulator, perhaps commercial, that provides an API that will be used? How much control do you, the integrator, have in modifying the behavior of the simulator? A number of existing simulation tools have *already been integrated into HELICS* and can serve as good references when considering how an integration might be accomplished.

2. **What programming language(s) will be used?** - HELICS has bindings for a number of languages and the one that is best to use may or may not be obvious. If your integration of the simulator will be through the API of the existing simulator, then you'll likely be writing a standalone executable that wraps that API. You may be constrained on the choice of languages based on the method of interaction with that API. If the API is accessed through a network socket then you likely have a lot of freedom in language choice. If the API is a library that you call from within wrapper, you will likely be best served using the language of that library.

   If you're writing your own simulator then you have a lot more freedom and the language you use may come down to personal preference and/or performance requirements of the federate.

   The languages currently supported by HELICS are:

   - C++
   - C
   - Python (2 and 3)
   - Java
   - MATLAB
   - Octave
   - C# (somewhat limited)
   - Julia
   - Nim

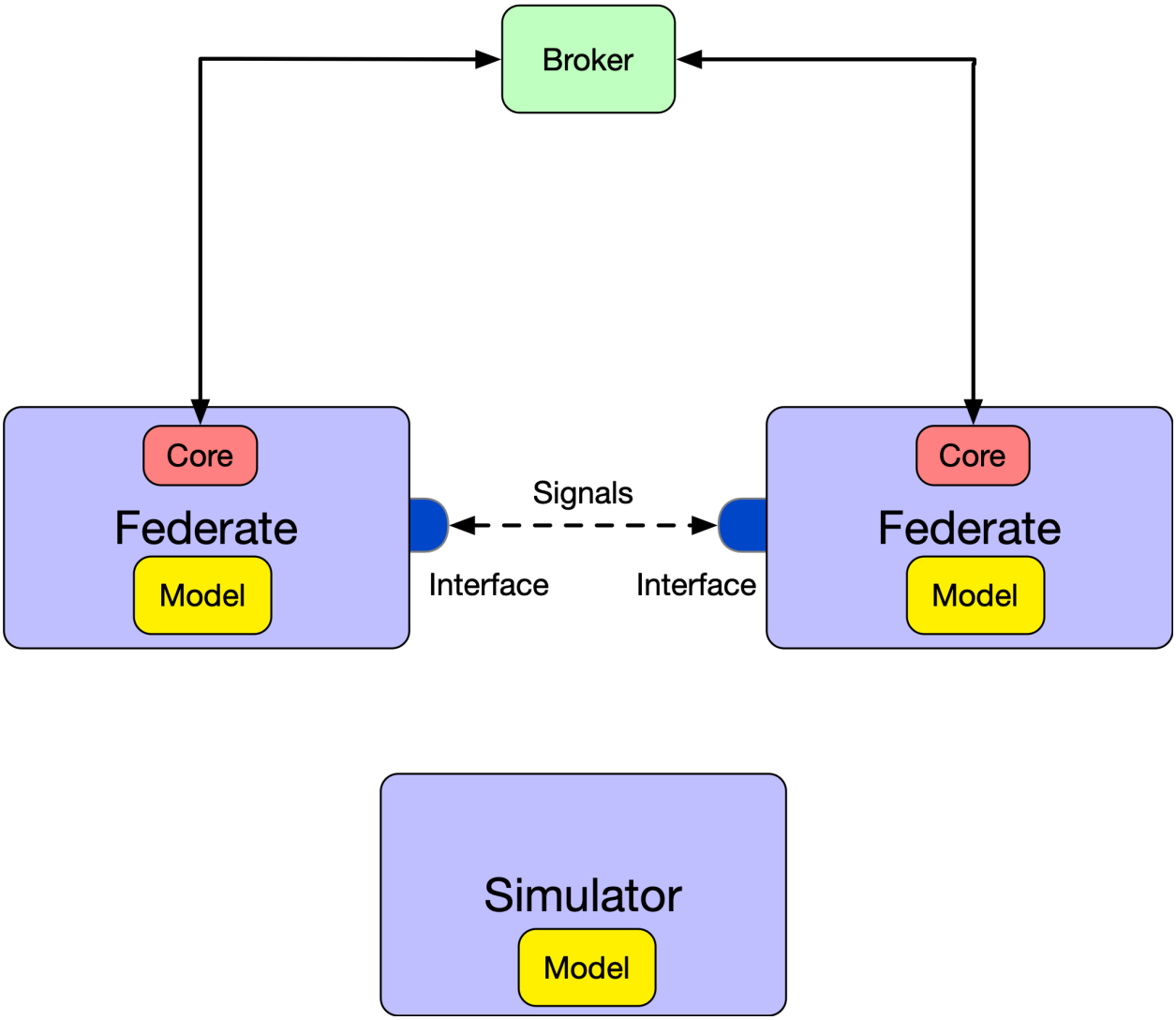
3. **What is the simulator's concept of time?** - Understanding how the simulator natively moves through time is essential when determining how time requests will need to be made. Does the simulator have a fixed time-step? Is it user-definable? Does the simulator have any concept of time or is it event-based?

4. **What is the nature of the data the simulator will send to and receive from the rest of the federation?** Often, this answer is in large part provided by the analysis need that is motivating the integration. However, there may be other angles to consider beyond what's immediately apparent. As a stand-alone simulator, what are its inputs and outputs? What are its assumed or provided boundary conditions? Where do interdependencies exist between the simulator and other simulators within the federation? What kinds of data will it be providing to the rest of the federation?

2.3.2 Fundamental Topics

HELICS Terminology

Before digging into the specifics of how a HELICS co-simulation runs, there are a number of key terms and concepts that need to be clarified first.



Simulator - A simulator is the executable that is able to perform some analysis function, often but not always, by solving specific problems to generate a time series of values.

- Simulators are abstract in the sense that it largely refers to the software in a non-executing state, outside of the co-simulation. We might say things like, "That simulator doesn't have feature xyz." or "This simulator has been parallelized and runs incredibly quickly on many-core computers."
- Any time we are talking about a specific instance of a simulator running a specific model you are really talking about a federate.
  - Simulators, on their own, do not have the ability to join a HELICS co-simulation but can still have a model associated them.

Federate - Federates are the running instances of simulators that have been assigned specific models and/or have specific values they are providing to and receiving from other federates.

- For example, we can have ten distribution circuit models that we are connecting for a co-simulation. Each could be run by the simulator GridLAB-D, and when they are running, they become ten unique federates providing unique values to each other.
- A collection of federates working together in a co-simulation is called a "federation."
  - Additional documentation can be found in *this tutorial's page on federates*.

Model - A model is the representation of some portion of reality being managed by a federate.

- A simulator contains the calculations for the model.
- Depending on the needs of the co-simulation, a federate can be configured to contain one or many models.
- For example, if we want to create a co-simulation of electric vehicles, we may write a simulator (executable program) to model the physics of the electric vehicle battery. We can then designate any number of agents/models of this battery by configuring the transfer of signals between the "Battery Federate" (which has N batteries modeled) and another federate.

Signals - Signals are the information passed between federates during the execution of the co-simulation. Fundamentally, co-simulation is about message-passing via these signals.

- We can notionally think of federates talking directly to each other by passing signals back and forth. Under the hood, the path the data takes is through the core and then broker.
- HELICS divides these messages into two types: value signals and message signal. The former is used when coupling two federates that share physics (*e.g.* batteries providing power to wheel motors on an electric car) and the later is used to couple two federates with information (*e.g.* a battery charge controller and a charge relay on a battery).
- There are also a variety of mechanisms within a co-simulation to define the nature of the data being exchanged (data type, for example) and how the data is distributed around the federation.

Interface - An object by which a federate pass signals to each other.

- Includes Endpoints, Publications, and Inputs.
- Additional documentation on interfaces can be found in *this tutorial's page on federates*.

Core - The core is the software that has been embedded inside a simulator to allow it to join a HELICS federation. Simulators without a core are not HELICS federates.

- In general, each federate has a single core, making the two synonymous (core <=> federate).
- The two most common configurations are: (1) one core, one federate, one model; (2) one core, one federate, multiple models.
- There are sometimes cases where a single executable is used to represent multiple federates and all of those federates use a single core (one core, multiple federates, multiple models).
- Cores are built around specific messaging protocols with HELICS supporting a number of different protocols (*e.g.* ZMQ, TCP, MPI). Selection of the messaging protocol is part of the configuration process required to form the federation. Additional information about cores can be found in the *Advanced Topics*.

Broker - The broker is a special executable distributed with HELICS; it is responsible for facilitating signal passing between federates.

- Each core (federate) must connect to a broker to be part of the federation.
- Brokers receive and distribute messages from any federates that are connected to it, routing them to the appropriate location.
- HELICS also supports a hierarchy of brokers, allowing brokers to pass messages between each other to connect federates associated with different brokers and thus maintain the integrity of the federation. The broker at the

top of the hierarchy is called the "root broker" and it is the message router of last resort. *Additional information about broker hierarchies can be found here*.

Federates

Value Federates

HELICS messages that are value-oriented are the most common type of messages. As mentioned in the *federate introduction*, value messages are intended to be used to represent the physics of a system, linking federates at their mutual boundaries and allowing a larger and more complex system to be represented than would be the case if only one simulator was used.

Value Federate Interface Types

There are four interface types for value federates that allow the interactions between the federates (a large part of co-simulation/federation configuration) to be flexibly defined. The difference between the four types revolve around whether the interface is for sending or receiving HELICS messages and whether the sender/receiver is defined by the federate (technically, the core associated with the federate):

- **Publications** - Sending interface where the federate core does not specify the intended recipient of the HELICS message
- **Named Inputs** - Receiving interface where the federate core does not specify the source federate of the HELICS message
- **Directed Outputs** - Sending interface where the federate core specifies the recipient of HELICS message
- **Subscriptions** - Receiving interface where the federate core specifies the sender of HELICS message

In all cases the configuration of the federate core declares the existence of the interface to use for communicating with other federates. The difference between "publication"/"named inputs" and "directed outputs"/"subscriptions" is where that federate core itself knows the specific names of the interfaces on the receiving/sending federate core.

The message type used for a given federation configuration is often an expression of the preference of the user setting up the federation. There are a few important differences that may guide which interfaces to use:

- **Which interfaces does the simulator support?** - Though it is the preference of the HELICS development team that all integrated simulators support all four types, that may not be the case. Limitations of the simulator may limit your options as a user of that simulator.
- **Is portability of the federate and its configuration important?** - Because "publications" and "named inputs" don't require the federate to know who it is sending HELICS messages to and receiving HELICS messages from as part of the federate configuration, it affords a slightly higher degree of portability between different federations. The mapping of HELICS messages still needs to be done to configure a federation, its just done separately from the federate configuration file via a broker or core configuration file. The difference in location of this mapping may offer some configuration efficiencies in some circumstances.

Though all four message types are supported, the remainder of this guide will focus on publications and subscriptions as they are conceptually easily understood and can be comprehensively configured through the individual federate configuration files.

Message Federates

As previously discussed in the *federate introduction*, message federates are used to create HELICS messages that model information transfers (versus physical values) moving between federates. Measurement and control signals are typical applications for these types of federates.

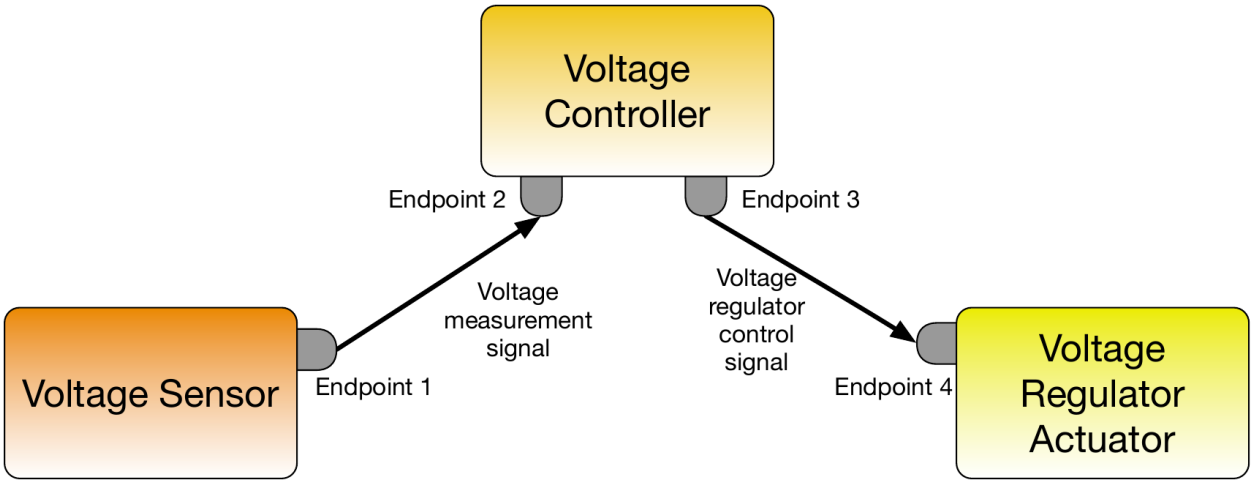
Unlike HELICS values which are persistent (meaning they are continuously available throughout the co-simulation), HELICS messages are only readable once when collected from an endpoint. Once that collection is made the message

only exists within the memory of the collecting message federate. If another message federate needs the information, a new message must be created and sent to the appropriate endpoint. Filters can be created to clone messages as well if that behavior is desired.

Message Federate Endpoints

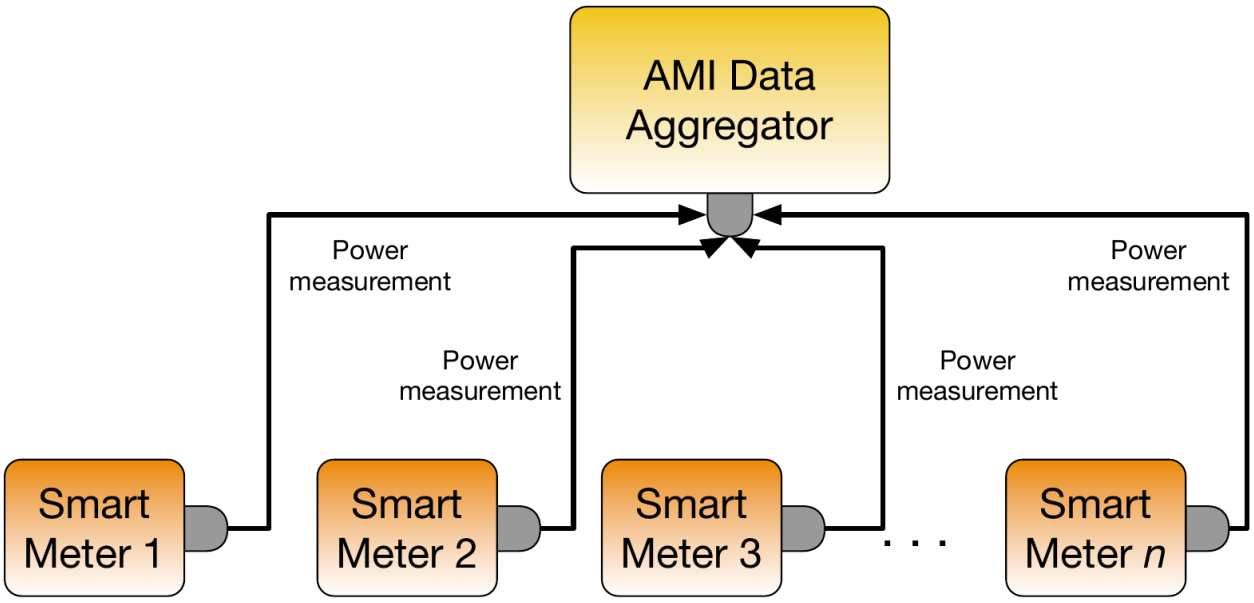
As previously discussed, message federates interact with the federation by defining an "endpoint" that acts as their address to send and receive messages. Message federates are typically sending and receiving measurements, control signals, commands, and other signal data with HELICS acting as a perfect communication system (infinite bandwidth, virtually no latency, guaranteed delivery).

In fact, as you'll see in *a later section*, it is possible to create more realistic communication-system effects natively in HELICS (as well as use a full-blown communication simulator like ns-3 to do the same). This is relevant now, though, because it influences how the endpoints are created and, as a consequence, how the simulator handles messages. You could, for example, have a system with three federates communicating with each other: a remote voltage sensor, a voltage controller, and a voltage regulation actuator (we'll pretend for the case of this example that the last two are physically separated though they often aren't). In this case, you could imagine that the voltage sensor only sends messages to the voltage controller and the voltage controller only sends messages to the voltage regulation actuator. That is, those two paths between the three entities are distinct, have no interaction, and have unique properties (though they may not be modeled). Given this, referring to the figure below, the voltage sensor could have one endpoint ("Endpoint 1") to send the voltage signal, the voltage regulator could receive the measurement at one endpoint ("Endpoint 2") and send the control signal on another ("Endpoint 3"), and the voltage regulation actuator would receive the control signal on its own endpoint ("Endpoint 4").



The federate code handling these messages can be relatively simple because the data coming in or going out of each endpoint is unique. The voltage controller always receives (and only receives) the voltage measurement at one endpoint and similarly only sends the control signal on another.

Consider a counter-example: automated meter-reading (AMI) using a wireless network that connects all meters in a distribution system to a data-aggregator in the substation (where, presumably, the data travels over a dedicated wired connection to a control room). All meters will have a single endpoint over which they will send their data but what about the receiver? The co-simulation could be designed with the data-aggregator having a unique endpoint for each meter but this implies come kind of dedicated communication channel for each meter; this is not the case with wireless communication. Instead, it is probably better to create a single endpoint representing the wireless connection the data aggregator has with the AMI network. In this case, messages from any of the meters in the system will be flowing through the same endpoint and to differentiate the messages from each other, the federate will have to be written to examine the metadata with the message to determine its original source.



Interactions Between Messages and Values

Though it is not possible to have a HELICS message show up at a value interface, the converse is possible; message_federates can subscribe to HELICS values. Every time a value federate publishes a new value to the federation, if a message federate has subscribed to that message HELICS will generate a new HELICS message and send it directly to the destination endpoint. These messages are queued and not overwritten (unlike in HELICS values) which means when a message federate is synchronized it may have multiple messages from the same source to manage.

This feature offers the convenience of allowing a message federate to receive messages from pure value federates that have no endpoints defined. This is particularly useful for simulators that do not support endpoints but are required to provide measurement signals controllers. Implemented in this way, though, it is not possible to later implement a full-blown communication simulator that these values-turned-messages can traverse. Such co-simulation architectures in HELICS require the existence of both a sending and receiving endpoint; this feature very explicitly bypasses the need for a sending endpoint.

Filters

As was introduced in the *introductory section on federates*, message federates (and combo federates) are used to send messages (control signals, measurements, anything traveling over some kind of communication network) via HELICS to other federates. Though they seem very similar, the way messages and values are handled by HELICS is very different and is motivated by the underlying reality they are being used to model.

1. **Messages are directed and unique, values are persistent.** - Because HELICS values are used to represent physical reality, they are available to any subscribing federate at any time. If the publishing federate only updates the value, say, once every minute, any subscribing federates that are granted a time during that minute window will all receive the same value regardless of when they requested it.

   HELICS messages, though, are much more like other kinds of real-world messages in that they are directed and unique. Messages are sent by a federate from one endpoint to another endpoint in the federation (presumably the receiving endpoint is owned by another federate but this doesn't have to be the case). Internal to HELICS, each message has a unique identifier and can be thought to travel between a generic communication system between the source and destination endpoints.

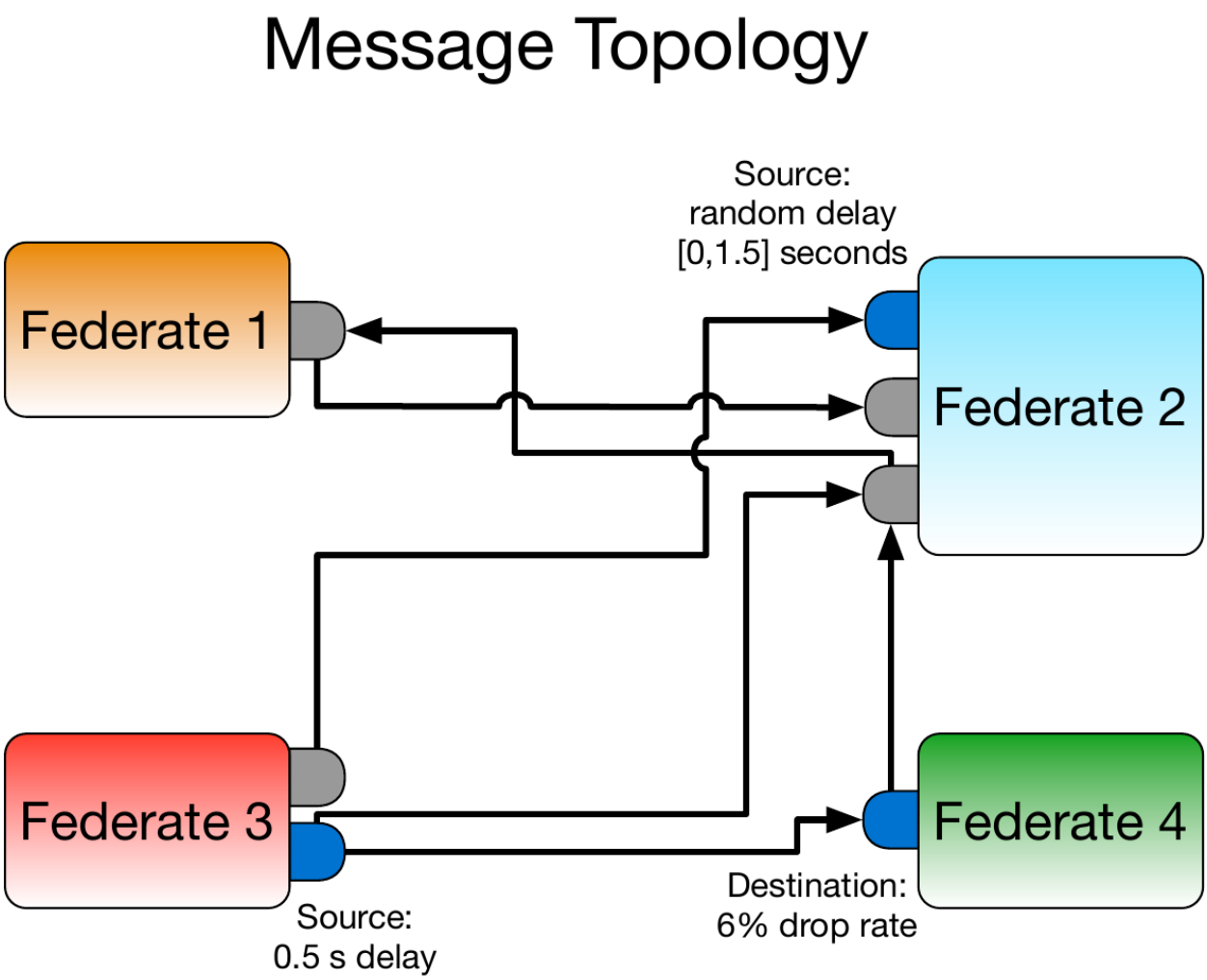
2. **Messages can be filtered, values cannot.** - By creating a generic communication system between two endpoints, HELICS has the ability to model simple communication system effects on messages traveling through said network. These effects are called "filters" and are associated with the HELICS core (which in turn manages

the federate's endpoints) embedded with the federate in question. Typical filtering actions might be delaying the transmission of the message or randomly dropping a certain percentage of the received messages. Filters can also be defined to operate on messages being sent ("source filters") and/or messages being received ("destination filters").

Because HELICS values do not pass through this generic network, they cannot be operated on by filters. Since HELICS values are used to represent physics of the system not the control and coordination of it, it is appropriate that filters not be available to modify them. It is entirely possible to use HELICS value federates to send control signals to other federates; co-simulations can and have been made to work in such ways. Doing so, though, cuts out the possibility of using filters and, as we'll see, the easy integration of communication system simulators.

The figure below is an example of a representation of the message topology of a generic co-simulation federation composed entirely of message federates. Source and destination filters have been implemented (indicated by the blue endpoints), each showing a different built-in HELICS filter function.

- As a reminder, a single endpoint can be used to both send and receive messages (as shown by Federate 4). Both a source filter and a destination filter can be set up on a single endpoint. In fact multiple source filters can be used on the same endpoint.
- The source filter on Federate 3 delays the messages to both Federate 2 and Federate 4 by the same 0.5 seconds. Without establishing a separate endpoint devoted to each federate, there is no way to produce different delays in the messages sent along these two paths.
- Because the filter on Federate 4 is a destination filter, the message it receives from Federate 3 is affected by the filter but the message it sends to Federate 2 is not affected.
- As constructed, the source filter on Federate 2 has no impact on this co-simulation as there are no messages sent from that endpoint.
- Individual filters can be targeted to act on multiple endpoints and act as both source and destination filters.



This section on Federates covers:

- *What is a Federate?*
- *Types of Federates*
  - *Value Federates*
    - *Value Information*
    - *Value Federate Interfaces*
  - *Message Federates*
    - *Message Information*
    - *Message Federate Interfaces*
      - *Endpoints*
      - *Native HELICS Filters*
    - *Interactions Between Messages and Values*
  - *Combination Federates*

What is a Federate?

A "federate" is an instance of a simulation executable that models a group of objects or an individual objects. For example, one can write a simulator to model the battery of an electric vehicle (EV). If we want to model multiple EVs connected to charging ports in a dedicated EV charging garage, we can use the EV simulator to model a group of EVs. We will need a second simulator to model a group of charging ports. Once we launch the simulators, each is called a "federate". Together, they are called a "federation" – multiple federates running simultaneously. This federation performs a co-simulation to achieve a particular analysis objective (*e.g.* replicate the behavior of a fleet of EVs charging to understand the charging power requirements).

This co-simulation is described in more detail in the *Fundamental Examples*. There are two federates in this co-simulation; one modeling the batteries on board the EVs, and one modeling the chargers of the batteries. Each federate in this example has multiple objects it is modeling; five batteries for the battery federate, and five chargers for the charger federate. Because the objects being modeled with the federates share most of the same properties – *e.g.* battery size, charge rate – a single federate can be used to model the five batteries. As the complexity of the co-simulation increases, it becomes increasingly difficult to group objects into one federate. In these situations, we could also design the co-simulation with one federate for each EV.

Co-simulations are designed to answer a research question. The question addressed by the *Fundamental Examples* is: How much power is needed to serve five EVs in a dedicated charging garage?

With this research question, we have identified that we want to model **batteries** and **chargers** and we want to monitor the power draw in **kW** over time. It's important to first identify the types of objects you want to model, as co-simulation in HELICS requires constructing federates based on the type of information they pass to other federates.

Types of Federates

Federates are distinguished by the types of information they model and the interfaces they use to pass the information. Interfaces define how signals are connected between federates in a federation. HELICS has three types of federates: **Value Federates, Message Federates, and Combination Federates**. Value federates model physics in a system, message federates model logic (i.e., controls), and combination federates model both.

Value Federates

Value federates are used when the federate is simulating the physics of a system. The data in the messages they send and receive indicate new values at the boundary of the federate system under simulation. Value federates interface with the federation using one-to-one correspondence to internal variables within the federate. These interfaces are commonly called publications and subscriptions (pubs/subs).

Value Information

The information modeled by value federates is physical values in a system with associated units. In the *Fundamental Example*, the batteries and chargers are both value federates; charger applies a voltage to the battery (Charger federate sending the Battery federate that value) and the Battery federate responds with the charging current which it sends back to the Charger. These two federates each update the other's boundary condition's state: the voltage and current. Federates that model physics must be configured as **value federates**. Value federates typically will update at very regular intervals based on the fidelity of their models and/or the resolution of any supporting data they are using.

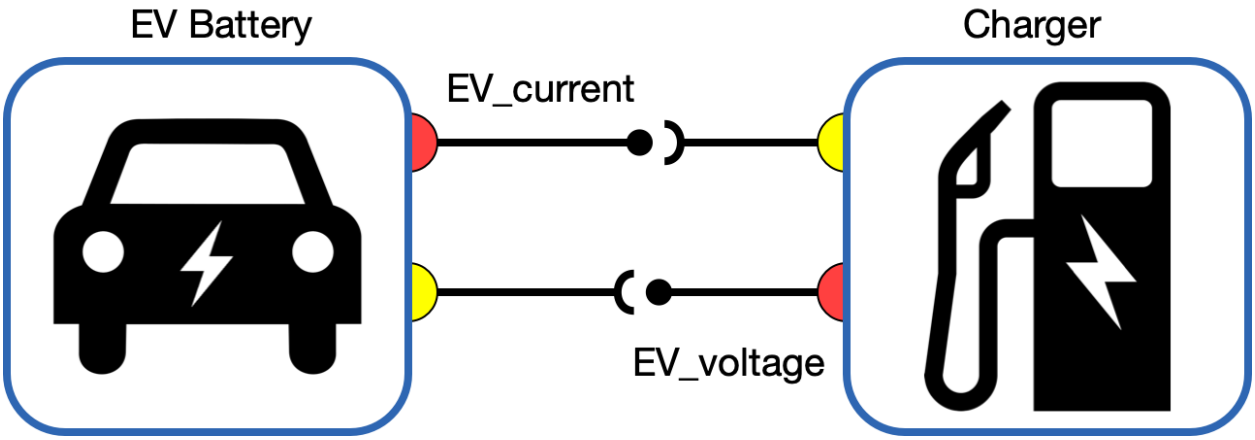
Value Federate Interfaces

Value federates have direct fixed connections through interfaces to other federates. There are three interface types for value federates that allow the interactions between the federates to be flexibly defined. The difference between the three types revolves around whether the interface is for sending or receiving values and whether the sender/receiver is defined by the federate:

- Publications

  - Sending interface
  - Interface named with `"key"` in configuration
  - Recipient interface value is not necessary, however it can be specified with `"targets"` in configuration
- Subscriptions (Unnamed Inputs)
  - Receiving interface
  - Not "named" (no identifier to the rest of the federation)
  - Source of value interface is specified with `"key"` in configuration
- Named Inputs
  - Receiving interface
  - Interface named with `"key"` in configuration
  - Source interface of value is not necessary, however it can be specified with `"targets"` – Named Inputs can receive values from multiple `"targets"`

The most commonly used of these fixed interfaces are publications and subscriptions. In the *Fundamental Example*, the Battery federate and the Charger federate have fixed pub/sub connections. In the figure below, publishing interfaces are in **red** and the subscription interfaces are in **yellow**. The Battery federate **publishes** the current flowing into the battery from the publication interface named `EV_Battery/EV_current` and does not specify the intended recipient. The Charger federate **subscribes** to the amps from the Battery with the subscription interface named `EV_Battery/EV_current` – the receiving interface only specifies the sender.



In all cases the configuration of the federate core declares the existence of the interface to use for communicating with other federates. The difference between publication/subscription and directed outputs/unnamed inputs is where the federate core knows the specific names of the interfaces on the receiving/sending federate core.

The interface type used for a federation configuration is a preference of the user setting up the federation. There are a few important differences that may guide which interfaces to use:

- Which interfaces does the simulator support?
  - Though it is the preference of the HELICS development team that all integrated simulators support all types, that may not be the case or even possible. Limitations of the simulator may limit your options as a user of that simulator.
- Is portability of the federate and its configuration important?
  - Because publications and subscriptions (unnamed inputs) don't require the federate to know who it is sending HELICS messages to and receiving HELICS messages from, it affords a slightly higher degree of portability between different federations. The mapping of HELICS messages still needs to be done to

configure a federation, it's just done separately from the federate configuration file via a broker or core configuration file. The difference in location of this mapping may offer some configuration efficiencies in some circumstances.

Message Federates

Message federates send packets of data with unfixed connections for things such as events, communication packets, or triggers. Message federates are used to model information transfers (versus physical values) moving between federates. Measurement and control signals are typical applications for these types of federates.

Message Information

Message federates are used when the HELICS signals being passed to and from the simulation are generic packets of information, often for control purposes. They are treated as data to be used by an algorithm, processor, or controller. If the inputs to the federate can be thought of as traveling over a communication network of some kind, it should be modeled as coming from/going to a message federate. For example, in the power system world, phasor measurement units (PMUs) have been installed throughout the power system and the measurements they make are collected through a communication system and would be best modeled through the use of HELICS messages.

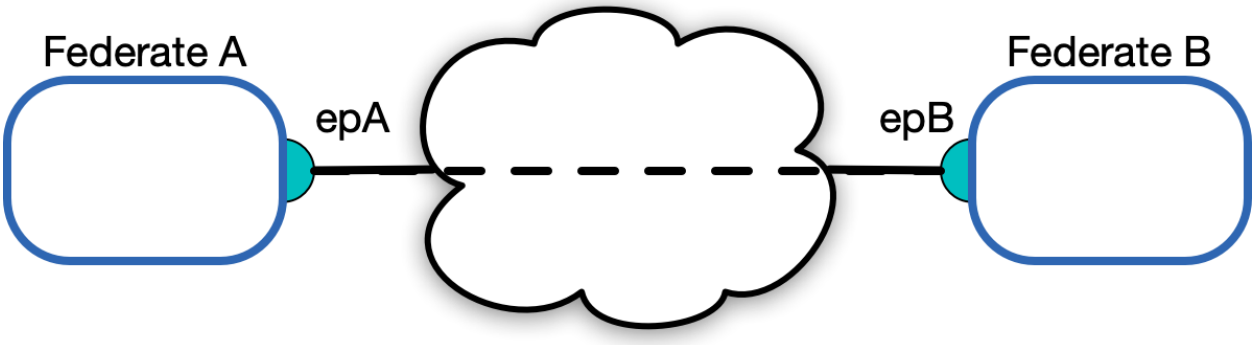
Message Federate Interfaces

Message federates interact with the federation through endpoints interfaces. Message federates can be thought of as attaching to communication networks, where the federate's **endpoints** are the specific interfaces to that communication network. By default, HELICS acts as the communication network, transferring messages between endpoint interfaces configured for message federates. Just as communications networks can be susceptible to failure, messages can be altered or delayed with **filter** which can be associated with specific endpoints. Filters can only act on messages and thus can only be associated with endpoints. Value signals are meant to replicate physical connections between models and thus are not susceptible to the frailty of communication systems.

Endpoints

Endpoints are interfaces used to pass packetized data blocks (messages) to another federate. Message federates interact with the federation by defining an endpoint that acts as their address to send and receive messages. Message federates are typically sending and receiving measurements, control signals, commands, and other signal data with HELICS acting as a perfect communication system (infinite bandwidth, no latency, guaranteed delivery).

In the figure below, Federate A and B are message federates with endpoints epA and epB. They do not have a fixed communication pathway; they have unique addresses (endpoints) to which messages can be sent. An endpoint can send data to any other endpoint in the system – it just needs the "address" (endpoint interface).



Endpoints can have a `"type"` which is a user defined string. HELICS currently does not recognize any predefined types. The data consists of raw binary data and optionally a send time. Messages are delivered first by time order, then

federate ID number, then interface ID, then by order of arrival.

Unlike HELICS values which are persistent (meaning they are continuously available throughout the co-simulation), HELICS messages are only readable once when collected from an endpoint. Once that collection is made, the message only exists within the memory of the collecting message federate. If another message federate needs the information, a new message must be created and sent to the appropriate endpoint.

Native HELICS Filters

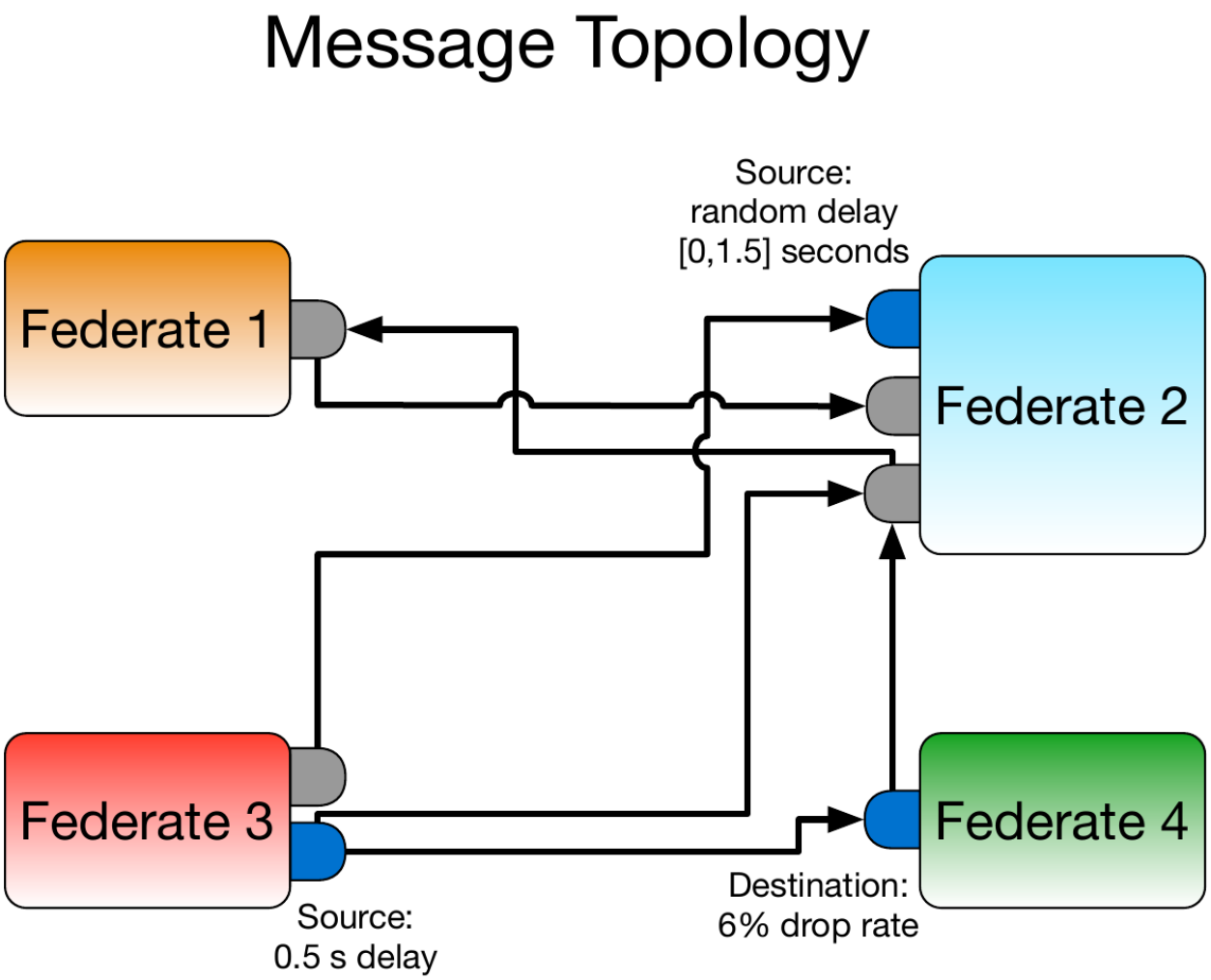
Filters are objects that can be used to disrupt message packets in a manner similar to communications networks. Filters are associated with the HELICS core, which in turn manages a federate's endpoints. Typical filtering actions might be delaying the transmission of a message or randomly dropping a certain percentage of the received messages. Filters can also be defined to operate on messages being sent ("source filters") and/or messages being received ("destination filters").

Messages can be filtered, values cannot. Messages are directed and unique, values are persistent. Internal to HELICS, each message has a unique identifier and can be thought to travel through a generic communication system between the source and destination endpoints. Since HELICS values model direct physical connections, they do not pass through this generic network and they cannot be operated on by filters. It is possible to create a federation where HELICS value interfaces are used to send control signals but this removes the possibility of using filters and the easy integration of communication system simulators.

Filters have the following properties:

1. Inline operations that can alter a message or events
    - Time Delay (Random or Fixed)
    - Packet Translation
    - Random Dropping
    - Message Cloning / Replication
    - Rerouting
    - Firewall
    - Custom
2. Filters are part of the HELICS core and the effect of a filter is not limited to the endpoints of local objects
3. A single Filter can be configured once and then applied to multiple endpoints as unique instances of that configuration. Filters can be triggered by either messages sent from an endpoint (source target) or messages received by an endpoint (destination targets)
4. Filters can be cloning or non-cloning filters. Cloning filters will operate on a copy of the message and in the simple form just deliver a copy to the specified destination locations. The original message gets delivered as it would have without the filter.

The figure below is an example of a representation of the message topology of a generic co-simulation federation composed entirely of message federates. Source and destination filters have been implemented (indicated by the blue endpoints – gray endpoints do not have filters), each showing a different built-in HELICS filter function.



- In this figure, Federate 4 has a single endpoint for sending and receiving messages. Both a source filter and a destination filter can be set up on a single endpoint, or multiple source filters can be used on the same endpoint.
- The source filter on Federate 3 delays the messages to both Federate 2 and Federate 4 by 0.5 seconds. Without establishing a separate source endpoint devoted to each federate, there is no way to produce different delays in the messages sent along these two paths.
- Because the filter on Federate 4 is a destination filter, the message it receives from Federate 3 is affected by the filter but the message it sends to Federate 2 is not affected.
- The source filter on Federate 2 has no impact on this co-simulation as there are no messages sent from that endpoint.

Interactions Between Messages and Values

Because HELICS values are used to represent physical reality, they are available to any subscribing federate at any time. If the publishing federate only updates the value, say, once every minute, any subscribing federates that are granted a time during that minute window will all receive the same value.

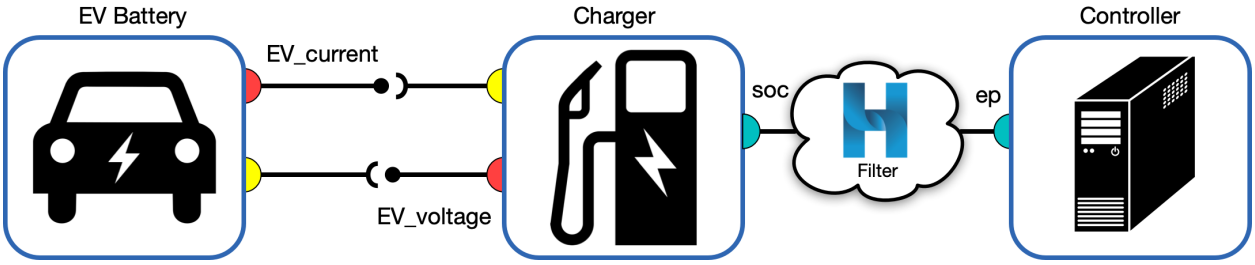
Though it is not possible to have a HELICS message show up at a value interface, the converse is possible; message federates can subscribe to HELICS values. Every time a value federate publishes a new value to the federation, if a message federate has configured an endpoint with a `"subscription"` parameter defined, HELICS will generate a new HELICS message and send it directly to the subscribing endpoint every time a new value is published. These messages

are queued and not overwritten (unlike in HELICS values) which means when a message federate is granted a time, it may have multiple messages from the same source to process.

This feature offers the convenience of allowing a message federate to receive messages from pure value federates that have no endpoints defined. This is particularly useful for simulators that do not support endpoints but are required to provide measurement signals for controllers. Implemented in this way, though, it is not possible to later implement a full-blown communication simulator that these values-turned-messages can traverse. Such co-simulation architectures in HELICS require the existence of both a sending and receiving endpoint; this feature very explicitly bypasses the need for a sending endpoint.

Combination Federates

Combination federates make use of both value signals and message signals for transferring data between federates. The *Combination Federation* in the Fundamental Examples learning track introduces a third federate to the *Base Example* – the combination federate passes values with the battery federate to monitor the physics of the battery charging, and it also passes messages with a controller federate to decide when to stop charging.



The following table may be useful in understanding the differences between the two methods by which federates can communicate:

	Values	Messages
Interface Type:	Publication/Subscription/Input	Endpoint/Filter
Signal Route:	Fixed, defined at initialization	Determined at time of transmission
Outgoing signal:	1 to n (broadcast)	1 to 1 - defined sender and receiver
Incoming signal:	n to 1 (promiscuous)	1 to 1 - defined sender and receiver
Status on Message Bus:	Current value always available	Removed when received
Fidelity:	Default value	Rerouting/modification through filters
Signal Contents:	Physical units	Generic binary blobs

Federate Interface Configuration

As soon as one particular instance of a simulator begins running in a co-simulation it is a federate. Every federate will require configuration of the way it will communicate (send signals) to other federates in the federation. For *simulators that already have HELICS support*, the configuration takes the form of a JSON (or TOML) file; bespoke simulators can be configured with the HELICS APIs in the code or via a JSON file. The essential information required to configure federate interfaces with HELICS is:

Federate name - The unique name this federate will be known as throughout the federation. It is essential this name is unique so that HELICS messages can route properly.

Core type - The core manages interfaces between the federation and the federate; there are several messaging technologies supported by HELICS.

Publications and Inputs - Publication configuration contains a listing of source interface, data types, and units being sent by the federate; input configuration does the same for values being received by the federate. If supported by

the simulator (e.g., *a Python simulator*), these values can be mapped to internal variables of the simulator from the configuration file.

Endpoints - Endpoints are sending and receiving points for HELICS messages to and from message federates. They are declared and defined for each federate.

Time step size - This value defines the resolution of the simulator to prevent HELICS from telling the simulator to step to a time of which it has no concept (e.g. trying to simulate the time of 1.5 seconds when the simulator has a resolution of one second).

This section describes how to configure the federate interfaces using JSON files and API calls. Extensive details on the options for configuring HELICS federates is available in the *Configurations Options Reference*. If the user has written the simulator, it may be preferable to use the HELICS APIs to configure the federates, because the interface registrations can be made into a variable of the simulation. For non-open-source simulators, the JSON configuration files must be written before the federation is launched.

- *JSON Configuration*
- *API Configuration*

JSON Configuration

Federate interfaces must be configured with JSON files if they are built from non-open-source tools, such as the simulators listed in the reference page on *Tools with HELICS Support*. Federates built from simulators written by the user can be configured with JSON files or API calls.

The *Examples* illustrate in detail how to integrate federates built from open source tools, such as Python. The *Fundamental Combination Federation* configures the Python federate interfaces with JSON files. In this co-simulation, there are three federates: a value Battery federate, a combination Charger federate, and a message Controller federate. The example JSON shows the interface configuration for the combination federate to illustrate the different types of interfaces.

Sample JSON configuration file

Below is a sample JSON configuration file with some of the more common options. There are many, many more configuration parameters that this file could include; a relatively comprehensive list along with explanations of the functionality they provide can be found in the *Configurations Options Reference*.

```
{
  "name": "sample_federate",
  "loglevel": "debug",
  "coreType": "zmq",
  "period": 60,
  "offset": 10,
  "uninterruptible": false,
  "terminate_on_error": true,
  "wait_for_current_time_update": false,
  "federate_init_string": "--broker_address=127.0.0.1 --port=23405"
  "endpoints": [
    {
      "name": "sample_federate/ep1",
      "destination": "other_federate/ep1",
      "global": true
    }
  ],
  "publications":[
```

(continues on next page)

(continued from previous page)

```
    {
      "key":"sample_federate/voltage",
      "type":"double",
      "unit":"V",
      "global": true,
      "only_transmit_on_change": true,
      "tolerance": 0.1,
      "tags": {
        "period": 0.5,
        "description": "a test publication"
      }
    }
  ],
  "subscriptions":[
    {
      "key":"other_federate/current",
      "type":"double",
      "unit":"A",
      "global": true,
      "only_update_on_change": true,
      "tolerance": 0.2,
      "default":0.91
    }
  ],
  "inputs": [
    {
      "key": "ipt2",
      "type": "double",
      "connection_required": true,
      "target": "pub1",
      "global": true,
      "default":"3.67",
      "tags": [
        { "name": "period", "value": "0.7" },
        { "name": "description", "value": "a test input" }
      ]
    }
    //specify an input with a target multiple targets could be specified like "targets":[
↪"pub1","pub2","pub3"]
  ],
}
```

JSON configuration file explanation

- **name** - Every federate must have a unique name across the entire federation; this is functionally the address of the federate and is used to determine where HELICS messages are sent. An error will be generated if the federate name is not unique.
- **loglevel** - The level of detail exported to the *log files* during run time ranges from "none" to "trace".
- **coreType** - There are a number of technologies or message buses that can be used to send HELICS messages among federates, detailed in *Core Types*. Every HELICS enabled simulator has code in it that creates a core which connects to a HELICS broker using one of these messaging technologies. ZeroMQ (zmq) is the default core type

and most commonly used, but there are also cores that use TCP and UDP networking protocols directly (forgoing ZMQ's guarantee of delivery and reconnection functions), IPC (uses Boost's interprocess communication for fast in-memory message-passing but only works if all federates are running on the same physical computer), and MPI (for use on high-performance computing clusters where MPI is installed).

- **period** and **offset** - The federate needs instruction for how to step forward in time in order to synchronize calculations. This is the simplest way to synchronize simulators to the same time step; this forces the federate to time step intervals of `n*period + offset`. The default units are in seconds. Timing configuration is explained in greater detail in the *Timing* page, with additional configuration options in the *Configuration Options Reference*.
- **uninterruptible** - Setting `uninterruptible` to `false` allows the federate to be interrupted if there is a signal available for it to receive. This is a *timing configuration* option.
- **terminate_on_error** - By default, HELICS will not terminate execution of every participating federate if an error occurs in one. However, in most cases, if such an error occurs, the co-simulation is no longer valid. Setting `terminate_on_error` frees the federate from the broker if there is an error in execution, which simplifies debugging. This will prevent your federate from hanging in the event that another federate fails.
- **wait_for_current_time_update** - There are times when HELICS will grant the same simulated time to a number of federates simultaneously. There is a possibility of this leading to unexpected co-simulation results if federates are unexpectedly operating on old data. Using this flag, HELICS uses this option to provide the ability for one federate to always be granted this time last, after all other federates that have been granted this time have requested a later time. This ensures that the federate with this flag set will have all the latest information from all other federates before it begins execution at the granted time.
- **federate_init_string** - This option provides a way of passing in a large number of configuration options that a federate needs during initialization. You can consult the *Configuration Options Reference* page for a more complete list but there are a few worth bringing up specifically. `--broker_address=<IP address>` and `--port=<port number>`- Allows you to specify the IP address and port number of the broker to which you want this federate to connect. You can consult the *Advanced Topics section of the User Guide* to see further explanation of how to handle more complex broker configuration.
- **endpoints**
  - `name` - The string in this field is the unique identifier for the endpoint interface.
  - `destination` - This option can be used to set a default destination for the messages sent from this endpoint. The default destination is allowed to be rerouted or changed during run time.
  - `global` - Just as in value federates, `global` allows for the identifier of the endpoint to be declared unique for the entire federation.
- **publications**
  - `key` - The string in this field is the unique identifier (at the federate level) for the value that will be published to the federation. In the example above, `global` is set to `true`, meaning the `key` must be unique to the entire federation.
  - `global` - Indicates that the value in `key` will be used as a global name when other federates are subscribing to the message. This requires that the user ensure that the name is used only once across all federates. Setting `global` to `true` is handy for federations with a small number of federates and a small number of message exchanges as it allows the `key` string to be short and simple. For larger federations, it is likely to be easier to set the flag to `false`.
  - `required` - At least one federate must subscribe to the publications.
  - `type` - Data type, such as integer, double, complex.
  - `units` - The units can be any sort of unit string, a wide assortment is supported and can be compound units such as m/s^2 and the conversion will convert as long as things are convertible. The unit match is also checked for other types and an error if mismatching units are detected. A warning is also generated if

the units are not understood and not matching. The unit checking and conversion is only active if both the publication and subscription specify units. HELICS is able to do some levels of unit conversion, currently only on double type publications but more may be added in the future.

  - `only_transmit_on_change` and `tolerance` - Publications will only send a new value out to the federation when the value has changed more than the delta specified by `tolerance`.
  - `alias` - an alternate name for the publication must be globally unique for publications
  - `tags` - Arbitrary string value pairs that can be applied to interfaces. Tags are available to others through queries but are not transmitted by default. They can be used to store additional information about an interface that might be useful to applications. At some point in the future automated connection routines will make use of them. "tags" are applicable to any interface and can also be used on federates.

- **`subscriptions`** - These are lists of the values being sent to and from the given federate.
  - `key` - This string identifies the federation-unique value that this federate wishes to receive. If `global` has been set to `false` in the `publications` JSON configuration file, the name of the value is formatted as `<federate name>/<publication key>`. Both of these strings can be found in the publishing federate's JSON configuration file as the `name` and `key` strings, respectively. If `global` is `true` the string is the publishing federate's `key` value.
  - `required` - The message being subscribed to must be provided by some other publisher in the federation.
  - `type` - Data type, such as integer, double, complex.
  - `units` - Same as with `publications`.
  - `global` - Applies to the `key`, same as with `publications`.
  - `default` - set the default value to return if no publications have been received
  - `only_update_on_change` and `tolerance` - Subscriptions will only consider a new value received when that value has changed more than the delta specified by `tolerance`.

- **`inputs`** - These are lists of the values being sent to and from the given federate.
  - `name` - the name of the input.
  - `required` - The input must have a valid target
  - `type` - Data type, such as integer, double, complex.
  - `units` - Same as with `publications`.
  - `global` - Applies to the `key`, same as with `publications`.
  - `default` - set the default value to return if no publications have been received
  - `target` - A key for a publication the input should receive, may be an array such as ["pub1","pub2","pub3"]
  - `tags` - name and value pairs defining user tags for the interface
  - `only_update_on_change` and `tolerance` - Inputs will only consider a new value received when that value has changed more than the delta specified by `tolerance`.

API Configuration

Configuring the federate interface with the API is done internal to a user-written simulator. The specific API used will depend on the language the simulator is written in. Native APIs for HELICS are available in *C++* and *C*. MATLAB, Java, Julia, Nim, and Python all support the C API calls (ex: `helicsFederateEnterExecutionMode()`). Python and Julia also have native APIs (see: Python (PyHELICS), Julia) that wrap the C APIs to better support the conventions of their languages. The *API References* page contains links to the APIs.

The *Examples* in this User Guide are written in Python – the following federate interface configuration guidance will use the PyHELICS API, but can easily be adapted to other C-based HELICS APIs.

Sample PyHELICS API configuration

The following example of a federate interface configuration with the PyHELICS API comes from the *Fundamental Integration Example*. This co-simulation has exactly the same interface configuration as the Combination Federation above. The only difference is that the federate interfaces are configured with the PyHELICS API.

In the `Charger.py` simulator, the following function calls the APIs to create a federate:

```
def create_combo_federate(fedinitstring, name, period):
    fedinfo = h.helicsCreateFederateInfo()
    # "coreType": "zmq",
    h.helicsFederateInfoSetCoreTypeFromString(fedinfo, "zmq")
    h.helicsFederateInfoSetCoreInitString(fedinfo, fedinitstring)
    # "loglevel": 11,
    h.helicsFederateInfoSetIntegerProperty(fedinfo, h.helics_property_int_log_level, 11)
    # "period": 60,
    h.helicsFederateInfoSetTimeProperty(fedinfo, h.helics_property_time_period, period)
    # "uninterruptible": false,
    h.helicsFederateInfoSetFlagOption(fedinfo, h.helics_flag_uninterruptible, False)
    # "terminate_on_error": true,
    h.helicsFederateInfoSetFlagOption(fedinfo, h.HELICS_FLAG_TERMINATE_ON_ERROR, True)
    # "name": "Charger",
    fed = h.helicsCreateCombinationFederate(name, fedinfo)
    return fed
```

The interface configurations are finalized and registered in one step using the following APIs:

```
fedinitstring = " --federates=1"
name = "Charger"
period = 60
fed = create_combo_federate(fedinitstring, name, period)

num_EVs = 5
end_count = num_EVs
endid = {}
for i in range(0, end_count):
    end_name = f"Charger/EV{i+1}.so"
    endid[i] = h.helicsFederateRegisterGlobalEndpoint(fed, end_name, "double")
    dest_name = f"Controller/ep"
    h.helicsEndpointSetDefaultDestination(endid[i], dest_name)

pub_count = num_EVs
pubid = {}
for i in range(0, pub_count):
    pub_name = f"Charger/EV{i+1}_voltage"
    pubid[i] = h.helicsFederateRegisterGlobalTypePublication(
        fed, pub_name, "double", "V"
    )

sub_count = num_EVs
subid = {}
```

(continues on next page)

(continued from previous page)

```
for i in range(0, sub_count):
    sub_name = f"Battery/EV{i+1}_current"
    subid[i] = h.helicsFederateRegisterSubscription(fed, sub_name, "A")
```

PyHELICS API configuration explanation

All the API calls reference the PyHELICS library with

```
import helics as h
```

Federate Creation `create_combo_federate()`

- **`h.helicsCreateFederateInfo()`** - Sets the federate information variable (set to `fedinfo`)
- **`h.helicsFederateInfoSetCoreTypeFromString(fedinfo, "zmq")`** - Sets the core type for `fedinfo` to `zmq`
- **`h.helicsFederateInfoSetCoreInitString(fedinfo, fedinitstring)`** - Sets the number of federates (`fedinitstring` has been passed as `" --federates=1"`)
- **`h.helicsFederateInfoSetIntegerProperty()`** - Sets log level calling another API, `h.helics_property_int_log_level`
- **`h.helicsFederateInfoSetTimeProperty()`** - Sets time information. This API must receive another API to distinguish which type of time property to set. The period is set with `h.helics_property_time_period`, and `period` has been pass to this function
- **`h.helicsFederateInfoSetFlagOption()`** - API to set a flag for the federate. The flag we are setting is `h.helics_flag_uninterruptible` to `False`, to mirror the JSON configuration
- **`h.helicsFederateInfoSetFlagOption()`** - API to set a flag for the federate. The flag we are setting is `h.HELICS_FLAG_TERMINATE_ON_ERROR` to `True`
- **`fed = h.helicsCreateCombinationFederate(name, fedinfo)`** - Creates the combination federate with the name passed to this function (`Charger`) and the information set above for `fedinfo`

Federate Interface Configuration and Registration

- **Endpoints**
  - **`h.helicsFederateRegisterGlobalEndpoint(fed, end_name, 'double')`** - The `fed` has been created, `end_name` is set in a loop, and the endpoint is registered as global `double`. This API registers the id object for each endpoint, `endid[i]`
  - **`h.helicsEndpointSetDefaultDestination(endid[i], dest_name)`** - As with the JSON configuration, a default destination is set with a destination name, `'Controller/ep'`, for each endpoint object
- **Publications**
  - **`h.helicsFederateRegisterGlobalTypePublication(fed, pub_name, 'double', 'V')`** - The publication interfaces are registered for the `fed` by looping through `pub_name`. The interface is given a datatype of `double`, units of `V` for volts, and designated as global type
- **Subscriptions**
  - **`h.helicsFederateRegisterSubscription(fed, sub_name, 'A')`** - The subscription interfaces are registered for the `fed` by looping through `sub_name`. The interface is given units of `A` for amps. Alternatively, the PyHELICS API for Inputs can be used: `h.helicsFederateRegisterGlobalTypeInput(fed, sub_name, 'double','A')`

Interface configuration, including federate creation and registration, is done prior to the co-simulation execution. The next section in this User Guide places federate interface configuration in the context of the co-simulation stages and discusses the four stages of the co-simulation.

Timing Configuration

Exercises in Co-simulation timing

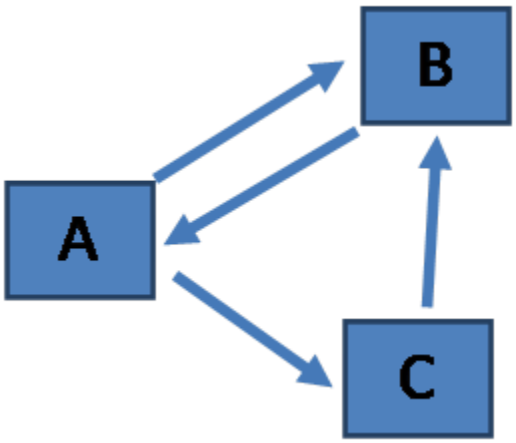
A few simple exercises about co-simulation timing

Key Parameters

- **Period**: The minimum time resolution a federate will allow.
- **Offset**: a shift in the period. Allowed times for federate grants after time 0 are `offset+N*period`, where `N` is a non-negative integer.
- **Time_delta**: the minimum time between grants, i.e. if a federate is granted time T, the next possible time is `T+Time_delta`.

Modifier Flags

- **Uninteruptible**: a federate can only be granted requested times
- **wait_for_current_time_update**: specify that a federate should wait until all federates executing at the current time have finished.



1. Federation Setup [A: period=1; B: period=2; C: period=3]

   a. Following time 0, which federate could execute next? _____

   b. If all federates execute at all allowed times, what is the next time Federate B could have access to data from Federate C. _____

   c. What is the next time all federate will be able to execute simultaneously? _____

   Key Principle: *Federates are interrupted if there is updated data available and allowed time prior to the requested time*

2. Federation Setup [A: period=1,`wait_for_current_time_update`; B: period=2; C: period=3]

   a. At what time will the data from Federate B published at time 2, be available to Federate A? _____

   b. Federate A requests time 4: Federate B publishes at time 2. What time is Federate A granted? _____

c. Federate A requests time 2: Federate B publishes at time 2. What time does Federate A receive the data? _____

e. If A did not have the `wait_for_current_time_update` flag active, what time would Federate A receive the data? _____

Key Principle: *Federates are granted the next allowed time after the time specified in a request if they are not interrupted.*

3. Federation Setup [A: period=1; B: period=2,offset=1,time_delta=2; C: period=3]

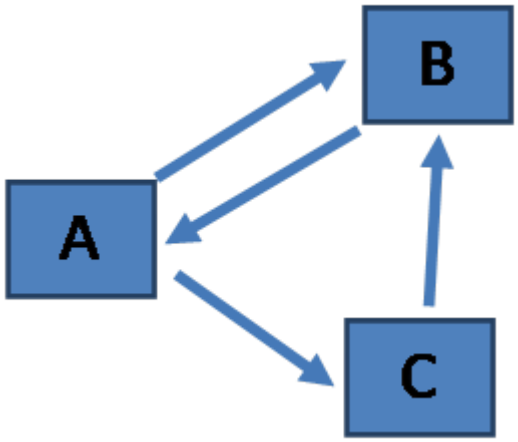
   a. After time=0 what is the next allowable time for Federate B? _____

   b. Federate C requested a time of 4, what time is Federate C granted? _____

4. Federation Setup [A: period=1; B: period=2,uninterruptible; C: period=3]

   a. Federate C Publishes at time 3, Federate B requests time 6, what time will it be granted?_____

   b. If Federate B were not uninterruptible what time would it be granted? _____



5. Federation Setup [A: period=1; B: period=2; C: period=3], Federates will send an update when they have received an update from all other connected federates. Federate A sends an update at time 0, what is the update sequence

Time	Federate(s)

For answers see *answers*

Exercises in Co-simulation timing

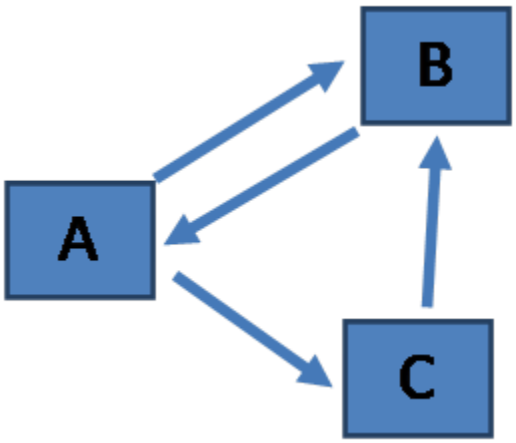
A few simple exercises about co-simulation timing

Key Parameters

- **Period**: The minimum time resolution a federate will allow.
- **Offset**: a shift in the period. Allowed times for federate grants after time 0 are `offset+N*period`, where `N` is a non-negative integer.
- **Time_delta**: the minimum time between grants, i.e. if a federate is granted time T, the next possible time is `T+Time_delta`.

Modifier Flags

- **Uninteruptible**: a federate can only be granted requested times
- **wait_for_current_time_update**: specify that a federate should wait until all federates executing at the current time have finished.



1. Federation Setup [A: period=1; B: period=2; C: period=3]

   a. Following time 0, which federate could execute next? ***A at time 1***

   b. If all federates execute at all allowed times, what is the next time Federate B could have access to data from Federate C. ***4***

   c. What is the next time all federate will be able to execute simultaneously? ***6***

   Key Principle: *Federates are interrupted if there is updated data available and allowed time prior to the requested time*

2. Federation Setup [A: period=1,`wait_for_current_time_update`; B: period=2; C: period=3]

   a. At what time will the data from Federate B published at time 2, be available to Federate A? ***2***

   b. Federate A requests time 4: Federate B publishes at time 2. What time is Federate A granted? ***2***

   c. Federate A requests time 2: Federate B publishes at time 2. What time does Federate A receive the data? ***2***

   e. If A did not have the `wait_for_current_time_update` flag active, what time would Federate A receive the data? ***3***

   Key Principle: *Federates are granted the next allowed time after the time specified in a request if they are not interrupted.*

3. Federation Setup [A: period=1; B: period=2,offset=1,time_delta=2; C: period=3]

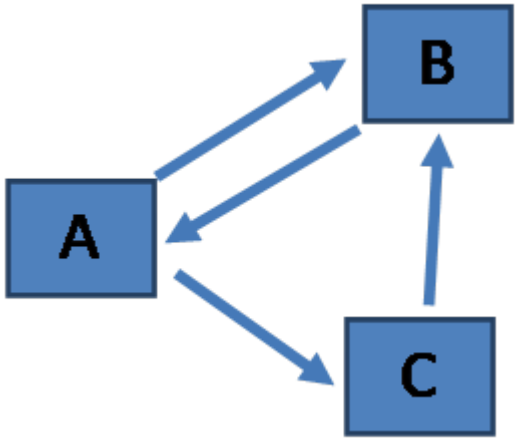
   a. After time=0 what is the next allowable time for Federate B? ***3***, *time=1 is not allowed due to time_delta*

   b. Federate C requested a time of 4, what time is Federate C granted? ***6***

4. Federation Setup [A: period=1; B: period=2,uninterruptible; C: period=3]

   a. Federate C Publishes at time 3, Federate B requests time 6, what time will it be granted?***6***, *B cannot be granted anything other than 6 due to uninterruptible flag*

   b. If Federate B were not uninterruptible what time would it be granted? ***4***



5. Federation Setup [A: period=1; B: period=2; C: period=3], Federates will send an update when they have received an update from all other connected federates. Federate A sends an update at time 0, what is the update sequence

Time	Federate(s)
3	C
4	B
4	A
6	C
6	B
6	A
9	C

The two fundamental roles of a co-simulation platform are to provide a means of data exchange between members of the co-simulation (federates) and a means of keeping the federation synchronized in simulated time.

In HELICS, time synchronization across the federates is managed by each federate requesting a time (via a HELICS API call). When granted a time, the federate generally does the following:

1. Checks all its inputs and grab any new signals (values or messages) that have been sent to it,
2. Execute its native simulation code and update its state to the current simulated time (*e.g.* solving equations governing the behavior of a physical model, calculating a control action, processing and logging data, etc),
3. Publish new values or send new messages to other federates, and
4. Request the next simulated time to update again.

Sometimes this timing configuration is determined by the construction of the simulator (for example, if it has a fixed simulation time step size) and sometimes the simulator will have nothing to do until it receives a new input (for example, with a controller).

In most federates there will be a line of code that look like this (at least if they are *using the Python API*):

```
t = h.helicsFederateRequestTime(fed, time_requested)
```

It is the role of each federate to determine which time it should request and it is the job of those integrating the simulator with HELICS to determine how best to estimate that value. For some simulators, `time_requested` will be the current simulated time (`t`) plus a time step. For other simulators, `time_requested` may be the final time (`HELICS_TIME_MAXTIME`) (see the example on *Combination Federates* for more details on this), and it will only be granted time (interrupted, configured as `"uninterruptible": false`) when there are relevant updates provided by other federates in the co-simulation. Generally, time requests are blocking calls and our federate will do nothing until the HELICS core has granted a time to it.

When a federate makes a time request it calls a HELICS function that blocks the execution of that thread in HELICS. (If the simulator in question is multi-threaded then other threads can continue to operate; hopefully whatever they're working on is largely independent of the rest of the federation.) The federate sits and waits for a return value from that function (the granted time), allowing the rest of the federation to execute. The implication of making a time request is that, given the current state of its boundary conditions, the federate has no tasks to execute until the time it is requesting, or until it receives from another federate a new value that changes its boundary conditions.

After making a time request, federates are granted a time by their HELICS core and the time they are granted will be one of two values: the time they requested (or the next available valid time as defined by their configuration) or an earlier valid time. Being granted a time earlier than requested is always accompanied by a new value or message in one of its inputs, subscriptions, or endpoints. A change in the federate's boundary conditions may require a change in one of the outputs for that federate and its core is obliged to wake up the federate so it can process this new information.

Based on the time requests and grants from all the connected federates, a core will determine the next time it can grant to a federate to guarantee none of the federates will be asked to simulate a point in time that occurs in the past. Every federate will receive a time that is the same as or larger than the last time it was granted. HELICS does support a configuration and some other situations that allows a federate to break this rule, but this is a very special situation and would require all the federates to support this jumping back in time, or accept non-causality and some randomness in execution.

The *section on federates* addressed the data-exchange responsibility of the co-simulation platform and this will address the timing and synchronization requirements. These two functions work hand-in-hand; for data-exchange between federates to be productive, data must be delivered to each federate at the appropriate time.

HELICS co-simulations end under one of two conditions: when all federates have been granted the time of `HELICS_TIME_MAXTIME` or when all federates have notified the broker (via their core) that they are terminating. The termination of the federates triggers a cascade of terminations throughout the federation: once all the federates associated with a core have terminated, the core itself terminates and once all cores associated with a broker have terminated, the broker itself terminates. This concludes the co-simulation and leaves the original models, configuration files, executing simulators, and results files in place for review.

Timing Configuration Options

Managing the timing of federate co-simulation is one of the most important and often challenging aspects of co-simulation. It is not uncommon for a federation to require that certain federates run at particular times or after certain other federates. HELICS provides a wide variety of timing parameters that can be configured for each federate (see the "Timing" section of the *Configuration Options Reference*).

The same JSON configuration file used to set the publications, subscriptions, and endpoints (as discussed in the *section on federates*) also controls how the federate manages its timing within the co-simulation.

Below is an example of how the most common of these are implemented in a federate configuration JSON file:

```
{
  "name": "generic_federate",
```

(continues on next page)

(continued from previous page)

```
  "period": 1.0,
  "offset": 1.0,
  "time_delta": 10.0,
  "uninterruptible": false,
  "wait_for_current_time_update": true
}
```

`period`, `offset`, and `time_delta` are all related and defined with units of seconds.

- `period`: Defines the resolution of the federate and is often tied to the underlying simulation tool. Period forces time grants to specific intervals.
- `offset`: Requires all time grants to be offset in time from the intervals defined by `period` by the amount indicated.
- `time_delta`: Forces the granted time to a minimum interval from the last granted time.

The granted time will be of value `n*period + offset` and it must be later than the last grant by time `time_delta`.

The other two options are common flags which may be invoked:

- `uninterruptible`: Forces the granted time to be the requested time. Generally HELICS will grant a federate a time when it receives new values on any of its inputs under the assumption that the federate is inherently interested in responding to new information to which it has subscribed. If that is not the case, setting this flag will reduce nuisance grants and move the federate forward in a predictable manner.
- `wait_for_current_time_update`: Force the federate with this flag set to be the last one granted a given time and thereby ensures that all other federates have produced outputs for that time. By being last, the federate in question will have updated outputs from all other federates and have the most comprehensive understanding of the system state at that simulated time.

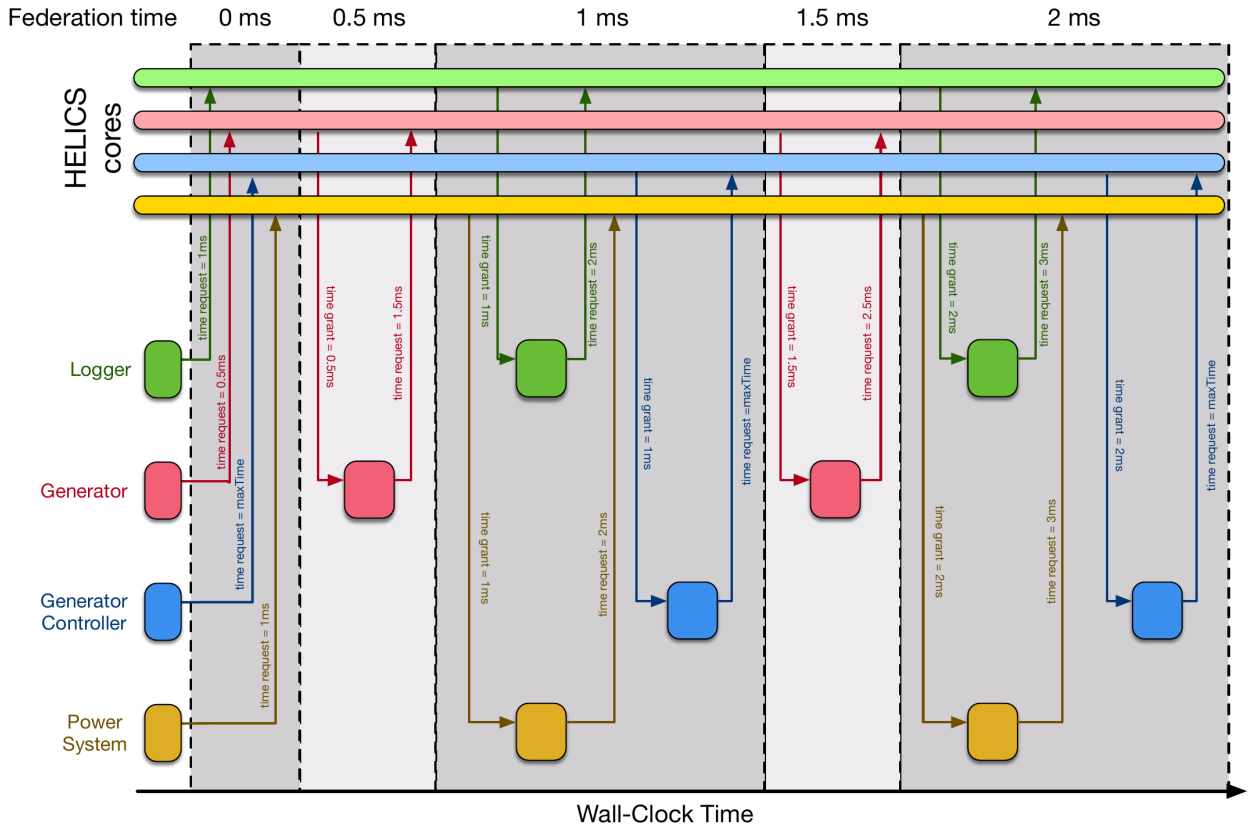
Example: Timing in a Small Federation

For the purposes of illustration, let's suppose that a co-simulation federation with the following timing parameters has been assembled:

- **Logger** - This federate is a results logger and simply writes out to files the current values of various publications made by the other federates in the co-simulation. This logging simulator will record values every 1 ms and as such, the JSON config sets `period` to this value and sets the `uninterruptible` flag.
- **Generator** - This is a generator simulator that specializes in comprehensive modeling of the machine dynamics. The Generator will have an endpoint used to receive commands from the Generator Controller and subscriptions to outputs from the Power System that provide inputs necessary to calculate its internal dynamics.

  The models of the generator are valid at a time-step of 0.1 ms and thus the simulator integrator requires that the `period` of the HELICS interface be set to some multiple of 0.1. In this case we'll use 1 ms and to ease integration with the Power System federate, it will also have an `offset` of 0.5 ms.
- **Generator Controller** - This is an event-based simulator, updating the control commands to the Generator federate whenever new inputs are received from the Power System federate (subscriptions to the physical values it calculates). As such, it will always request `HELICS_TIME_MAXTIME`, expecting to be granted times whenever the state of the Power System federate changes. The `timeDelta` will be set to 0.010 ms to replicate the time it takes to calculate and communicate the command signals to the Generator.
- **Power System** - This federate is a classic power system dynamics simulator with a fixed time-step of 1 ms. The integrator of this simulator choose to realize this by setting the `uninterruptible` flag and hard-coding the time requests to advance at 1 ms intervals.

Below is a timing diagram showing how these federates interact during a co-simulation. The filled blocks show when each federate has been woken up and is active.



Items of notes:

- Generator Controller gets granted a time of 1 ms (at the first grant time) even though is requested `HELICS_TIME_MAXTIME` because a message was created by the Power System federate at that time stamp. As Generator Controller depends on nothing else, HELICS was able to grant it the same time as Power System even though it is clearly performing its calculations after Power System has performed its.
- Relatedly, Generator Controller requests a time of `HELICS_TIME_MAXTIME` once it has calculated the new control signals for Generator. Due to the value set by `timeDelta`, the soonest time it can be granted would be 0.01 ms after its most recent granted time (1.01 in the case of the first operational period, 2.01 in the case of the second period.)
- When Logger is granted a time of 1 ms, the values it will record are those previously published by other federates. Specifically, the new values that Power System is calculating are not available for Logger to record.

Co-simulation Stages

HELICS has several stages to the co-simulation. Creation, initialization, execution, and final state. A call to `helicsFederateEnterExecutingMode()` is the transition between initialization and execution.

- *Creation*
  - *Registration*
    - *Using a JSON Config File*
    - *Using PyHELICS API Calls*
  - *Collecting the Interface Objects*

- *Initialization*
- *Execution*
  - *Get Inputs*
  - *Internal Updates and Calculations*
  - *Publish Outputs*
- *Final State*

Creation

For the purposes of these examples, we will assume the use of a Python binding. If, as the simulator integrator, you have needs beyond what is discussed here you'll have to dig into the *developer documentation on the APIs* to get the details you need.

To begin, at the top of your Python module (*after installing the Python HELICS module*), you'll have to import the HELICS library, which will look something like this:

```
import helics as h
```

Registration

As discussed in the previous section on *Federate Interface Configuration*, configuration of federates can be done with either JSON config files or with the simulator's API.

Using a JSON Config File

In HELICS there is a single API call that can be used to read in all of the necessary information for creating a federate from a JSON configuration file. The JSON configuration file, as discussed earlier in this guide, contains both the federate info as well as the metadata required to define the federate's publications, subscriptions and endpoints. The API calls for creating each type of federate are given below.

For a value federate:

```
fed = h.helicsCreateValueFederateFromConfig("fed_config.json")
```

For a message federate:

```
fed = h.helicsCreateMessageFederateFromConfig("fed_config.json")
```

For a combination federate:

```
fed = h.helicsCreateCombinationFederateFromConfig("fed_config.json")
```

In all instances, this function returns the federate object `fed` and requires a path to the JSON configuration file as an input.

Using PyHELICS API Calls

Additionally, there are ways to create and configure the federate directly through HELICS API calls, which may be appropriate in some instances. First, you need to create the federate info object, which will later be used to create the federate:

```
fedinfo = h.helicsCreateFederateInfo()
```

Once the federate info object exists, HELICS API calls can be used to set the *configuration parameters* as appropriate. For example, to set the `only_transmit_on_change` flag to true, you would use the following API call:

```
h.helicsFederateInfoSetFlagOption(fed, 6, True)
```

(The "6" there is the integer value for appropriate HELICS enumeration. The definition of the enumerations can be found in the C++ API reference and also cross shown in the *Configurations Options Reference*.)

Once the federate info object has been created and the appropriate options have been set, the helics federate can be created by passing in a unique federate name and the federate info object into the appropriate HELICS API call. For creating a value federate, that would look like this:

```
fed = h.helicsCreateValueFederate(federate_name, fedinfo)
```

Once the federate is created, you now need to define all of its publications, subscriptions and endpoints. The first step is to create them by registering them with the federate with an API call that looks like this:

```
pub = h.helicsFederateRegisterPublication(fed, key, data_type)
```

This call takes in the federate object, a string containing the publication key (which will be prepended with the federate name), and the data type of the publication. It returns the publication object. Once the publication, subscription and endpoints are registered, additional API calls can be used to set the info field in these objects and to set certain options. For example, to set the only transmit on change option for a specific publication, this API call would be used:

```
pub = h.helicsPublicationSetOption(pub, 454, True)
```

Once the federate is created, you also have the option to set the federate information at that point, which - while functionally identical to setting the federate info in either the federate config file or in the federate info object - provides integrators with additional flexibility, which can be useful particularly if some settings need to be changed dynamically during the co-simulation. The API calls are syntactically very similar to the API calls for setting up the federate info object, except instead they target the federate itself. For example, to revisit the above example where the `only_transmit_on_change` flag is set to true in the federate info object, if operating on an existing federate, that call would be:

```
h.helicsFederateSetFlagOption(fed, 6, True)
```

Collecting the Interface Objects

Having configured the publications, subscriptions and endpoints and registered this information with HELICS, the channels for sending and receiving this information have been created within the co-simulation framework. If you registered the publication, subscriptions and endpoints within your code (i.e., using HELICS API calls), you already have access to each respective object because it was returned when you made the registration call. However, if you created your federate using a configuration file which contained all of this information, you now need to retrieve these objects from HELICS so that you can invoke them during the execution of your co-simulation. The following calls will allow you to query HELICS for the metadata associated with each publication. Similar calls can be used to get input (or subscription) and endpoint information.

```
pub_count = h.helicsFederateGetPublicationCount(fed)
pub = h.helicsFederateGetPublicationByIndex(fed, index)
pub_key = h.helicsPublicationGetKey(pub)
```

The object returned when the helicsFederateGetPublicationByIndex() method is invoked is the interface object used for retrieving other publication metadata (as in the helicsPublicationGetKey() method) and when publishing data to HELICS (as described in the execution section below).

Initialization

Initialization mode exists to help a federation reach a consistent state or otherwise generally prepare to begin the advancement through time. Each federate can call `helicsFederateEnterInitializingMode()` and perform whatever internal set-up it needs to do as well as publish outputs that will be available to the rest of the federation at simulation time t=0 when entering execution mode (see the next section).

If the federation needs to iterate in initialization mode prior to entering execution mode each federate calls `helicsFederateEnterExecutingModeIterative()`. This API has two special aspects:

1. Calling the API requires that the federate declare its needs for iteration using an enumeration:

   `NO_ITERATION` – don't iterate

   `FORCE_ITERATION` – guaranteed iteration, stay in iteration mode

   `ITERATE_IF_NEEDED` – If there is data available from other federates Helics will iterate, if no additional data is available it will move to execution mode and have granted time=0.

2. The API returns an enumeration indicating the federation's iteration state:

   `NEXT_STEP` - Iteration has completed and the federation should move to the next time step. In the case of exiting initialization, this will be the time between t=0 (which was just completed by the iteration process) and the next time grant.

   `ITERATING` - Federation has not ceased iterating and will iterate once again. During this time the federate will need to check all its inputs and subscriptions, recalculate its model, and produce new outputs for the rest of the federation.

To implement this initialization iteration, all federates need to implement a loop where `helicsFederateEnterExecutingModeIterative()` is repeatedly called and the output of the call is evaluated. The call to the API needs to use the federate's internal evaluation of the stability of the solution to determine whether it needs to request another iteration. The returned value of the API will determine whether the federate needs to re-solve its model with new inputs from the rest of the federation or enter normal execution.

Execution

Once the federate has been created, all subscriptions, publications and endpoints have been registered and the federation initial state has been appropriately set, it is time to enter execution mode. This can be done with the following API call:

```
h.helicsFederateEnterExecutingMode(fed)
```

This method call is a blocking call; your custom federate will sit there and do nothing until all other federates have also finished any set-up work and have also requested to begin execution of the co-simulation. Once this method returns, the federation is effectively at simulation time of zero.

At this point, each federate will now step through time, exchanging values with other federates in the co-simulation as appropriate. This will be implemented in a loop where each federate will go through a set of prescribed steps at each time step. At the beginning of the co-simulation, time is at the zeroth time step (t = 0). Let's assume that the co-simulation will end at a pre-determined time, t = max_time. The nature of the simulator will dictate how the time loop is handled. However, it is likely that the co-simulation loop will start with something like this:

```
t = 0
while t < end_time:
    pass  # co-simulation code would go here
```

Now, the federate begins to step through time. For the purposes of this example, we will assume that during every time step, the federate will first take inputs in from the rest of the co-simulation, then make internal updates and calculations and finish the time step by publishing values back to the rest of the co-simulation before requesting the next time step.

Get Inputs

The federate will first listen on each of its inputs (or subscriptions) and endpoints to see whether new information has been sent from the rest of the federation. The first code sample below shows how information can be retrieved from an input (or subscriptions) through HELICS API calls by passing in the subscription object. As can be seen, HELICS has built in type conversion (where possible) and regardless of how the sender of the data has formatted it, HELICS can present it as requested by the appropriate method call.

```
int_value = h.helicsInputGetInteger(sub)
float_value = h.helicsInputGetDouble(sub)
real_value, imag_value = h.helicsInputGetComplex(sub)
string_value = h.helicsInputGetChar(sub)
...
```

It may also be worth noting that it is possible on receipt to check whether an input has been updated before retrieving values. That can be done using the following call:

```
updated = h.helicsInputIsUpdated(sid)
```

Which returns true if the value has been updated and false if it has not.

Receiving messages at an endpoint works a little bit differently than when receiving values through a subscription. Most fundamentally, there may be multiple messages queued at an endpoint while there will only ever be one value received at a subscription (the last value if more than one is sent prior to being retrieved). To receive all of the messages at an endpoint, they need to be popped off the queue. An example of how this might be done is given below.

```
while h.helicsEndpointPendingMessages(end) > 0:
    msg_obj = h.helicsEndpointGetMessageObject(end)
```

To get the source of each of the messages received at an endpoint, the following call can be used:

```
msg_source = h.helicsMessageGetOriginalSource(msg_obj)
```

Internal Updates and Calculations

At this point, your federate has received all of its input information from the other federates in the co-simulation and is now ready to run whatever updates or calculations it needs to for the current time step.

Publish Outputs

Once the new inputs have been collected and all necessary calculations made, the federate can publish whatever information it needs to for the rest of the federation to use. The code sample below shows how these output values can be published out to the federation using HELICS API calls. As in when reading in new values, these output values can be published as a variety of data types and HELICS can handle type conversion if one of the receivers of the value asks for it in a type different than published.

```
h.helicsPublicationPublishInteger(pub, int_value)
h.helicsPublicationPublishDouble(pub, float_value)
h.helicsPublicationPublishComplex(pub, real_value, imag_value)
h.helicsPublicationPublishChar(pub, string_value)
...
```

For sending a message through an endpoint, that once again looks a little bit different, in this case because - unlike with a publication - a message requires a destination. If a default destination was set when the endpoint was registered (either through the config file or through calling `h.helicsEndpointSetDefaultDestination()`), then an empty string can

be passed. Otherwise, the destination must be provided as shown in API call below where `dest` is the destination and `msg` is the message to be sent.

```
h.helicsEndpointSendMessageRaw(end, dest, msg)
```

Final State

Once the federate has completed its contribution to the co-simulation, it needs to close out its connection to the federation. Typically a federate knows it has reached the end of the co-simulation when it is granted `maxTime`. To leave the federation cleanly (without causing errors for itself or others in the co-simulation) the following process needs to be followed:

```
h.helicsFederateFinalize(fed)
h.helicsFederateFree(fed)
h.helicsCloseLibrary()
```

`helicsFederateFinalize()` signals to the core and brokers that this federate is leaving the co-simulation. This process will take an indeterminate amount of time and thus it is necessary to poll the connection status to the broker. Once that connection has closed, the memory of the federate (associated with HELICS) is freed up with `helicsFederateFree()` and the processes in the HELICS library are terminated with `helicsCloseLibrary()`. At this point, the federate can safely end execution completely.

Logging

Logging in HELICS provides a way to understand the operation of a federate and is normally handled through an independent thread. The thread prints message to the console and or to a file as events within a federate occur. This section discusses how to use the log files to confirm the co-simulation executed properly and to debug when it doesn't.

- *Log Levels*
- *Setting up the Simulator for Logging*
- *Setting up the Federate for Logging*
- *Setting up the Core/Broker for Logging*

Log Levels

There are several levels used inside HELICS for logging. The level can be set with the enumerations when using an API to set the logging level. When configuring the log level via an external JSON config, the enumerations are slightly different:

API enumeration	JSON config keyword
HELICS_LOG_LEVEL_NO_PRINT	no_print
HELICS_LOG_LEVEL_ERROR	error
HELICS_LOG_LEVEL_PROFILING	profiling
HELICS_LOG_LEVEL_WARNING	warning
HELICS_LOG_LEVEL_SUMMARY	summary
HELICS_LOG_LEVEL_CONNECTIONS	connections
HELICS_LOG_LEVEL_INTERFACES	interfaces
HELICS_LOG_LEVEL_TIMING	timing
HELICS_LOG_LEVEL_DEBUG	debug
HELICS_LOG_LEVEL_DATA	data
HELICS_LOG_LEVEL_TRACE	trace

- `HELICS_LOG_LEVEL_NO_PRINT` Don't log anything
- `HELICS_LOG_LEVEL_ERROR` Log error and faults from within HELICS
- `HELICS_LOG_LEVEL_PROFILING` Log profiling messages
- `HELICS_LOG_LEVEL_WARNING` Log warning messages of things that might be incorrect or unusual
- `HELICS_LOG_LEVEL_SUMMARY` Log summary messages on startup and shutdown. The Broker will also generate a summary with the number of federates connected and a few other items of information
- `HELICS_LOG_LEVEL_CONNECTIONS` Log a message for each connection event (federate connection/disconnection)
- `HELICS_LOG_LEVEL_INTERFACES` Log messages when interfaces, such as endpoints, publications, and filters are created
- `HELICS_LOG_LEVEL_DEBUG` Log messages related to debugging, similar to timing
- `HELICS_LOG_LEVEL_TIMING` Log messages related to timing information such as mode transition and time advancement
- `HELICS_LOG_LEVEL_DATA` Log messages related to data passage and information being sent or received
- `HELICS_LOG_LEVEL_TRACE` Log all internal messages being sent

NOTE: the numerical values of these levels is subject to change

`timing`, `data` and `trace` log levels can generate a large number of messages and should primarily be used for debugging. `trace` will produce a very large number of messages most of which deal with internal communications and is primarily for debugging timing in HELICS.

Log lines will often look like

```
echo1 (131072) (0)[t=4.0]::Time mismatch detected granted time >requested time 5.5 vs 5.0
```

or

```
commMessage||26516-enRPa-PzaBB-ZG190-lj14t:got new broker information
```

which includes a name and internal id code for the federate followed by a time in parenthesis and the message. If it is a warning or error, there will be an indicator before the object name. Names for brokers or cores are often auto generated and look like `26516-enRPa-PzaBB-ZG190-lj14t` which is essentially a random string with a thread id in the front. In this case, the `commMessage` indicates it came from one of the communication modules.

Setting up the Simulator for Logging

The *Fundamental Base Example* incorporates simple logging for the two federate co-simulation. These federates, Battery and Charger, are user-written in Python with PyHELICS, so we have the luxury of setting up the simulator for logging.

In the Battery simulator, we need to import `logging` and set up the logger:

```
import logging


logger = logging.getLogger(__name__)
logger.addHandler(logging.StreamHandler())
logger.setLevel(logging.DEBUG)
```

Now we can use the `logger` to print different levels of detail about the co-simulation execution to log files. These files will be generated for each federate in the co-simulation and the broker with the naming convention "name assigned to federate/broker"`.log`.

A set of functions are available for individual federates to generate log messages. These functions must be placed in the simulator. In the *Fundamental Base Example*, the `logger.info()` and `logger.debug()` methods are used. Stipulating different types of log messages allows the user to change the output of the log files in one location – the config file for the federate. These will log a message at the `log_level` specified in the config file.

```
logger.info("Only prints to log file if log_level = 2 or summary")
logger.debug("Only prints to log file if log_level = 6 or data")
logger.error("Only prints to log file if log_level = 0 or error")
logger.warning("Only prints to log file if log_level = 1 or warning")
```

Setting up the Federate for Logging

Most of the time the log for a federate is the same as for its core. This is managed through a few properties in the HelicsFederateInfo class which can also be directly specified through the property functions.

- HELICS_PROPERTY_INT_LOG_LEVEL - General logging level applicable to both file and console logs
- HELICS_PROPERTY_INT_FILE_LOG_LEVEL Level to log to the file
- HELICS_PROPERTY_INT_CONSOLE_LOG_LEVEL Level to log to the console

These properties can be set using the JSON configuration for each federate:

```
{
  "name": "Battery",
  "log_level": 1
}
```

Or with the API interface functions for each federate:

```
h.helicsFederateInfoSetIntegerProperty(fed, h.HELICS_PROPERTY_INT_LOG_LEVEL, 1)
```

Setting up the Core or Broker for Logging

It is possible to specify a log file to use on a core. This can be specified through the coreinit string `--logfile logfile.txt`

or on a core object

```
h.helicsCoreSetLogFile(core, "logfile.txt")
```

A similar function is available for a broker. The Federate version will set the logFile on the connected core.

With the API:

```
h.helicsFederateSetLogFile(fed, "logfile.txt")
```

Within the HELICS runner JSON:

```
{
  "federates": [
    {
      "directory": ".",
```

(continues on next page)

(continued from previous page)

```
      "exec": "helics_broker -f 2 --loglevel=7",
      "host": "localhost",
      "name": "broker"
    }
  ],
  "name": "fundamental_default"
}
```

A federate also can set a logging callback so log messages can be processed in whatever fashion is desired by a federate.

In C++ the method on a federate is:

```
setLoggingCallback (const std::function<void(int, const std::string &, const std::string␣
↪&)> &logFunction);
```

The logging callback function take 3 parameters about a message and in the case of C callbacks a pointer to user data.

- **loglevel**: an integer describing the level of the message
- **identifier**: a string that may contain information on the source of the log message and state/time information
- **message**: the actual message to log

In PyHELICS:

```
h.helicsFederateSetLoggingCallback(fed, logger, user_data)
```

- **fed**: the helics.HelicsFederate that is created with helics.helicsCreateValueFederate, helics.helicsCreateMessageFederate or helics.helicsCreateCombinationFederate
- **logger**: a callback with signature void(int, const char _, const char _, void *); the function arguments are loglevel, an identifier string, and a message string to log, and a pointer to user data
- **user_data**: a pointer to user data that is passed to the function when executing

Log Buffer

As of Version 3.2, HELICS cores, brokers, and federates have the capability to buffer log messages. This can be activated via the `--logbuffer` flag, or `--logbuffer=X` option. The default size is 10 messages for the `--logbuffer` flag. For cores and federates there is a `HELICS_PROPERTY_INT_LOG_BUFFER` property that can be set. And it can also be activated via the *`logbuffer <X>` command* remotely. <X> is the desired size of the buffer. `logbuffer stop` will deactivate the buffer in a remote command. The logs can be retrieved via the *"logs"* query. The buffer is available even after disconnect for cores and brokers from the local object.

Value Buffer Warning

Federates can warn when queued future value buffers exceed the `value_buffer_warning` threshold. The threshold is specified in MiB, defaults to 100 MiB, and can be disabled by setting it to 0. This warning can indicate that publishers are advancing faster than subscribers and queued value payload data is accumulating.

Remote Logging

As of version 3.2 it is possible to have a HELICS object clone its logging to another object. For example a federate can request the broker log messages be sent to a federate log via a *`remotelog` command*. Multiple objects can receive the same log and a single object can receive multiple remote logs. The federate/broker/core that sends the remote log command is the one who will receive the logs and can include logging level to receive debugging logs from a particular location. This can be stopped as well via the `remotelog stop` command.

Additional Broker Features

Time monitor

As of Version 3.2 Brokers have the capability to specify a federate as a time monitor, this does not affect the co-simulation but instructs the Broker to get time messages from a particular federate. There are two new command line arguments for brokers `--timemonitor=fedName` and `--timemonitorperiod=<time>` The second can only be used if the first is specified as well.

When specified the broker creates a link to that federate and will generate a log message of form

```
TIME: exec granted
TIME: granted time=2
TIME: disconnected, last time=4
```

The monitor can be stopped via the *command* interface through the `monitor` keyword.

```
//changing the federate to use as a monitor or setting one up the first time
broker->sendCommand("broker","monitor newfed");
//changing the federate and period to use as a monitor or setting one up the first time
broker->sendCommand("broker","monitor newfed 4sec");
//stopping the time_monitor
broker->sendCommand("broker","monitor stop");
```

The time generated by the monitor federate is also the time displayed in some log messages generated by the broker and can be queried via the `time_monitor` query.

Running HELICS Co-Simulations

Execution of the HELICS co-simulation is done from the command line using the `helics run ...` command in PyHELICS. Each simulator must be executed individually in order to join the federation. The `helics run ...` command condenses these individual command line execution commands through the use of a JSON called the "runner file"".

All the *examples* are written with a runner file for execution. In the *Fundamental Base Example*, we need to launch three things: the broker, the Battery federate, and the Charger federate.

The file `fundamental_default_runner.json` includes:

```
{
  "federates": [
    {
      "directory": ".",
      "exec": "helics_broker -f 2 --loglevel=7",
      "host": "localhost",
      "name": "broker"
    },
    {
      "directory": ".",
      "exec": "python -u Charger.py 1",
      "host": "localhost",
      "name": "Charger"
    },
    {
      "directory": ".",
      "exec": "python -u Battery.py 1",
```

(continues on next page)

(continued from previous page)

```
      "host": "localhost",
      "name": "Battery"
    }
  ],
  "name": "fundamental_default"
}
```

This tells HELICS to launch three federates named `broker`, `Charger`, and `Battery`. The `directory` tells HELICS the location of the executable. For the `broker`, the executable `helics_broker` should be *configured to suit your needs*. The `broker` is launched with the executable `helics_broker`, to which we pass the information `-f 2` meaning there will be two federates, and `--loglevel=7` meaning that we want *all internal messages* to be sent to the log file.

The other two federates are Python based and just need to be called with `python -u`.

Once the runner file is specified to include information about where the executables live (`directory`), the execution command for each (`exec`), the host, and the name, the entire federation can be launched with the following command:

```
> helics run --path=fundamental_default_runner.json
```

The next section discusses using the Web Interface to interact with a running HELICS co-simulation. The federation must be launched in this ways in order to use the Web Interface.

Simulator Integration

A "simulator" is the executable program. As soon as one particular instance of that simulator begins running in a co-simulation it is considered a "federate". Every federate (instance of a simulator) will require configuration of the way it will communicate (send signals) to other federates in the federation. For simulators that already have HELICS support, the configuration takes the form of a JSON (or TOML) file; bespoke simulators can be configured with the HELICS APIs in the code or via a JSON file. The essential information that HELICS configuration defines is:

Federate name - The unique name this federate will be known as throughout the federation. It is essential this name is unique so that HELICS messages can route properly.

Core type - The core manages interfaces between the federation and the federate; there are several messaging technologies supported by HELICS.

Publications and Inputs - Publication configuration contains a listing of source interface name, data types, and units being sent by the federate; input configuration does the same for values being received by the federate. If supported by the simulator (e.g., *a Python simulator*), these values can be mapped to internal variables of the simulator from the configuration file.

Endpoints - Endpoints are sending and receiving points for HELICS messages to and from message federates. They are declared and defined for each federate.

Time step size - This value defines the resolution of the simulator to prevent HELICS from telling the simulator to step to a time of which it has no concept (e.g. trying to simulate the time of 1.5 seconds when the simulator has a resolution of one second).

Integration of Federates

A co-simulation is, in some sense, a simulation of simulations. There will be two types of configuration required:

1. Individual federates (identifying models to be used, defining the start and stop time of the simulation, defining how the results of the simulation should be stored, etc…) and
2. How each federate will connect to and interact with the other federates in the co-simulation.

One of the goals of a co-simulation platform like HELICS is to make the connecting easier and more efficient by providing a standardized method of configuration.

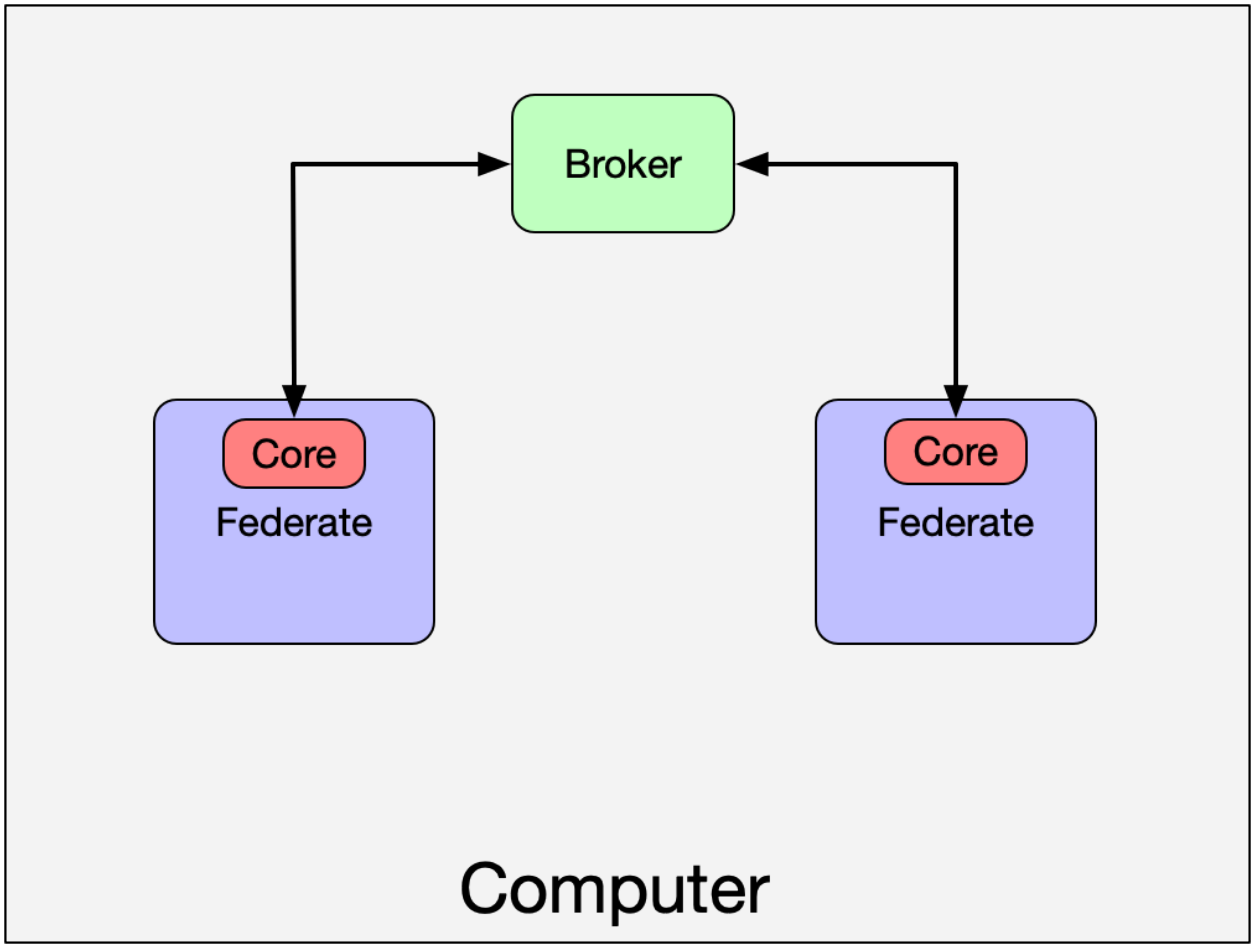
Integration of federates requires definition of the message topology (who is passing what information to whom) and the broker topology (which federates/cores are connected to which brokers). Message topology requires understanding the interactions of the system the simulators are trying to replicate and identifying the boundaries where they could exchange data. Broker topology will be kept simple for the Fundamental Topics and Examples.

This section introduces the simplest broker topology for integrating federates into a federation, and the basics for integrating federates with a JSON and with API calls.

Broker Topology

Broker topology is somewhat optional for simple co-simulations, but offers an increase in performance if it is possible to identify groups of federates that interact often with each other but rarely with the rest of the federation. In such cases, assigning that group of federates their own broker will remove the congestion their messages cause with the federation as a whole. The Fundamental Topics and Examples are built with a single broker.

The figure below shows the most common architecture for HELICS co-simulation. Each core has only one federate as an integrated executable, all executables reside on the same computer and are connected to the same broker. This architecture is particularly common for small federates and/or co-simulations under development. This is also the architecture for the *Fundamental Examples*.



Configuring the federate

Let's look at a generic JSON configuration file as an example with the more common parameters shown. As we'll see *later in this section*, this file is loaded by the federate using a specific API, allowing the same simulator to be used to create many federates that are all unique without having to modify the source code of the simulator. There are many, many more configuration parameters that this file could include; a relatively comprehensive list along with explanations of the functionality they provide can be found in the *federate configuration* guide.

Sample federate JSON configuration file

```
{
    "name":"generic_federate",
    "coreType": "zmq"
    "publications" : [
          {
               "key" : "IEEE_123_feeder_0/totalLoad",
               "global" : true,
               "type" : "complex",
               "unit" : "VA",
          }
     ],
     "subscriptions" : [
          {
               "required": true,
               "key" : "TransmissionSim/transmission_voltage",
               "type" : "complex",
               "unit" : "V",
               "info" : "{
                    \"object\" : \"network_node\",
                    \"property\" : \"positive_sequence_voltage\"
                    }"
          }
     ],
     "endpoints" : [
        {
            "name" : "voltage_sensor",
            "global" : true,
            "destination" : "voltage_controller",
            "info" : ""
        }
     ]
}
```

JSON configuration file explanation

- **name** - Every federate must have a unique name across the entire federation; this is functionally the address of the federate and is used to determine where HELICS messages are sent. An error will be generated if the federate name is not unique.
- **coreType** - There are a number of technologies or message buses that can be used to send HELICS messages among federates. Every HELICS enabled simulator has code in it that creates a core which connects to a HELICS broker using one of these messaging technologies. ZeroMQ (zmq) is the default core type and most commonly used but there are also cores that use TCP and UDP networking protocols directly (forgoing ZMQ's guarantee of delivery and reconnection functions), IPC (uses Boost's interprocess communication for fast in-memory

message-passing but only works if all federates are running on the same physical computer), and MPI (for use on HPC clusters where MPI is installed).

- **`publications` and/or `subscriptions`** - These are lists of the values being sent to and from the given federate.
- **`key`** -
  - `publications` - The string in this field is the unique identifier (at the federate level) for the value that will be published to the federation. If `global` is set (see below) it must be unique to the entire federation.
  - `subscriptions` - This string identifies the federation-unique value that this federate wishes to receive. Unless `global` has been set to `true` in the publishings JSON configuration file, the name of the value is formatted as `<federate name>/<publication key>`. Both of these strings can be found in the publishing federate's JSON configuration file as the `name` and `key` strings, respectively. If `global` is `true` the string is just the `key` value.
- **`global`** - (publications only) `global` is used to indicate that the value in `key` will be used as a global name when other federates are subscribing to the message. This requires that the user ensure that the name is used only once across all federates. Setting `global` to `true` is handy for federations with a small number of federates and a small number of message exchanges as it allows the `key` string to be short and simple. For larger federations, it is likely to be easier to set the flag to `false` and accept the extra naming
- **`required`** -
  - `publications` - At least one federate must subscribe to the publications.
  - `subscriptions` - The message being subscribed to must be provided by some other publisher in the federation.
- **`type`** - HELICS supports data types and data type conversion (as best it can).
- **`units`** - HELICS is able to do some levels of unit conversion, currently only on double type publications but more may be added in the future. The units can be any sort of unit string, a wide assortment is supported and can be compound units such as m/s^2 and the conversion will convert as long as things are convertible. The unit match is also checked for other types and an error if mismatching units are detected. A warning is also generated if the units are not understood and not matching. The unit checking and conversion is only active if both the publication and subscription specify units.
- **`info`** - The `info` field is entirely ignored by HELICS and is used as a mechanism to pass configuration information to the federate so that it can properly integrate into the federation. Thus, there is no standard content or format for this field; it is entirely up to the individual simulators to decide how the data in this field (if any) should be used. Often it is used by simulators to map the HELICS names into internal variable names as shown in the above example. In this case, the object `network_node` has a property called `positive_sequence_voltage` that will be updated with the value from the subscription `TransmissionSim/transmission_voltage`.
- **`global`** - Just as in value federates, `global` allows for the identifier of the endpoint to be declared unique for the entire federation.
- **`destination`** - For endpoints that send all outgoing messages to only a single endpoint, `destination` allows the endpoint to be specified in the JSON configuration. This allows for a more modular implementation of the federate since this parameter is externally defined rather than being hardcoded in the federate itself.
- **`info`** - Just as in the value federate, the string in this field is ignored by HELICS and can be used by the federate for internal configuration purposes.

Typical Federate Execution

For the remainder of this section of the guide, we'll walk through the typical stages of co-simulation, providing examples of how these might be implemented using HELICS API calls. For the purposes of these examples, we will assume the use of a Python binding. If, as the simulator integrator, you have needs beyond what is discussed here you'll have to dig into the *developer documentation on the APIs* to get the details you need.

To begin, at the top of your Python module (*after installing the Python HELICS module*), you'll have to import the HELICS library, which will look something like this:

```
import helics as h
```

Federate Information

Each federate has a core set of configuration information and metadata associated with it, which will either need to be set within your code or will be set based on defaults. When creating a new federate, only one piece of metadata is actually required, and that is the federate name, which must be unique within the federation. However, there are many other configuration options that can be set for the federate, including whether the federate can be interrupted between its native time steps, a minimum time step for its execution and the level to use when the federate logs information. Information on all of these configuration options, including default settings, can be found in the *Configurations Options Reference*.

Publications, Subscriptions and Endpoints

One of the first design choices you have to make is the type of federate that you will create to instantiate your simulator within the co-simulation. At this point, we will revisit the question on what kind of data you expect your simulator to exchange with the rest of the federation. There are three kinds of federates within HELICS: *value federates*, *message federates*, and combination federates.

Value federates are used to exchange physical values through HELICS using a publication/subscription architecture, where only a single value can be received at a given subscription at each time step. Value federates are used to represent physics-based interdependencies. An example of where the exchange of values is probably most appropriate is where the same data point is represented in two different simulators, such as the voltage at a transmission bus that corresponds to the voltage at a distribution feeder head.

By contrast, message federates are used to exchange messages through HELICS that look and behave more like communications-based data. Examples of this might include control signals or measurement data. This is done using endpoints, rather than publications and subscriptions, and unlike in the value case, more than one message can be received at an endpoint at any given time step.

A combination federate is one that handles both values and messages. More details on the differences between these federate types are provided elsewhere in this guide.

Create the HELICS Federate

Now that you've decided what kind of federate you are going to use to instantiate your simulator within the federation, you'll need to actually create that federate in your code. There are two ways to do this: from a configuration file or programmatically, using a sequence of HELICS API calls. In most instances, using a configuration file is probably simpler and more modular. However, we will go through both options below as there may be times when creating the federate in your source code is necessary or more appropriate.

Using a Config File

In HELICS there is a single API call that can be used to read in all of the necessary information for creating a federate from a JSON configuration file. The JSON configuration file, as discussed earlier in this guide, contains both the federate info as well as the metadata required to define the federate's publications, subscriptions and endpoints. The API calls for creating each type of federate are given below.

For a value federate:

```
fed = h.helicsCreateValueFederateFromConfig("fed_config.json")
```

For a message federate:

```
fed = h.helicsCreateMessageFederateFromConfig("fed_config.json")
```

For a combination federate:

```
fed = h.helicsCreateCombinationFederateFromConfig("fed_config.json")
```

In all instances, this function returns the federate object `fed` and requires a path to the JSON configuration file as an input.

Using HELICS API Calls

Additionally, there are ways to create and configure the federate directly through HELICS API calls, which may be appropriate in some instances. First, you need to create the federate info object, which will later be used to create the federate:

```
fi = h.helicsCreateFederateInfo()
```

Once the federate info object exists, HELICS API calls can be used to set the *configuration parameters* as appropriate. For example, to set the only_transmit_on_change flag to true, you would use the following API call:

```
h.helicsFederateInfoSetFlagOption(fi, 6, True)
```

Once the federate info object has been created and the appropriate options have been set, the helics federate can be created by passing in a unique federate name and the federate info object into the appropriate HELICS API call. For creating a value federate, that would look like this:

```
fed = h.helicsCreateValueFederate(federate_name, fi)
```

Once the federate is created, you now need to define all of its publications, subscriptions and endpoints. The first step is to create them by registering them with the federate with an API call that looks like this:

```
pub = h.helicsFederateRegisterPublication(fed, key, data_type)
```

This call takes in the federate object, a string containing the publication key (which will be prepended with the federate name), and the data type of the publication. It returns the publication object. Once the publication, subscription and endpoints are registered, additional API calls can be used to set the info field in these objects and to set certain options. For example, to set the only transmit on change option for a specific publication, this API call would be used:

```
pub = h.helicsPublicationSetOption(pub, 454, True)
```

Once the federate is created, you also have the option to set the federate information at that point, which - while functionally identical to setting the federate info in either the federate config file or in the federate info object - provides integrators with additional flexibility, which can be useful particularly if some settings need to be changed dynamically during the co-simulation. The API calls are syntactically very similar to the API calls for setting up the federate info object, except instead they target the federate itself. For example, to revisit the above example where the only_transmit_on_change flag is set to true in the federate info object, if operating on an existing federate, that call would be:

```
h.helicsFederateSetFlagOption(fi, 6, True)
```

Error Handling

By default, HELICS will not terminate execution of every participating federate if an error occurs in one. However, in most cases, if such an error occurs, the co-simulation is no longer valid. It is therefore generally a good idea to set the following flag in your simulator federate so that its execution will be terminated if an error occurs anywhere in the co-simulation.

```
h.helicsFederateSetFlagOption(fed, helics_flag_terminate_on_error)
```

This will prevent your federate from hanging in the event that another federate fails.

Collecting the Publication, Subscription and Endpoint Objects

Having configured the publications, subscriptions and endpoints and registered this information with HELICS, the channels for sending and receiving this information have been created within the co-simulation framework. If you registered the publication, subscriptions and endpoints within your code (i.e., using HELICS API calls), you already have access to each respective object because it was returned when you made the registration call. However, if you created your federate using a configuration file which contained all of this information, you now need to retrieve these objects from HELICS so that you can invoke them during the execution of your co-simulation. The following calls will allow you to query HELICS for the metadata associated with each publication. Similar calls can be used to get input (or subscription) and endpoint information.

```
pub_count = h.helicsFederateGetPublicationCount(fed)
pub = h.helicsFederateGetPublicationByIndex(fed, index)
pub_key = h.helicsPublicationGetKey(pub)
```

The object returned when the helicsFederateGetPublicationByIndex() method is invoked is the interface object used for retrieving other publication metadata (as in the helicsPublicationGetKey() method) and when publishing data to HELICS (as described in the execution section below).

Federate Execution

Once the federate has been created, all subscriptions, publications and endpoints have been registered and, all the federate information has been appropriately set, it is time to enter executing mode. This can be done with the following API call:

```
h.helicsFederateEnterExecutingMode(fed)
```

This method call is a blocking call; your custom federate will sit there and do nothing until all other federates have also finished any set-up work and have also requested to begin execution of the co-simulation. Once this method returns, the federation is effectively at simulation time of zero.

At this point, each federate will now step through time, exchanging values with other federates in the co-simulation as appropriate. This will be implemented in a loop where each federate will go through a set of prescribed steps at each time step. At the beginning of the co-simulation, time is at the zeroth time step (t = 0). Let's assume that the co-simulation will end at a pre-determined time, t = max_time. The nature of the simulator will dictate how the time loop is handled. However, it is likely that the co-simulation loop will start with something like this:

```
t = 0
while t < end_time:
    pass  # co-simulation code would go here
```

Now, the federate begins to step through time. For the purposes of this example, we will assume that during every time step, the federate will first take inputs in from the rest of the co-simulation, then make internal updates and calculations and finish the time step by publishing values back to the rest of the co-simulation before requesting the next time step.

Get Inputs

The federate will first listen on each of its inputs (or subscriptions) and endpoints to see whether new information has been sent from the rest of the federation. The first code sample below shows how information can be retrieved from an input (or subscriptions) through HELICS API calls by passing in the subscription object. As can be seen, HELICS has built in type conversion (where possible) and regardless of how the sender of the data has formatted it, HELICS can present it as requested by the appropriate method call.

```
int_value = h.helicsInputGetInteger(sub)
float_value = h.helicsInputGetDouble(sub)
real_value, imag_value = h.helicsInputGetComplex(sub)
string_value = h.helicsInputGetChar(sub)
...
```

It may also be worth noting that it is possible on receipt to check whether an input has been updated before retrieving values. That can be done using the following call:

```
updated = h.helicsInputIsUpdated(sid)
```

Which returns true if the value has been updated and false if it has not.

Receiving messages at an endpoint works a little bit differently than when receiving values through a subscription. Most fundamentally, there may be multiple messages queued at an endpoint while there will only ever be one value received at a subscription (the last value if more than one is sent prior to being retrieved). To receive all of the messages at an endpoint, they needed to be popped off the queue. An example of how this might be done is given below.

```
while h.helicsEndpointPendingMessages(end) > 0:
    msg_obj = h.helicsEndpointGetMessageObject(end)
```

To get the source of each of the messages received at an endpoint, the following call can be used:

```
msg_source = h.helicsMessageGetOriginalSource(msg_obj)
```

Internal Updates and Calculations

At this point, your federate has received all of its input information from the other federates in the co-simulation and is now ready to run whatever updates or calculations it needs to for the current time step.

Publish Outputs

Once the new inputs have been collected and all necessary calculations made, the federate can publish whatever information it needs to for the rest of the federation to use. The code sample below shows how these output values can be published out to the federation using HELICS API calls. As in when reading in new values, these output values can published as a variety of data types and HELICS can handle type conversion if one of the receivers of the value asks for it in a type different than published.

```
h.helicsPublicationPublishInteger(pub, int_value)
h.helicsPublicationPublishDouble(pub, float_value)
h.helicsPublicationPublishComplex(pub, real_value, imag_value)
h.helicsPublicationPublishChar(pub, string_value)
...
```

For sending a message through an endpoint, that once again looks a little bit different, in this case because - unlike with a publication - a message requires a destination. If a default destination was set when the endpoint was registered (either through the config file or through calling `h.helicsEndpointSetDefaultDestination()`), then an empty string can

be passed. Otherwise, the destination must be provided as shown in API call below where dest is the destination and msg is the message to be sent.

```
h.helicsEndpointSendMessageRaw(end, dest, msg)
```

The fundamental topics listed below cover the material necessary to build a fully functional co-simulation with HELICS, written for users who have little to no experience with co-simulation. Each section makes reference to the *Fundamental Examples* to allow the user to scaffold their learning with concrete and detailed examples. After working through the topics below, the user should be able to write their own simple co-simulation in Python with PyHELICS and understand how access resources improve the development of their co-simulations.

The topics considered "fundamental" to building a co-simualtion with HELICS are:

- ***HELICS Terminology*** - Key terms and concepts to understand before running co-simulations with HELICS
- ***Federates*** - Discussion of the different types of federates in HELICS and how to configure them.
  - *Value Federates*
  - *Message Federates*
  - *Filters*
- ***Federate Interface Configuration*** - How to connect an existing simulator with HELICS
  - ***With JSON config file***
  - ***With HELICS APIs***
- ***Timing Configuration*** - How HELICS coordinates the simulation time of all the federates in the federation
  - *Timing Exercise*
  - *Timing Exercise answers*
- ***Stages of the Co-simulation***
- ***Logging*** - Discussion of logging within HELICS and how to control it.
- ***Execution with `helics run ...`*** - The HELICS team has developed a standardized means of running HELICS co-simulations.
- ***Simulator Integration*** - A guide for integrating HELICS into simulators.

2.3.3 Advanced Topics

Aliases

Aliasing in HELICS allows an interface (publication, input, endpoint, filter, translator) to be linked via a different name than the one given on registration. This can be used to simplify commonly used names or interfaces, or match up federates with different naming conventions. Aliases can be defined from anywhere in the co-simulation (like connections), including via config files.

API

Cores, brokers, and Federates all have a single API call

```
addAlias(std::string_view interfaceName, std::string_view alias);
```

In the C and language API's the call will have a form such as:

```
helicsXXXAddAlias(const char * interfaceName, const char * alias);
```

Rules

The `addAlias` operation creates an equivalence relationship between the two strings. As such `addAlias("string1", "string2")` is equivalent to `addAlias("string2","string1")` in terms of the overall co-simulation connectivity. That being said there is a small performance difference between the two and it generally is recommended that the first string be an existing interface name, and the underlying interface be created before the alias. Once again, everything will work in other orders but there is a performance difference.

In general, all operations are allowed unless they create a situation which would map a single string to multiple interfaces of the same type, if that situation occurs a global error is generated and the co-simulation fails.

Multiple aliases are allowed including aliases of aliases.

File Configuration

In addition to the API calls, aliases can be specified in federation and connection configuration files in either TOML or JSON. The keyword is "aliases" followed by an array of string pairs, which is the exact same structure as specifying "globals"

```
aliases=[["comboFed/pub2","dpub"],["cfed_agasagadsag","c2"]]
```

```
"aliases": [
    ["comboFed/pub2", "dpub"],
    ["cfed_agasagadsag", "c2"]
]
```

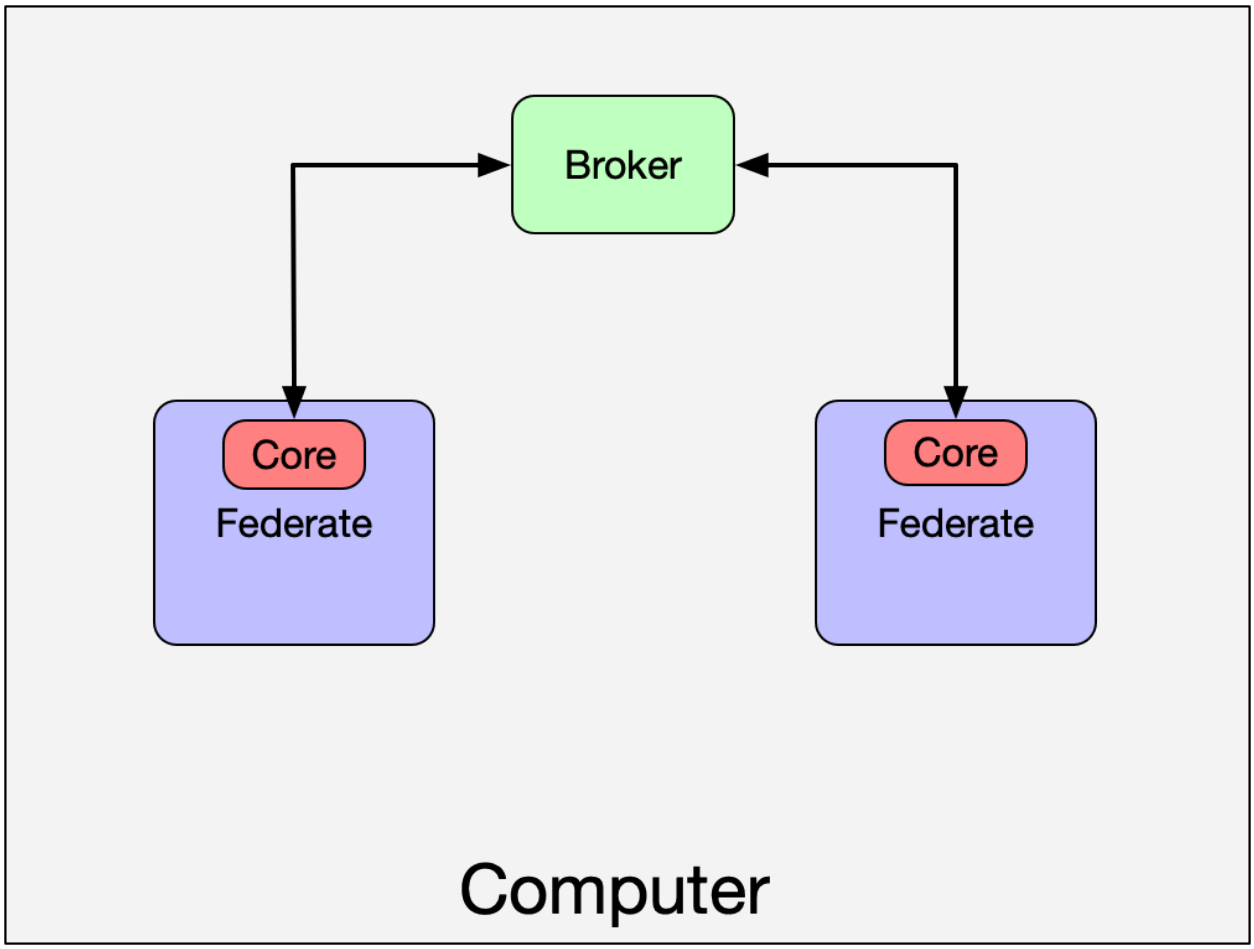
Co-simulation Architectures

There are several co-simulation architectures that can be constructed where the relationships between the federates, cores, and brokers can vary.

Simple Co-simulation

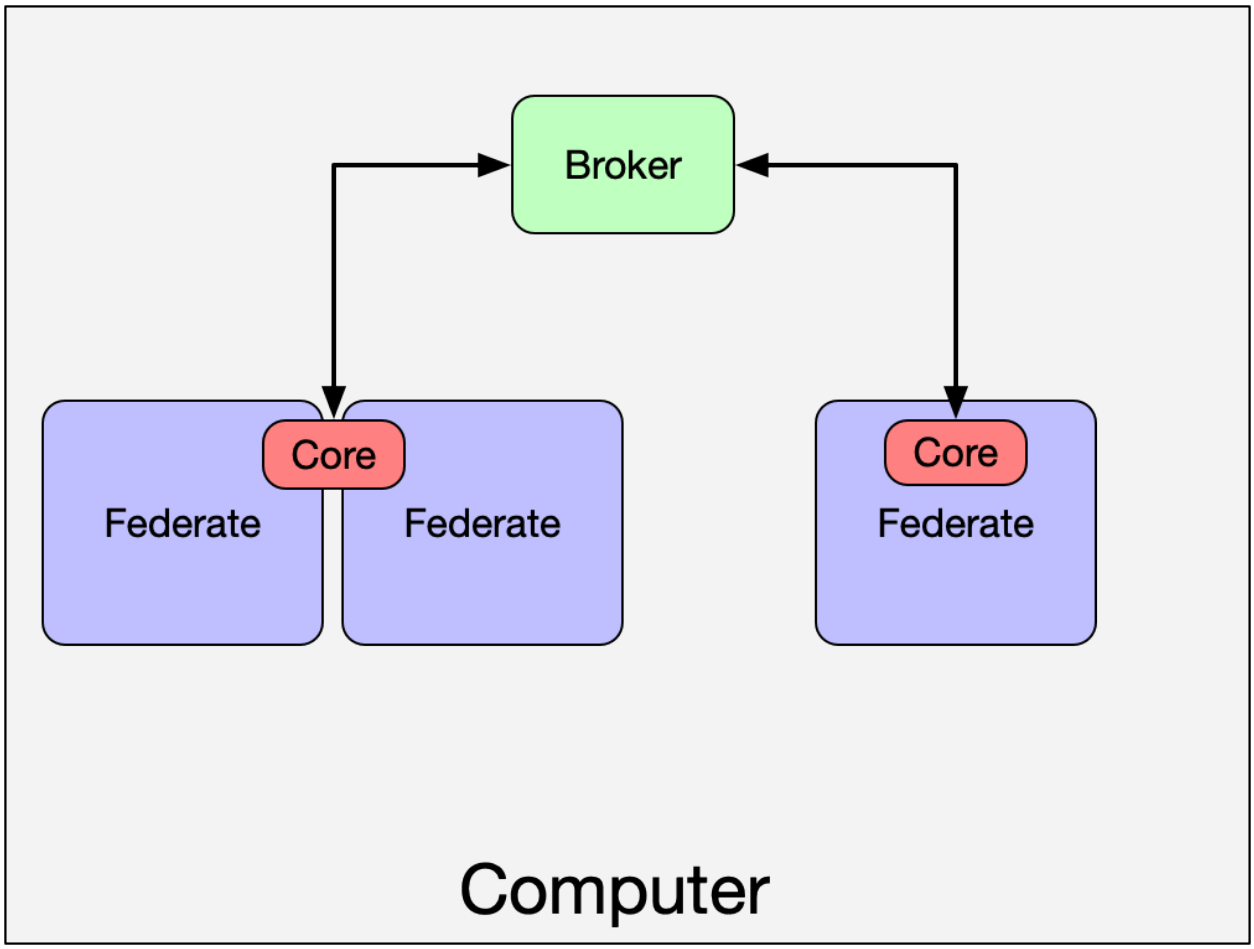
Broker topology is somewhat optional for simple co-simulations, but offers an increase in performance if it is possible to identify groups of federates that interact often with each other but rarely with the rest of the federation. In such cases, assigning that group of federates their own broker will remove the congestion their messages cause with the federation as a whole. The Fundamental Topics and Examples are built with a single broker.

The figure below shows the most common architecture for HELICS co-simulation. Each core has only one federate as an integrated executable, all executables reside on the same computer and are connected to the same broker. This architecture is particularly common for small federates and/or co-simulations under development. This is also the architecture for the *Fundamental Examples*.



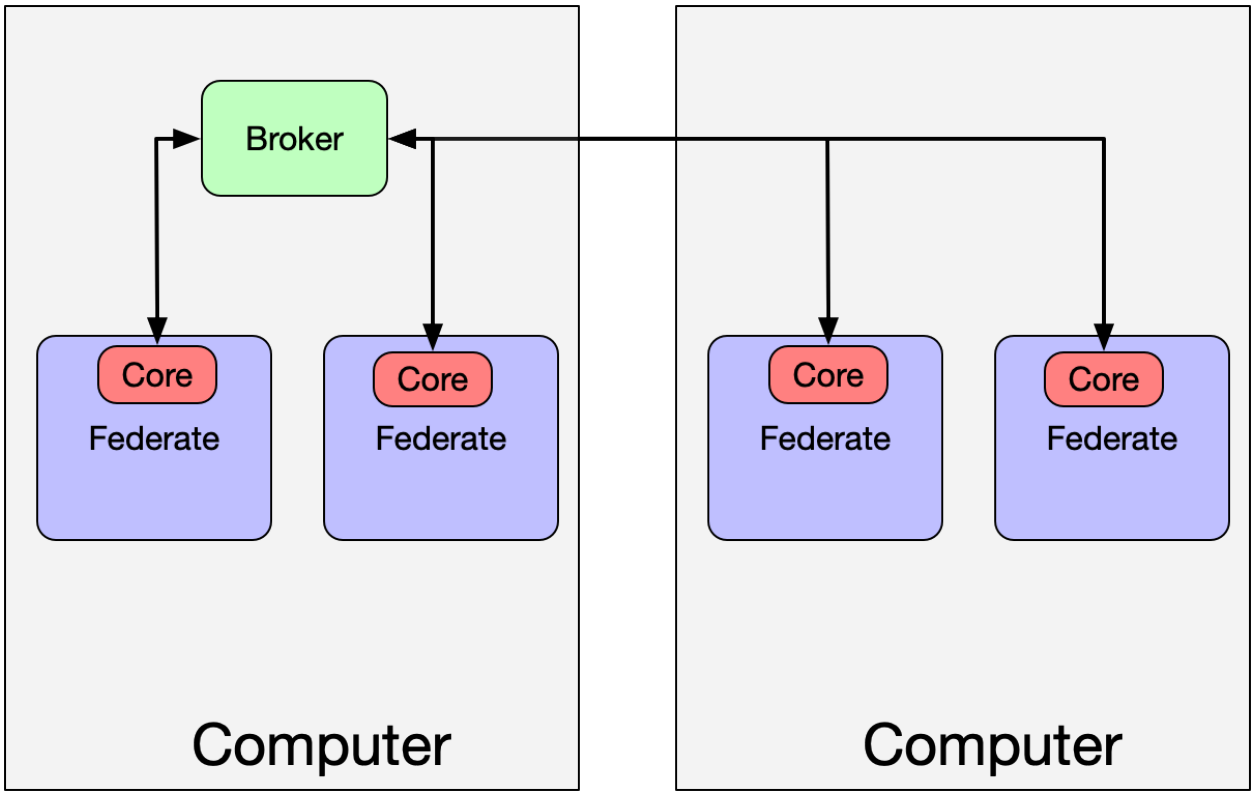
Multiple Federates on a Single Core

The architecture below shows a scenario where more than one federate is associated with a single core. For most simulators that have already been integrated with HELICS this architecture will generally not be used. For simulators that are multi-threaded by nature and typically represent multiple independent simulated entities (federates to HELICS), HELICS can be configured to facilitate message passing between threads. For a co-simulation that exists entirely within a single executable, this architecture will provide the highest performance. For example, if a large number of small controllers are written as a single, multi-threaded application (perhaps all the thermostats in a large commercial building are being managed by a centralized controller), particularly where there is communication between the federates, using a single core inside a single multi-threaded application (with typically one thread per federate) will provide the highest level of performance.



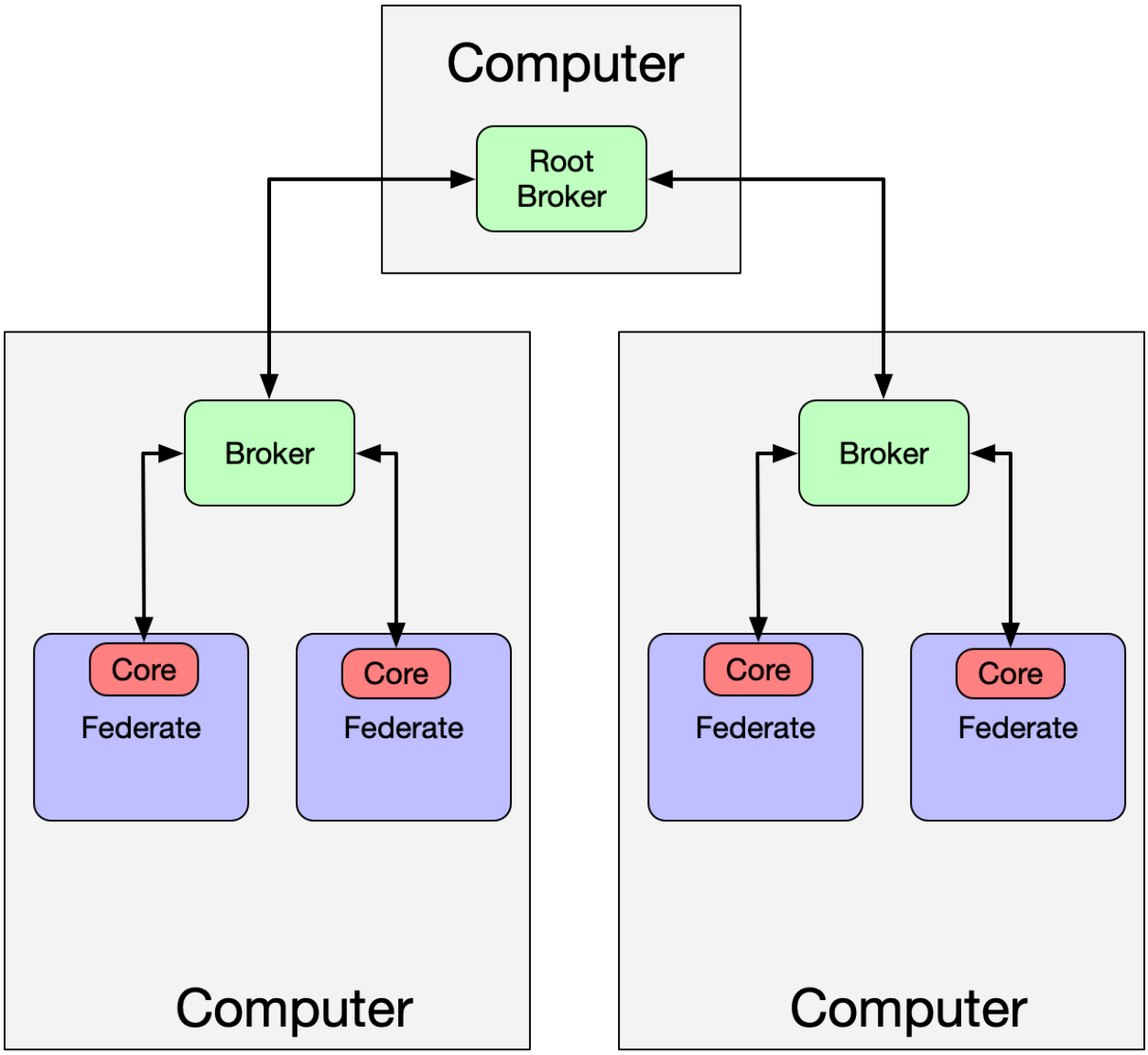
Computationally Heavy Federates

For co-simulations on limited hardware where a federate requires significant computational resources and high performance is important, it may be necessary to spread the federates out across a number of compute nodes to give each federate the resources it needs. All federates are still connected to a common broker and it would be required that the computers have a valid network connection so all federates can communicate with this broker. In this case, it may or may not be necessary to place the broker on its own compute node, based on the degree of competition for resources on its current compute node.



Multiple brokers

Alternatively, it would be possible to locate a broker on each computer and create a root broker on a third node. This kind of architecture could help if higher performance is needed and the federates on each computer primarily interact with each other and very little with the federates on the other computer. As compared to the previous architecture, adding the extra layer of brokers would keep local messages on the same compute node and reduce congestion on the root broker. An overview of how this is constructed is provided in the *section on broker hierarchies* and an example of this architecture (though running on a single compute node for demonstration purposes), is shown in the *broker hierarchy example*



Broker Hierarchies

The simplest and most straightforward way HELICS co-simulations are constructed is with a single broker. If all federates are running on a single compute node, a single broker is likely all you'll need. In situations where the co-simulation is running across multiple compute nodes, with a little bit of planning, the use of a hierarchy of brokers can help improve co-simulation performance.

Why Multiple Brokers?

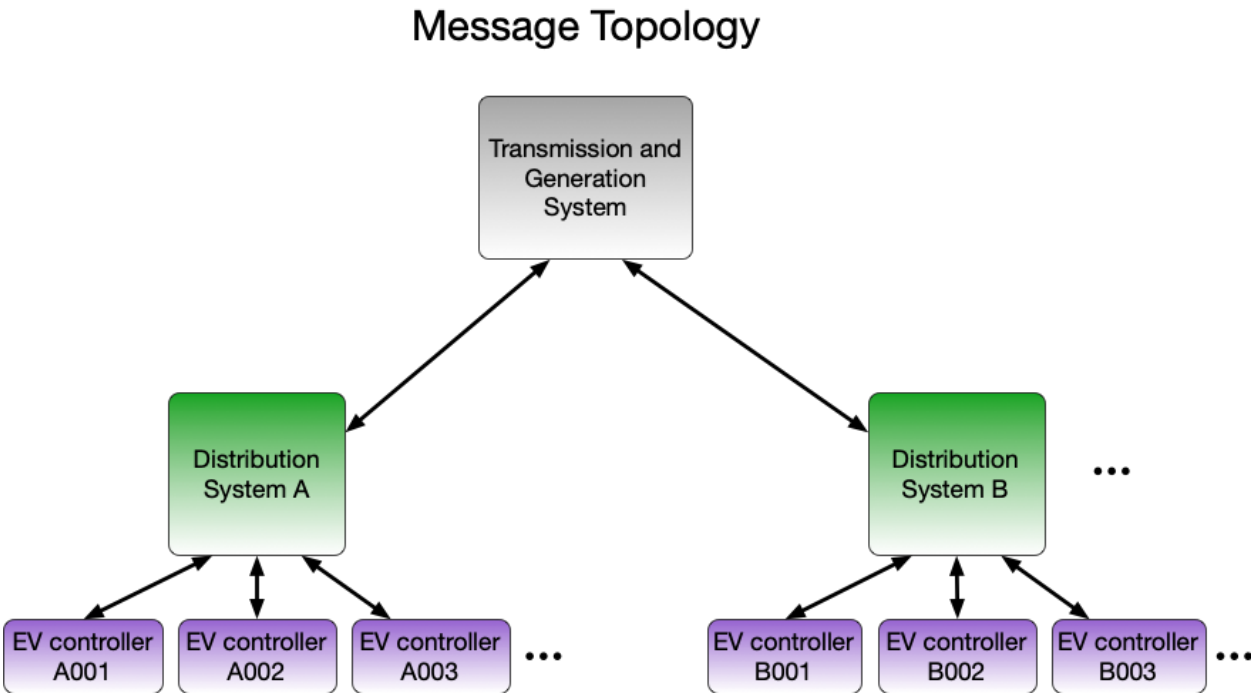
Brokers are primarily concerned with facilitating the message exchanges between federates. As the number of messages that must be passed increase, the load on the broker(s) increase as well. In cases where the co-simulation is deployed across multiple compute nodes, the total cost of sending messages over the network (primarily in terms of latency and the corresponding slowdown in federation performance) can become non-trivial. The worst case develops when the compute node hosting the broker is physically distant (and thus, experiences higher latency) from the compute nodes where the individual federates are running. If two federates running on the same node need to talk to each other but have a high-latency connection to the broker, their communication will be significantly hampered as they coordinate with that distant broker.

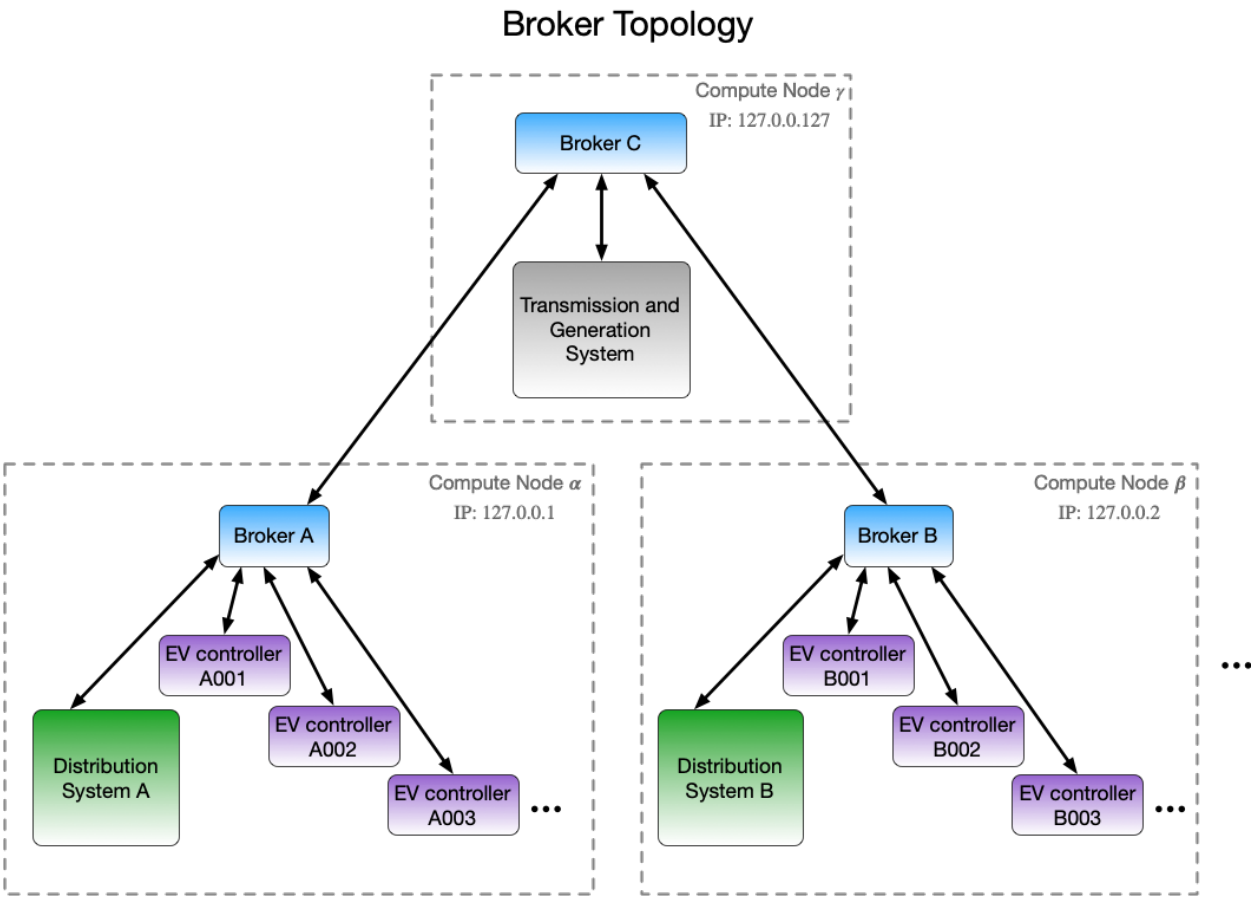
And thus the solution presents itself: a broker local to the compute node where the federates are located will have a very low latency connection and will generally experience a lower load of messages it needs to process. To receive this benefit, though, the co-simulation needs to be deployed to the compute nodes in such a way that the federates that talk to each other most frequently/heavily are located on the same node and a broker is also deployed to that node.

When federates join the federation they can be assigned to specific brokers and when the co-simulation begins, each local broker uses this information to route the messages it can. Any messages that it is not connected to, it sends up the broker hierarchy. The broker at the top of the hierarchy is the broker of last resort and it is guaranteed to be able to route all messages down the hierarchy to their intended destination.

So, using an example we've seen several times, imagine a scenario where a single transmission system covering the western US is being simulated with many, many individual neighborhood level power systems attached to that regional system and controllers manage a hypothetical fleet of electric vehicles (EVs) whose owners are in these neighborhoods. The EV controllers and the distribution system federates interact frequently and the distribution system federates and the transmission system federates also interact frequently.

The diagrams below show the message and broker topologies for this hypothetical example.





Broker Hierarchy Implementation

To implement a broker hierarchy, modifications to the configuration files for the federates and command line options for the brokers need to be made.

For example, the config JSON for the Distribution System A (where Broker A is at IP 127.0.0.1) would look something like this:

```
{
  "name": "DistributionSystemA",
  "coreInit": "--broker_address=tcp://127.0.0.1"
}
```

The command line for launching Broker A also needs to be adjusted. For this example, let's assume there are 200 total federates expected by Broker A and any sub-brokers below it. Note that `broker_address` is used to define the address of the broker next up in the hierarchy; in this case, Broker C.

```
$ helics_broker -f200 --broker_address=tcp://127.0.0.127
```

The JSON config file for the Transmission and Generation System federate needs to indicate where its broker (Broker C) is located (IP 127.0.0.127):

```
{
  "name": "TransmissionGenerationSystem",
  "coreInit": "--broker_address=tcp://127.0.0.127"
```

(continues on next page)

(continued from previous page)

```
}
```

Lastly, when Broker C, the root broker, is instantiated, it may optionally specify the number of direct child brokers that are expected to connect to it as well as the total number of federates in its full branch of the hierarchy. If Broker C has one local federate and two child brokers, each managing 200 federates, the `-f` value for Broker C is 401. Since it is the root broker, there is no parent broker address to specify.

```
$ helics_broker -f401 --local_subbrokers=2
```

The `--federates/-f` option always counts the total federates known through that broker, including federates forwarded from sub-brokers. When you need to wait only for federates attached through direct child cores, use `--local_federates`. When you need to wait for a number of direct child brokers, use `--local_subbrokers`. To wait for specific federates by name, use `--required_federates=name1,name2`.

Hierarchies with Complex Networks

In more complex networking environments (*see dedicated documentation*), it may be necessary to include an additional specification of the interface the broker, sub-broker, or federate would like to talk to the rest of the federation on. In these cases, typically only a port specification is added to the configuration. Adding the port can be done in one of two ways:

1. Appending a `:<port number>` after the IP address (*e.g.* `tcp://127.0.0.1:23405`)
2. Using `broker_port=<broker_port>` and/or `local_port=<local_port>`

`broker_port` is used to specify the port on which a broker or sub-broker should talk to its broker. `local_port` is used to define which port a broker, sub-broker, or federate will be listening on for its communication with the rest of the federation. For example, a federate may define its `local_port` as 23500; as HELICS is setting-up the federation anything that needs to talk to the federate in question will know to use port 23500. Again, this can be particularly important if firewalls and networking policies are making it difficult or impossible to connect in using the defaults.

Lastly, there is a full IP and/or socket specification that is analogous to `local_port`: `local_interface`. This is analogous to `broker_address` in that it allows the specification of an IP and/or a socket and is the complement to `local_port` in that it specifies how the federation could talk to the federate, broker, or sub-broker that is specifying it.

Using the above example, if the networking environment were more complex, the Distribution System A configuration could have been extended as follows to force the rest of the federation to talk to it on port 23500 as well as indicate the port its broker wants to use for communication (25000):

```
{
  "name": "DistributionSystemA",
  "coreInit": "--broker_address=tcp://127.0.0.1:25000",
  "local_port": 23500
}
```

Similarly, Broker A could have been instantiated to force communication on certain ports:

```
$ helics_broker -f200 --broker_address=tcp://127.0.0.127:24000 --local_port=25000
```

Adding the port number to the broker address indicates Broker C (the broker for Broker A) should be contacted on port 24000 and that the rest of the federation (including Broker C) should contact Broker A on port 25000. Similarly, the transmission federate would need similar additional specification to contact Broker C. Note that the port number is just appended to the IP; this could have also been specified with a new line in the JSON file defining `broker_port`.

```
{
  "name": "TransmissionGenerationSystem",
  "coreInit": "--broker_address=tcp://127.0.0.127:24000"
}
```

Broker B would need the same details so it could contact its parent broker and could additionally specify a unique port for its federates to contact it on:

```
$ helics_broker -f200 --broker_address=tcp://127.0.0.127:24000 --local_port=27000
```

Lastly, to complete the configuration implied by the above, Broker C would need to be called like this:

```
$ helics_broker -f401 --local_subbrokers=2 --local_port=24000
```

Example

A full co-simulation example showing how to implement a broker hierarchy *has been written up over here* (and the source code is in the HELICS Examples repository).

Callbacks

Federates have a number of callbacks that can be specified that trigger under different stages of the co-simulation operation or conditions. These callbacks can be used to simplify the management of a federate or enable new capabilities to be integrated with HELICS. This document is a listing and description of the callbacks available on Value, Message, and Combination Federates. Callbacks are user-specified code that is executed inline with other HELICS operations.

In C++, callbacks generally take a std::function object which can be structured as a lambda or direct object. In C and language API's the callback is structured as a function pointer, and pass through a userData object into the callback. HELICS does not manipulate the userData and simply passes it through.

Specific purpose callbacks

Some callbacks respond to specific events from a federate this includes logging, queries, and errors.

Logging Callback

The logging callback allows a specific user operation to handle log messages

```
void setLoggingCallback(
        const std::function<void(int, std::string_view, std::string_view)>& logFunction);
```

```
void
    helicsFederateSetLoggingCallback(HelicsFederate fed,
                                     void (*logger)(int loglevel, const char* identifier,
↪ const char* message, void* userData),
                                     void* userdata,
                                     HelicsError* err);
```

The main arguments to the callback are a loglevel integer code corresponding to log levels in HELICS, a string identifier, and a log message.

Query Callback

The query callback allows a federate to respond to custom queries. If a federate receives a query that it does not know the answer to, it executes the query callback if supplied. In C++ the return type is a string, in C and the other language API's the user supplied callback is expected to fill in a query buffer.

```
void setQueryCallback(const std::function<std::string(std::string_view)>& queryFunction);
```

```
void
    helicsFederateSetQueryCallback(HelicsFederate fed,
                                   void (*queryAnswer)(const char* query, int querySize,␣
↪HelicsQueryBuffer buffer, void* userdata),
                                   void* userdata,
                                   HelicsError* err);
```

Error Handler Callback

The error handler callback will be executed when an error is encountered and includes arguments for an integer error code and an error message.

```
void setErrorHandlerCallback(std::function<void(int, std::string_view)>␣
↪errorHandlerCallback);
```

```
void helicsFederateErrorHandlerCallback(HelicsFederate fed,
                                                    void (*errorHandler)(int errorCode,
↪ const char* errorString, void* userdata),
                                                    void* userdata,
                                                    HelicsError* err);
```

LifeCycle Callbacks

Life Cycle callbacks occur at specific stages or transitions in a co-simulation

Initializing Mode Entry

The InitializingEntry callback is executed when moving into initializing mode. The boolean parameter indicates whether this is iterative. It is set to false the first time this callback is executed (when moving from CREATED mode) and true if returning to this mode from an enterExecutingModeIterative Call.

```
void setInitializingEntryCallback(std::function<void(bool)> callback);
```

```
void helicsFederateInitializingEntryCallback(HelicsFederate fed,
                                                          void␣
↪(*initializingEntry)(HelicsBool iterating, void* userdata),
                                                          void* userdata,
                                                          HelicsError* err);
```

Executing Entry

This callback is executed exactly once, when moving from INITIALIZING to EXECUTING mode and has no parameters.

```
void setExecutingEntryCallback(std::function<void()> callback);
```

```
void
    helicsFederateExecutingEntryCallback(HelicsFederate fed, void␣
↪(*executingEntry)(void* userdata), void* userdata, HelicsError* err);
```

Time Request Entry

This callback is executed on a time request call, prior to calling the blocking Core API call. The arguments are the current time of the federate, the requested time, and whether an iterative call is being made.

```
void setTimeRequestEntryCallback(std::function<void(Time, Time, bool)> callback);
```

```
void helicsFederateSetTimeRequestEntryCallback(
    HelicsFederate fed,
    void (*requestTime)(HelicsTime currentTime, HelicsTime requestTime, HelicsBool␣
↪iterating, void* userdata),
    void* userdata,
    HelicsError* err);
```

Time Update

The time update callback is executed after the Core API returns from a time request, and prior to any value based callbacks executing or having been updated. The arguments are the new time and the boolean argument will be set to true if this is an iterative time.

```
void setTimeUpdateCallback(std::function<void(Time, bool)> callback);
```

```
void helicsFederateSetTimeUpdateCallback(HelicsFederate fed,
                                                    void (*timeUpdate)(HelicsTime␣
↪newTime, HelicsBool iterating, void* userdata),
                                                    void* userdata,
                                                    HelicsError* err);
```

Time Request Return

The Time request return callback will execute as the last operation prior to return from a time request. It executes after all value based callbacks, and like the Time Update contains arguments for the new time and an indicator if the time is iterative.

```
void setTimeRequestReturnCallback(std::function<void(Time, bool)> callback);
```

```
void
    helicsFederateSetTimeRequestReturnCallback(HelicsFederate fed,
                                               void (*requestTimeReturn)(HelicsTime␣
↪newTime, HelicsBool iterating, void* userdata),
                                               void* userdata,
                                               HelicsError* err);
```

Mode Update

The mode update callback executes whenever the Mode of the federate changes. It will execute prior to timeUpdate-Callback when both would be called. The arguments are the old and new Modes.

```
void setModeUpdateCallback(std::function<void(Modes, Modes)> callback);
```

```
void
   helicsFederateSetStateChangeCallback(HelicsFederate fed,
                                        void (*stateChange)(HelicsFederateState␣
↪newState, HelicsFederateState oldState, void* userdata),
                                        void* userdata,
                                        HelicsError* err);
```

NOTE: notice the different names between the C and C++ API's, this discrepancy is noted and will be corrected in a future release, likely a potential HELICS 4.0 but for now retains the consistency internal to the individual API's.

Termination

This callback will execute exactly once when the finalize or error state is reached.

```
void setCosimulationTerminatedCallback(std::function<void()> callback);
```

```
void helicsFederateCosimulationTerminationCallback(HelicsFederate fed,
                                                   void␣
↪(*cosimTermination)(void* userdata),
                                                   void* userdata,
                                                   HelicsError* err);
```

Value Federate Callbacks

These callbacks will execute when an input is updated and can be general for all inputs or specific to a single one. This callback is not currently available in the C API.

```
   void setInputNotificationCallback(std::function<void(Input&, Time)> callback);

   void setInputNotificationCallback(Input& inp, std::function<void(Input&, Time)>␣
↪callback);
```

Message Federate Callbacks

These callbacks will execute when an endpoint is updated and can be general for all endpoints or specific to a single one. This callback is not currently available in the C API.

```
void setMessageNotificationCallback(const std::function<void(Endpoint&, Time)>&␣
↪callback);

   void setMessageNotificationCallback(const Endpoint& ept,
                                       const std::function<void(Endpoint&, Time)>&␣
↪callback);
```

Callback Federates

Starting in HELICS v3.3 Callback federates were added as way to entirely inline the operation of a federate, allowing significant increases in the number of federates that could partake in a co-simulation for a given system size. Callback federates are a specific type of federate that adds some additional requirements when implementing a federate. This includes a couple new callbacks and properties not available in a regular federate.

Expected Use Cases

Callback federates allow the potential for a large number of small federates to execute in a core, up to 131,072 in each core. Trying this with normal federates rapidly exceeds the thread capacity of a typical system, so in the cases where the federate can be defined in terms of callbacks and has only a few simple connections and minimal computation the callback federate will be much more efficient. Since all callback federates execute serially internally to the core, the callback federate is not appropriate for federates with significant computation load as it will block the execution of the other callback federates.

Additional Callbacks

The Callback Federate adds a few additional callbacks necessary for operation.

Initialize

The initialize callback's purpose is to allow a callback federate to attempt to enter executing mode iteratively if desired. The return value is an IterationRequest enumeration. If the callback is not specified for a callback federate NO_ITERATIONS is assumed.

```
void setInitializeCallback(std::function<IterationRequest()> initializeCallback);


void helicsCallbackFederateInitializeCallback(HelicsFederate fed,
                                                HelicsIterationRequest␣
↪(*initialize)(void* userdata),
                                                void* userdata,
                                                HelicsError* err);
```

Time requests

The purpose of the following callbacks is to allow a user to specify what the next time request should be. This request can be a simple time value (integer) or an iteration_time if iterations are needed. If the iteration time callback method is specified it overrides the simpler callback. The callback must be cleared if desired to revert either through a NULL object or the `clearNextTimeCallbacks` operation.

```
void setNextTimeCallback(std::function<Time(Time)> nextTimeCallback)

void helicsCallbackFederateNextTimeCallback(HelicsFederate fed,
                                              HelicsTime␣
↪(*timeUpdate)(HelicsTime time, void* userdata),
                                              void* userdata,
                                              HelicsError* err);
```

```
void setNextTimeIterativeCallback(
        std::function<std::pair<Time, IterationRequest>(iteration_time)>␣
↪nextTimeCallback);
```

(continues on next page)

(continued from previous page)

```
void helicsCallbackFederateNextTimeIterativeCallback(
    HelicsFederate fed,
    HelicsTime (*timeUpdate)(HelicsTime time, HelicsIterationResult,␣
↪HelicsIterationRequest* iteration, void* userdata),
    void* userdata,
    HelicsError* err);
```

There is also a method to clear the time callbacks.

```
void clearNextTimeCallback();
```

All callback are optional.

Properties

Callback federates define an additional property `HELICS_PROPERTY_TIME_MAXTIME`. This property allows the callback federate to finalize once a time returned is greater or equal to the specified maxtime.

Other key properties

Specifying a period `HELICS_PROPERTY_TIME_MAXTIME` will allow the callback federate to automatically compute the next time request with no additional callback for the time computation.

Specifying that the federate is event-driven will send the next time request as `HELICS_TIME_MAXTIME` which will grant only when there is some new data to process. This can specified with `HELICS_FLAG_EVENT_TRIGGERED`.

Execution

Callback federates are Combination federates so any operation that works on Combination federates works on Callback federates. Callback federates behave identically to regular federates until a call is made to `enterInitializingMode()`. In regular federates this call is blocking; in Callback federates this call is an asynchronous call and will transfer control of the federate to the HELICS core. All other blocking calls in HELICS will produce an error. All callbacks defined in a regular federate are available for use and are expected to be used. The additional callbacks defined here are to control the next time step and iterations which are directly user controlled in normal federates.

Command Interface

The command interface for HELICS was introduced in HELICS 3. It is an asynchronous communication mechanism to send commands or other information to other components. Cores and Brokers will respond to a small subset of commands known by HELICS, but federates have a command interface to allow retrieval of commands for interpretation by a federate.

The general function appears like

```
void sendCommand(const std::string& target, const std::string& commandStr,
                              HelicsSequencingModes mode)
```

the same function is available for federates, cores, and brokers. Sequencing Mode determines the priority of the command and can be either

- `HELICS_SEQUENCING_MODE_FAST` : send on priority channel
- `HELICS_SEQUENCING_MODE_ORDERED` : send on normal channels ordered with other communication

- HELICS_SEQUENCING_MODE_DEFAULT : use HELICS determined default mode

```
helicsFederateSendCommand(HelicsFederate fed, const char* target, const char* command,␣
↪HelicsError* err)
```

All commands in C are sent with the default ordering for now. The use case for ordered commands is primarily testing for the time being so the interface has not been added to the C API as of yet.

Targets

A target is specified, and can be one of the following. A federate named one of the key words is valid for the federation, but cannot be queried using the name.

target	Description
broker	The first broker encountered in the hierarchy from the caller
root, federation	The root broker of the federation
core	The core of a federate, this is not a valid target if called from a broker
<object name>	any named object in the federation can also be queried, brokers, cores, and federates

Command String

The commandStr is a generic string, so can be anything that can be contained in a string object. It is expected that most command strings will have a json format, though a few simple ones are just plain strings.

HELICS supported commands

The following queries are defined directly in HELICS. Federates may specify a callback function which allows arbitrary user-defined queries. The queries defined here are available inside of HELICS.

Command String	Description
terminate	[all objects] disconnect the object from the federation
echo	[all objects] send a command with a *command-Str* `= `*echo_reply* back to the sender
log <string>	[all objects] generate a log message in a particular object
logbuffer <size>	[all objects] set the log buffer to a particular size or *stop*
monitor <args...>	[brokers] set up a federate as the time monitor <args...> = <federate names> <logperiod>
set barrier <args...>	[brokers,federates] set a time barrier <args...> = <time> <barrier_id*>
clear barrier <id*>	[brokers,federates] clear a time barrier <barrier_id*>
remotelog <level>	**[all objects] instruct the object to send log messages to a remote location in addition to local** logging. The <level> is a [log level string](../fundamental_topics/logging.md) or *stop*
command_status	[federates] when received will send a string back to the source of the command looking like "X unprocessed commands" where X is the number of unprocessed commands

* argument is optional

Future

How this will get used is somewhat up in the air yet. It is expected that future commands to the objects will help support debugging and other diagnostics but beyond that it is expected to evolve considerably.

Usage Notes

Commands that must traverse the network travel along priority paths unless specified with the `HELICS_SEQUENCING_MODE_ORDERED` option in the C++ API.

Application API

There are two basic calls in the application API as part of a federate object. To retrieve a command addressed to a federate there are two commands

```
std::pair<std::string,std::string>     getCommand();
std::pair<std::string,std::string>     waitCommand();
```

The first will return immediately but the strings will be empty if there is no command available. The second is a blocking call and will only return if a command is available.

Equivalent calls in the C API are

```
const char *helicsFederateGetCommand(HelicsFederate fed, HelicsError *err);
const char *helicsFederateWaitCommand(HelicsFederate fed, HelicsError *err);
```

The only error paths are if the federate is not valid or not in a state to receive commands. The python calls are similar to other python calls.

Only one command is returned per use of the `helicsFederateGetCommand()` or `helicsFederateWaitCommand()` API calls. It is possible multiple commands have been queued and to retrieve all of them, the API must be called multiple times. An API call when the command queue is empty will return a null string.

Connector Federate

For some co-simulation workflows or software architectures, it is convenient for the mapping of HELICS interfaces (publications, inputs, and endpoints) between federates to be managed outside the federate configuration. For example, federates could be configured to expose all their interfaces and allow another software entity to actually make the mapping between the interfaces using a definition that can change based on the use case. This allows the federates to remain untouched and effectively be use-case independent (at least as far as their interfaces go) while still having their connections in the federation adjusted as a given use case requires.

To support this HELICS comes with a connector application called "helics_connector" that uses a "match-file" to create the connections between federates.

"helics_connector" Federate

The role of the "helics_connector" federate is to evaluate a mapping of the connections between federate interfaces and create the connections between these interfaces. Under the hood, it's using public APIs that allow the sources ("targets") for inputs and endpoints to be defined and destinations ("targets") to be added to publications. (Which is to say, if the stock "helics_connector" federate doesn't work the way you would like, you could actually make your own.) The documentation of the "helics_connector" app can be found on its *dedicated documentation page* but I'll summarize the highlights here:

- Matches are defined in a plain text or JSON structured text file.

- Matches can be defined as one-to-one connections or use regular expressions to match many interfaces in one succinct line.
- Tags can be used to add an extra dimension to the matching process, only allowing federates with certain tags to be have their interfaces be candidates for the matching process.
- If federates support it, the connector can query a federate about the interfaces it could make and then, based on the results of the matching operation, command said federate to create the necessary interfaces.

Callback Complications

The "query-and-command" form of operation is the most complex, largely because it requires the implementation of callback functions. Unless you're using the C API directly, implementing the callback functions takes a little bit of extra effort to get the C-based libraries under the hood working. Since most of the User Guide examples are in Python and most of our users are using Python, we've documented callbacks in PyHELICS, have a *general page discussing the supported callbacks*, and use them in *this example*. Take a look at these pages for further details to get a slightly better understanding why C-based callbacks are more complicated in Python.

"helics_connector" Interface creation

The order in which "helics*connector" creates the interfaces in the federate is not deterministic. This is especially important when using the APIs that get the interface by index (_e.g* `helicsFederateGetInputByIndex`, `helicsFederateGetPublicationByIndex`, `helicsFederateGetEndpointByIndex`, `helicsFederateGetEndpointByIndex`). There is no guarantee that the order the interfaces are added to the federate will be the same from computer to computer. To continue using this API

Examples

Three examples have been created to demonstrate the use of the "helics_connector" federate: one uses a direct match-file, one uses a regex match-file, and the last uses a direct match-file while using the query-and-command method of interface creation. Links to the examples are provided below; all produce identical results.

An example utilizing both kinds of iteration is available *here*.

Core Types

HELICS leverages a number of underlying communication technologies to achieve the coordination between the co-simulation components. Some of these technologies are designed to be general in nature (such as ZeroMQ) and others are designed for particular situations (such as MPI in HPC contexts).

There are several different core/broker types available in HELICS, each providing advantages in particular circumstances depending on the architecture of the federation and the underlying computation and network environment on which it is executing.

Generally speaking, the performance of the various cores is as follows (from best to worst)

1. MPI
2. IPC
3. UDP
4. TCP
5. ZMQ

Test

The Test core functions in a single process, and works through inter-thread communications. It's primary purpose is to test communication patterns and algorithms. However, in situations where all federates can be run in a single process it is probably the fastest and easiest to setup.

Interprocess (IPC)

The Interprocess core leverages Boost's interprocess communication (a part of the HELICS library) and uses memory-mapped files to transfer data rather than the network stack; in some circumstances it can be faster than the other cores. It can only be used inside a single, shared-memory compute environment (generally a single compute node). It also has some limitations on message sizes. It does not support multi-tiered brokers.

ZMQ

The ZMQ is the default core type and provides effective and robust communication for federations spread across multiple compute nodes. It uses the ZMQ mechanisms. Internally, it makes use of the REQ/REP mechanics for priority communications (such as *queries*) and PUSH/PULL for non-priority communication messages.

ZMQ_SS

The ZMQ_SS core also uses ZMQ for the underlying messaging technology but was developed to minimize the number of sockets in use, supporting very high federate counts on a single machine. It uses the DEALER/ROUTER mechanics instead of PUSH/PULL

UDP

UDP cores sends IP messages and caries with it the traditional limitation of UDP messaging: no guaranteed delivery or order of received messages. It may be faster in cases with highly reliable networking. It's primary use is for performance testing and the UDP core uses asio for networking.

TCP

TCP communications is an alternative to ZMQ on platforms where ZMQ is not available. Since the ZMQ messaging bus is built on-top of TCP it is expected that TCP provides higher performance than ZMQ. Performance comparisons have not been done, so it is unclear as to the relative performance differences between TCP, UDP, and ZMQ. It uses the asio library for networking

TCP_SS

The TCP_SS core uses TCP as the underlying messaging technology and is targeted at at networking environments where it is convenient or required that outgoing connections be made from the cores or brokers but have only a single external socket exposed.

MPI

MPI communications is often used in HPC systems. It uses the message passing interface to communicate between nodes in an HPC system. It is still in testing and over time there is expected to be a few different levels of the MPI core used in different platforms depending on MPI versions available and federation needs.

Example

Generally, all federates in a federation utilize the same core type. There could be reasons for this not to be the case, though. For example, part of the federation could be running on MPI in an HPC environment while the rest is running on one or more compute nodes outside that environment. To allow the federates in the HPC environment to take advantage of the high-speed MPI bus but to allow the rest of the federation without access to MPI to use ZMQ, a "multi-broker" or "multi-protocol broker" must be set up.

A full co-simulation example showing how to implement a multi-core federation *is written up here* (and the source code can be found HELICS Examples repository).

Dynamic Federations

In general, a dynamic federation is one in which one or more federates joins the co-simulation after the co-simulation begins. Instead, some federates join the running co-simulation part-way through its execution. For example, a co-simulation with EV federates being charged by a power system federate may have EV federates joining the co-simulation to charge and leave it when charging is done. Alternatively, real-time or hardware-in-the-loop co-simulations may have federates (components) that are only needed for part of the co-simulation and join late. Dynamic federations provide greater flexibility in constructing and running co-simulations.

Levels of Dynamic Federations

Dynamic federations can be thought of as being composed of features in increasing levels of complexity:

1. Allowing the additional of subscriptions to an existing federate publication by existing members of the federation.
2. Allowing federates that only receive information ("observers") to join the federation after execution has begun. As a part of joining the co-simulation the observer would need the functionality in level one to successfully subscribe to the necessary publications of other federates.
3. Allowing the creation of new publications, endpoints, or filters by existing federates which other members of the federation could then connect to as targets.
4. Allowing federates to join the co-simulation after execution has begun and create arbitrary interfaces (publications, subscriptions, endpoints, etc). This relies on all previous levels of complexity being implemented.
5. Allowing federate to disconnect and reconnect throughout the simulation.

HELICS v3.1 supported levels 1 and 2. HELICS v3.4 supports full dynamic federations (level 4). Level 5 is supported as of version v3.5.1.

Dynamic Subscriptions

In the normal case, the `helicsFederateRegisterInput()`, `helicsFederateRegisterSubscription()`, or `helicsFederateregisterTargetedEndpoint()` methods are called in the creation phase of co-simulation to allow for the creation of the data exchanges between federates prior to the start of co-simulation. If these calls are made after the `helicsFederateEnterInitializingMode()` call, the topology of the data exchanges between federates is altered and with it the timing dependencies. In non-dynamic federations, data published in the initialization phase is available to all subscribers of that publication as soon as as any other federate enters executing mode or makes a time request. In a dynamic federation, subscription data by default is not sent to the new (dynamically added) subscriber (or subscription) until a new publication is made after a time request or `helicsFederateEnterExecutingMode()` call has been made by the subscriber. For example, let's say Federate A publishes a value at simulation time zero and never after that point. If Federate B joins the co-simulation late and enters executing mode at simulation time five, it will not see the value published by Federate A.

In many use cases, this lack of visibility to previously published values is a problem. For example, if the published value is a voltage, it would make sense that this value would be available to all federates even if they join late; the voltage always exists and thus should conceptually always be accessible for use by the model inside the federate. To

provide persistence in this data for dynamic co-simulations, each publication can set a `bufferData` flag that retains the last value published so that any late-joining federates can access it, and this value will be sent immediately when the connection is made. This buffer is disabled by default as it does slightly increase the memory footprint of each federates using it. It is anticipated that for dynamic federations, many use cases will want to employ this flag to preserve the "always available" concept of published values. The flag can be set at the federate level so all created publications have the flag set automatically.

Dynamic Observer Federates

A federate may declare itself to be an observer in the FederateInfo structure when a federate is declared. This can be done via the command line (`--observer`) or through a HELICS federate flag as shown below. This declares that the federate will only be receiving data, not sending any so there is no time dependency of any other federate to this one.

C++:

```
FederateInfo fi;
fi.observer=true;
```

C:

```
helicsFederateInfoSetFlagOption(fi, HELICS_FLAG_OBSERVER,HELICS_TRUE,&err);
```

Python:

```
import helics as h

fi = h.helicsCreateFederateInfo()
h.helicsFederateInfoSetFlagOption(fi, h.HELICS_FLAG_OBSERVER, True)
```

The observer flag triggers functionality in the corresponding HELICS core and federate to notify the broker that it can be dynamically added. If this flag is not set an error message will be produced indicating that the federation is not accepting new federates. If a HELICS core is created before the new federate it must also be created with the observer flag enabled.

Once joined, subscriptions can be added with the timing limitations as described in the previous section. Be aware that dynamic federates, after calling `helicsFederateEnterExecutingMode()`, the time returned is not necessarily zero, but will depend on the time of federates containing the publications or endpoints that are being linked. The late-joining observer federate can get current simulation time by calling `helicFederateGetCurrentTime()`, as necessary.

Observer federates are useful for co-simulation debugging and monitoring purposes. Using an observer, it is possible to join the federation, get the latest data and make some queries about the current state of the co-simulation and then disconnect. At present, a new name is required each time an observer connects.

Dynamic Publications

Dynamic publications and endpoints are implemented as of HELICS v3.4. For any late-joining federate, the process by which the federate joins can introduce delays in subscribers seeing the values that are published. Just as in a static co-simulation, after calling `helicsFederateEnterExecutingMode()` a federate can publish values and these values will be available to any subscribing federates the next time they make a time request. This time request is also the point at which a federate could add a subscription to the late-joining federate's publication; that is, the simulation time at which the publications of the late-joining federate become visible to the rest of the federation. No values published by the late-joining federate will be visible to the federate adding this new subscription until a subsequent time request is granted.

Full Dynamic Federations

Given the above limitations, as of HELICS v3.4 fully dynamic federations are supported. By setting the `--dynamic` flag on the root broker of a federation and any intermediate brokers or cores to which dynamic federates may be added, federates may join the federation late. (HELICS have always been able to leave a federation early.) And as with observer federates, after calling `helicsFederateEnterExecutingMode()`

Reentrant federates

As of v3.5.1 reentrant federates are allowed. They must be specified with the reentrant flag (`--reentrant`) and be used in a dynamic federation. This allows a federate to disconnect, then rejoin with the same name at a later time in the co-simulation. The second and later joins are like a dynamic federate in terms of timing. Interfaces connecting to a reentrant federate may use the HELICS_OPTION_FLAG_RECONNECTABLE to allow for dynamic/automatic reconnections when an interface with the same name is created from a reentrant federate. This may be done on the same core or a new/different core. The new (reentrant) federate does not inherit any properties (other than the name) from the previous federate with the same name.

Example

An example of dynamic federation operation is available in the *Advanced Examples*, though HELICS makes it very easy to support a dynamic federation. Simply add `--dynamic` to the broker initialization string for the root broker (if you are employing a *broker hierarchy*). For example, in a federation with four federates (one of which will be joining late), the call to start the broker is

```
$ helics_broker -f3 --dynamic
```

In this example, three federates must be in the federation in order to enter initialization and then executing mode and the broker is ready if more join later.

Encrypted Communication

> **Warning** This guide is only meant to show how to use the encryption features in HELICS, and is not a substitute for getting help from a security expert to make sure the way you use the encryption features (generating keys, signing certificates, etc) is actually secure. In particular, any self-signed certificates or private key files you find online or in the HELICS repository **are for testing purposes only** and should not be used if secure communication is required.

Some applications may exchange data between federates that requires encryption in order to ensure its confidentiality. This could be of particular concern when the federates are communicating over the public internet between different institutions, and restrictions on a data set or simulator prohibit the co-simulation from being executed within a private network at a single institution.

The HELICS TCP and TCPSS cores support encryption using the OpenSSL library. By default peer verification is used which requires both the HELICS broker and federates using encrypted communication to be set up with their own certificate and private key, which also allows the encrypted communication feature to act as a form of authentication between a broker and the federates connecting to it.

This guide will provide information on building a copy of HELICS with encryption support, and show how to run a simulation with encrypted communication. It will not go into best practices for generating and distributing certificates or private keys. It is your responsibility to ensure that private keys and certificates are managed in a way that does not compromise the security of the encrypted communication. A good starting point for learning more about this topic is reading up on Public Key Infrastructure (PKI) and SSL/TLS fundamentals.

Building HELICS with Encryption Support

To build HELICS with encryption support, you must have a copy of OpenSSL installed on your system. HELICS has been tested using OpenSSL 1.1; OpenSSL 3 was recently released and includes breaking changes to the library API, though based on initial testing it seems to work with HELICS.

With most Linux package managers you will want to install the `libssl-dev` and `openssl` packages; the former provides the shared libraries and header files needed (including `libcrypto` and `libssl`), and the latter installs the `openssl` command which provides tools for managing SSL certificates and private keys.

On macOS the system copy of libssl is likely pretty old, so Homebrew should be used to install the `openssl@1.1` package, which includes a much more recent copy of libssl that includes fixes for a number of vulnerabilities.

For Windows users, a precompiled OpenSSL installer can be downloaded from https://slproweb.com/products/Win32OpenSSL.html. The full version of the installer is needed (**NOT THE LIGHT VERSION**) which includes the required OpenSSL header files, and the Win64 version is recommended for almost all users.

After that, continue as usual for *building HELICS from source* but at the CMake configure step turn on the `HELICS_ENABLE_ENCRYPTION` option. In most cases CMake will find the installed copy of OpenSSL without errors, but if not CMake will give an error message saying what options needs to be set to manually specify where OpenSSL was installed.

Enabling Encryption

After successfully building and installing a copy of HELICS with encryption support, encryption can be enabled for the HELICS broker and federates either via command line arguments, environment variables, federate configuration files, or modifying the code for custom federates. Depending on your set up for launching a co-simulation, a combination of these methods may be used.

Command-line

Encryption can be enabled by providing the `--encrypted` command-line flag to the `helics_broker` binary, or a HELICS federate that accepts command line arguments (such as one of the `helics_app` subcommands). When encryption is disabled (the default if no `--encrypted` flag or other mechanism is used to enable it), any other encryption related settings or arguments will be ignored.

Along with enabling encryption, an encryption config file must be provided with information on private key and certificate files. The `--encryption_config` command line argument can be used to specify the name and location of the encryption configuration file that should be used.

As an example, here is what the command line arguments look when starting a HELICS broker and HELICS echo app (assuming the encryption config file is named `openssl.json` and located in the current working directory):

```
helics_broker --encrypted --encryption_config=openssl.json
helics_app echo --encrypted --encryption_config=openssl.json
```

Environment Variables

As an alternative to providing command-line arguments for each broker and federate launched, the `HELICS_ENCRYPTION` and `HELICS_ENCRYPTION_CONFIG` environment variables can be set as alternatives to either or both of the `--encrypted` and `--encryption_config` arguments. This can make it easier to change the setting for many federates all at the same time, and require less typing when manually launching a co-simulation.

To set these environment variables, the following commands can be used on most Linux/macOS shells (on Windows the command will depend on if you are using Command Prompt, Powershell, or Windows Subsystem for Linux):

```
export HELICS_ENCRYPTION=true
export HELICS_ENCRYPTION_CONFIG="$HOME/example-directory/openssl.json"
```

After that, a `helics_broker` or federate can be run as usual without needing to explicitly provide `--encrypted` or `--encryption_config` arguments on the command line. As an additional note, using an absolute path to the encryption config file will help reduce the chance of errors that might happen if different federates can run from different directories.

Federate Configuration Files

The `encrypted` and `encryption_config` options can also be set from a federate configuration file, such as this:

```
{
  "encrypted": true,
  "encryption_config": "/home/username/example-directory/openssl.json"
}
```

Federate Code

Finally, it is also possible to set these options in the code of a federate (example shown in C++, but similar mechanisms can also be used from other language interfaces).

Using a `FederateInfo` object named `fi`, this might look like:

```
fi.encrypted = true;
fi.encryptedConfig = "/home/username/example-directory/openssl.json";
```

Or, the federate info args string can provide the settings in the form of a string matching the command-line arguments described earlier:

```
const std::string fedArgs =
    "--core_type=tcp --encrypted --encryption_config=" + std::string(TEST_BIN_DIR) +
    "encryption_config/openssl.json";
helics::FederateInfo fi_enc(fedArgs);
helics::ValueFederate fed1("fed1", fi_enc);
```

To check if the HELICS library being linked to has encryption support, the `helics-config.h` header can be imported, which will define `HELICS_ENABLE_ENCRYPTION` if encryption is supported:

```
#include "helics/helics-config.h"

#ifdef HELICS_ENABLE_ENCRYPTION
// <code that requires HELICS compiled with encryption support>
#endif
```

If CMake is being used, the following snippet (taken from the main HELICS repository `tests/helics/CMakeLists.txt` file around lines 37-46) shows how CMake's `configure_file` function can be used to automatically fill in the location of certificate and private key files (WARNING: do NOT put private key files in a public repository; ensuring you are using the encryption features securely is your own responsibility):

```
if(HELICS_ENABLE_ENCRYPTION)
    configure_file(
        "test_files/encryption_config/openssl.json.in" "test_files/encryption_config/
↪openssl.json"
```

(continues on next page)

(continued from previous page)

```
    )
    configure_file(
        "test_files/encryption_config/multiBroker_encrypted_bridge.json.in"
        "test_files/encryption_config/multiBroker_encrypted_bridge.json"
    )
endif()
```

And the corresponding input `openssl.json.in` file looks like:

```
{
  "encrypted": true,
  "verify_file": "${CMAKE_CURRENT_SOURCE_DIR}/test_files/encryption_config/openssl_certs/
↪ca.pem",
  "certificate_chain_file": "${CMAKE_CURRENT_SOURCE_DIR}/test_files/encryption_config/
↪openssl_certs/server.pem",
  "private_key_file": "${CMAKE_CURRENT_SOURCE_DIR}/test_files/encryption_config/openssl_
↪certs/server.pem",
  "tmp_dh_file": "${CMAKE_CURRENT_SOURCE_DIR}/test_files/encryption_config/openssl_certs/
↪dh4096.pem",
  "password": "test"
}
```

Another big point of CAUTION: do NOT use the example/test SSL certificates or private key files provided in the HELICS repository! These files are **for testing only** with data that does not require protection. By using them with a real use case, you might as well not be using any encryption at all.

Configuring Encryption Settings for a Co-simulation

Besides enabling encryption and providing an encryption configuration file using one of the previously described methods, there are a few settings that the encryption configuration file should provide.

The general outline of what the file looks is:

```
{
    "encrypted": <true|false; this setting is optional and sort of an alternative to the␣
↪--encrypted flag so can be omitted>,
    "verify_file": "<path to certificate authority (CA) file to verify connecting␣
↪federates>",
    "certificate_chain_file": "<path to certificate file to send to other federates>",
    "private_key_file": "<path to private key file matching the certificate sent to␣
↪other federates>",
    "tmp_dh_file": "<optional file with parameters to use for the key exchange>",
    "password": "<password to decrypt the private key file; default empty>"
}
```

An example of what this might look like after being filled in is:

```
{
  "encrypted": true,
  "verify_file": "/home/username/openssl_certs/ca.pem",
  "certificate_chain_file": "/home/username/openssl_certs/server.pem",
  "private_key_file": "/home/username/openssl_certs/server.pem",
  "tmp_dh_file": "/home/username/openssl_certs/dh4096.pem",
```

(continues on next page)

(continued from previous page)

```
  "password": "test"
}
```

In an actual set up, it is likely that brokers and federates would have different private key files. A later section in the documentation has a brief description of how to generate self-signed certificates. It is important to be aware that the choices you make while generating the certificates and private keys, and how you get them onto individual machines running the federates can have an impact on the security of communication for the co-simulation. This is not intended as a guide to those issues, and you should seek outside help in the form of other resources on the topic and from security experts at your organization.

Search terms to get started are: "Public Key Infrastructure (PKI)", "TLS fundamentals/basics", "secure private key management". Some of the resources out there (as of early 2023, check archive.org if the site goes offline in the future) include:

- https://github.com/ssllabs/research/wiki/SSL-and-TLS-Deployment-Best-Practices covers a good number of the issues for using encryption securely
- https://opensource.com/article/19/6/cryptography-basics-openssl-part-1 at the section title "The hidden security pieces in the client program"
- https://geekflare.com/tls-101/ provides a decent overview, though website/browser related parts aren't relevant to HELICS
- https://www.internetsociety.org/deploy360/tls/basics/ also seems to be a decent source of information

Courses provided at a local university or online can also be a good source of information on best practices, such as:

- https://www.udemy.com/course/ssl-tls-intro/
- https://immersivelabs.online/ "TLS fundamentals" series of labs

Bridge Encrypted and Unencrypted Communication

A mix of encrypted and unencrypted communication types is possible using the *HELICS Multi Broker* functionality. See the linked article for more details on how to set this up.

An example of a configuration file that creates both an encrypted tcpss core and an unencrypted tcpss core is shown below.

```
{
  "master": {
    "core_type": "test"
  },
  "comms": [
    {
      "core_type": "tcp_ss",
      "local_port": 30000,
      "encrypted": true,
      "encryption_config": "<path to encryption config JSON file>"
    },
    {
      "core_type": "tcp_ss",
      "local_port": 40000
    }
  ]
}
```

Federates using encryption can connect to the broker on port 30000, and federates that don't require encrypted communication can connect to the broker using port 40000.

Creating Self-Signed SSL/TLS Certificates

This section assumes you have the `openssl` command installed and working on a Unix based system (Linux, macOS, Windows Subsystem for Linux).

If desired, a `openssl.cnf` file can be used to set some default values and provided to the following commands using the `-config` command line argument.

The OpenSSL Essentials tutorial from Digital Ocean, which can be found at https://www.digitalocean.com/community/tutorials/openssl-essentials-working-with-ssl-certificates-private-keys-and-csrs, gives a good overview of the commands listed here and more.

Creating a Root CA

The first step to creating certificates is creating a self-signed certificate for our Root CA (Certificate Authority):

```
openssl req -new -x509 -keyout root_ca.key -out root_ca.pem
```

This command will ask you for various bits of information, and a password that will be used when creating certificates later. The output will be a `root_ca.key` file which has the private key for the self-signed Root CA, and a `root_ca.pem` with the public key certificate. It is not necessary to fill in all of the fields, some may have a default value though can also be left blank.

The `root_ca.pem` file will need to be copied to each machine as a means to verify the signed certificates presented by each of the other machines.

Creating a certificate for brokers/federates

After creating a Root CA, you can sign certificates for the other machines or brokers/federates that will be connecting to your co-simulation.

First, each machine (or federate/broker) should generate its own public/private key pair using a command such as:

```
openssl genrsa -aes128 -out machine1.pem 2048
```

This will ask for a password to encrypt the key, and both keys will be stored in the resulting machine1.key file.

After generating the key pair, a certificate signing request (CSR) needs to be generated, which the Root CA will use to create a signed certificate that brokers/federates can use to verify that the machine they are connecting to is the one that they expect. The command to create a CSR is:

```
openssl req -new -key machine1.pem -out machine1.csr
```

Using the Root CA created in the first step, a signed certificate is created that can be given to the other machines:

```
openssl ca -in machine1.csr -out machine1.crt -cert root_ca.pem -keyfile root_ca.key
```

Finally, the private key and the signed certificate file can be combined to form a single file for convenience:

```
cat machine1.crt >> machine1.pem
```

Creating a temporary Diffie-Hellman key exchange parameters file

If desired, the optional file containing temporary Diffie-Hellman key exchange parameters can be created with: `openssl dhparam -out dh4096.pem 4096`

Create HELICS encryption configuration files

After generating private keys and certificates, encryption configuration files can be created for use by brokers and federates running on machine1. The `verify_file` would be the `root_ca.pem` file. If the signed machine certificate and private key were combined into a single file, `certificate_chain_file` and `private_key_file` would both be the `machine1.pem` file from our above example. Otherwise, the certificate chain file would just be the output certificate from the signing step.

An example of what this might look like after being filled in (assuming the private key for machine1 is "test") would be:

```
{
  "verify_file": "root_ca.pem",
  "certificate_chain_file": "machine1.pem",
  "private_key_file": "machine1.pem",
  "tmp_dh_file": "dh4096.pem",
  "password": "test"
}
```

The private key generating and certificate signing steps would then repeated for each additional machine that will be joining the co-simulation.

Debugging Connection Errors

As a first step, disable encryption by removing the `--encrypted` flag when launching the broker and all federates. If establishing a connection still doesn't work, the issue is with the networking configuration such as wrong ports, IP addresses, or firewalls.

If the simulation runs successfully with encryption disabled, then the issue is likely either forgetting to enable encryption on either the broker or one of the federates, or not giving the broker and federates the right set of certificates, CA files, and private keys. Setting the logging level for the broker and all federates to a higher level such as `debug` or `trace` may be able to provide more insight into the underlying problem.

To check what certificate the broker server is sending to clients when using an encrypted tcpss core, you can use `openssl s_client -connect localhost:33133 -showcerts` if you have the `openssl` command/package installed. This may be useful for confirming that it is sending the certificate that you expect.

Environment variables

The HELICS command line processor has some ability to read and interpret command line environment variables. These can assist in setting up co-simulations. In general the configuration of HELICS comes from 3 sources during setup. After setup API's exist for changing the configuration later. The highest priority is given to command line arguments. The second priority is given to configuration files, which can be given through a command string such as `--config=configFile.ini`. The file can be a `.ini, .toml, or .json`. By default HELICS looks for a `helicsConfig.ini` file. The lowest priority option is though environment variables. Only a subset of controls work with environment variables. All environment variables used by HELICS begin with `HELICS_`.

Federate environment variables

For setting up a federate a few environment variables are used

- HELICS_LOG_LEVEL: the log level for the federate to use. Equivalent to --loglevel=X
- HELICS_CORE_INIT_STRING: the init string to pass to the core when creating it. Equivalent to --coreinit=X
- HELICS_CORE_TYPE: the type of core to use e.g. "ZMQ", "TCP", "IPC", "MPI", etc. Equivalent to specifying --coretype=X

Core and Broker environment variables

- HELICS_BROKER_LOG_LEVEL: the log level for the broker to use. Equivalent to --loglevel=X
- HELICS_BROKER_KEY: the key to use for connecting a core to a broker. See *broker key*
- HELICS_BROKER_ADDRESS: the interface address of the broker. Equivalent to --brokeraddress=X
- HELICS_BROKER_PORT: the port number of the broker. Equivalent to --brokerport=X
- HELICS_CONNECTION_PORT: the port number to use for connecting. This has different behavior for cores and brokers. For cores this is the broker port and for brokers this is the local port
- HELICS_CONNECTION_ADDRESS: the interface address to use for connecting. This has different behavior for cores and brokers. For cores this is the broker address and for brokers this is the interface.
- HELICS_LOCAL_PORT: the port number to use on the local interface for external connections. Equivalent to --localport=X
- HELICS_BROKER_INIT: the command line arguments to give to an autogenerated broker. Equivalent to --brokerinit=X
- HELICS_CORE_TYPE : the type of core to use e.g. "ZMQ", "TCP", "IPC", "MPI", etc. Equivalent to specifying --coretype=X

Networking environment variables

- HELICS_ENCRYPTION: set to 1 to activate encryption if built into HELICS for the TCP core types and ZMQ core types
- HELICS_ENCRYPTION_CONFIG: set to the path to a config file to specify the options for encryption including any necessary keys and certificate authorities

Environment variables used in the Web server

- HELICS_HTTP_ADDRESS: The address of the interface to use in the webserver
- HELICS_HTTP_PORT: The port to use on the webserver to accept connections
- HELICS_WEBSOCKET_ADDRESS: The interface address to use to accept incoming websocket connections
- HELICS_WEBSOCKET_PORT: The port number to use for accepting websocket connections

Iteration

In simulation in general, iteration at a time step is can be helpful in the case of algebraic loops, that is, when two state variables are co-dependent on each other. If the analysis requires the two sub-systems to reach a consistent state, then iteration is required. In these situation, different methods (generally Fixed-Point Methods, like Newton's method), can be used to arrive at the desired solution. When modeling such systems in a single executable a traditional solver can be used as all the system states are visible to said solver. When modeling these systems in a co-simulation environment, no single solver has the necessary visibility to force convergence between the two sets of states variables and iteration

must instead by facilitated by HELICS. Iterations at a single time step are intended to resolve this issue by allowing Federates to exchange data back and forth until convergence to a consistent state is reached.

Note

Since neither federate has a full view of the problem, convergence issues cannot be dealt with quite in the same way as general numerical solvers. In fact, convergence cannot be guaranteed, because no federate really knows the logic/trajectory of the state variables in the other federates it is iterating against.

Simulation Tool Support for Iteration

Though HELICS provides specific APIs to iterate between federates, the underlying tools may or may not support such iteration. Generally speaking, if the models inside the simulation tools are stateless (that is, only dependent on the current input to produce a new output) then iteration at a given time will be possible. As other federates iterate and produce new outputs, a stateless simulator can take those as its new inputs and likewise produce new outputs.

Many simulation tools, though, do not fit into this category. The use of previous system states to find the current state of the model is very common. For example, if the model of the system includes differential equations, at least one previous state will be required to iterate at a fixed time. Not all simulation tools appropriately store these previous states and is is the job of the person integrating the simulation tool to determine if iteration is supported by the tool in question and use the HELICS APIs appropriately.

HELICS APIs

There are two possible co-simulation states where iterations are possible in HELICS:

State	Iteration call
`INITIALIZATION`	`helicsFederateEnterExecutingModeIterative()`
`EXECUTION`	`helicsFederateRequestTimeIterative()`

Both calls take an `iteration_request` as an input and return an `iteration_result` as an output. The only difference between the two, is that `helicsFederateRequestTimeIterative()` additionally takes requested time as an argument and additionally returns granted time as an output alongside `iteration_result`. If iterating in execution mode, it is necessary that the requested time be the next time appropriate for the federate (as if you were not iterateing). Requesting the same simulated time a federate is at AND making it an iterative time request is a recipe for HELICS disaster; don't do it.

`iteration_request`

When using the iteration APIs, the `iteration_request` parameter of the call indicates to HELICS what what the federate's ability and desire is when iterating.

- `NO_ITERATION`: forces a result of `NEXT_STEP` and is equivalent to not using the iterative APIs at all. Using this value for `iteration_request`, `NO_ITERATION` effectively opts the given federate out of the iteration at the given timestep, allowing other federates to continue iterating if they so desire. This can be useful if a given federate only needs to be involved in iteration under specific conditions.
- `FORCE_ITERATION`: forces `iteration_result` for all iterating federates to be `ITERATING`. This should only be used in rare cases.
- `ITERATE_IF_NEEDED`: is the normal request when iteration is desired. This could have been called `ITERATE_IF_NEW_INPUTS`; if new data is available to the federate making this request, `iteration_result` will be `ITERATING`. When no new data is available it returns `NEXT_STEP` and in the case of Execution mode, a

new granted time is also returned. Return values of NEXT_STEP will always be accompanied with granted times that move the simulation forward.

iteration_result

There are two relevant iteration results from the calls mentioned above:

- ITERATING: means that *new data* is available to the federate and it should could iterating at the current simulated time. When the co-simulation is in Execution mode, this result is equivalent to receiving the *same* granted time as the previous call.
- NEXT_STEP: means that the federation is ready to proceed. During initialization, this means the federation can enter Execution mode, while during Execution mode this means that the federation can proceed to the next time instant.

Convergence Criterion Definition

A core tenent of HELICS co-simulation philosophy is local autonomy with distributed control. That is, no federate can force another federate to do something. For co-simulation to work at all, the federates have to choose to cooperate (obviously) but each federate is responsible only for itself and does not rely on external federates to manage its own operation. When it comes to iteration, this can get a bit tricky.

Traditional integrated simulation tools with iterative solvers generally have access to all the information necessary and can use a variety of convergence criterion when deciding when to iterate and when to move on. In a co-simulation environment, there is no central authority for convergence and it is the obligation of each federate to appropriately implement its own convergence criterion. It is also entirely possible that these criterion are different across federates and, if that is the case, there is no guarantee that any given federate's convergence criterion will be met. (Generally, only the loosest criterion can be guaranteed to be satisfied.)

Because federates don't generally expose all of their internal states (just those that other federates need for a particular analysis or use case), it is highly recommended that each federate base its own convergence criterion on received changes in inputs. That is, each federate should be evaluating how much its subscribed values are changing from iteration to iteration and when those inputs have settled down and are changing sufficiently little from iteration to iteration, the federate should consider itself converged. At this point, it has been demonstrated that changes in the federates outputs are having negligible impact on the rest of the federation and it is likely the federation as a whole has convereged.

Guiding Principles

Publish Before Iterative Time Requests

Iteration in HELICS is driven by the presence of new data produced by other members of federation. If a federate does not have new input data from the federations then HELICS assumes there is no need to iterate and will return NEXT_STEP from the iterating call. Because of this, a critical aspect to iteration in HELICS is: ***publish before making the iterating call***. This will ensure all other federates relying on these inputs will have them and know they need to iterate. Producing new outputs effectively forces all federates that have requested ITERATE_IF_NEEDED to iterate once more.

Indicate Convergence By Not Publishing Before Iterative Time Requests

The big and important exception to this is if a given federate has reached a state where it considers itself converged. At this point, the only thing the federate needs to do to indicate this to the rest of the federation is not publish any new values before making the exact same iterative time request. That iterative time request should still use ITERATE_IF_NEEDED and if it is returned ITERATING it should look for new inputs. If the convergence criterion is met, it once again would not publish anything and make the same request again. If the convergence criterion is no longer met it should recalculate the model state and publish new outputs.

only_update_on_change

Since iteration is highly dependent on received inputs, an implicit convergence criterion is implemented through the *only_update_on_change* flag. When set this flag will only show new inputs to federates if they have changed since the last time they have been published. This can be helpful in that if there is a slightly mis-behaving federate that published every time, regardless of whether it thinks it has converged, those republications can be screened out. Additionally, only_update_on_change has a related parameter called *tolerance* that allows values within a certain numerical range from the previously published value to be considered "unchanged" and will not be presented to the federate. This can effectively be used to define the convergence criterion as it limits the changes in inputs to a federate.

Example Iterative Federate Psuedo-code

A rough outline for how iteration can be implemented using the python API is:

```
# Time simulation loop
while grantedtime < maxtime:
    # initial publication to make sure there is new data
    h.helicsPublicationPublishDouble(pubid, pubval)

    # update requested time
    requestedtime = grantedtime + deltatime

    converged = false

    # Iteration loop
    while not converged:
        grantedtime, iterative_return = h.helicsFederateRequestTimeIterative(
            fed, requestedtime, h.helics_iteration_request_iterate_if_needed
        )
        # If the iterative call is telling us to move on, we can break out
        #  of the iterative while loop and move on in time
        if iterative_return == h.helics_iteration_result_next_step:
            break

        # Save old input states and get new one
        s_old = s
        s = h.helicsInputGetDouble(subid)

        # check convergence (e.g. any significant change on the inputs)
        converged = check_convergence(s, s_old)

        if not converged():
            # perform internal update
            pubval = state_update(s)

            # publish results
            h.helicsPublicationPublishDouble(pubid, pubval)
```

Example

An example utilizing both kinds of iteration is available in *here*

Multi-Protocol Broker

Starting in HELICS 2.5 there is a Multi Broker type that allows connection with multiple communication types simultaneously. The multibroker allows an unlimited number of communication operations to interact.

Starting a multiBroker

A multibroker can be started as BrokerApp or a helics_broker. For the HELICS Broker the configuration must given as a file since each of the core types linked must be configured independently. Using the helics_broker the startup commands would look something like

```
helics_broker --type multi --config=helics_mb_config.json --name=broker1
```

A couple example configurations follow.

```
{
  "master": {
    "type": "test"
  },
  "comms": [
    {
      "type": "zmq",
      "interfaceport": 23410
    },
    {
      "type": "zmq",
      "interfaceport": 23700
    }
  ]
}
```

The primary communication pathway can be specified in a master object or on the root of the configuration file.

```
{
  "type": "test",
  "comms": [
    {
      "type": "zmq"
    },
    {
      "type": "tcp"
    }
  ]
}
```

Master comm information can also be given through the command line. The master comm is the only on in which higher level broker information may be specified. Any broker related specification in the comms sections will result in an error. If the MultiBroker is intended to be a root broker then no master section is required. Multiple network communication pathways of the same type are allowed assuming they use different ports.

Programmatically multibrokers can also be started using the BrokerApp and giving it the type `helics::core_type::MULTI` for arguments to the multibroker the type of the master comm can be specified on the command line arguments as well.

Limitations

- Using TCPSS comms in the multibroker does not currently support outgoing connections like a full TCPSS broker would. This will likely be fixed in upcoming releases.
- Configuration files must currently be in JSON, in a few limited cases TOML files may work, but configuration of multiple comms in a toml file will not work. This will also likely be fixed in upcoming releases.
- General support for multibrokers is not provided in the webServer due to limitations on the configuration files. Some mechanism for this will be allowed in a future release.

Example

An example implementation of a multi-protocol broker with explanation *can be found here* with the source code over here in the repository.

Multi-Computer Co-simulations

Though often it may make sense to put a HELICS broker on every compute node used in the co-simulation (as shown in the *Broker Hierarchy example*), particularly for small co-simulations that for various reasons may not fit on a single computer, it may only be necessary to use a single broker on one computer and point the federates on the other computer(s) towards it.

Broker Configuration

Generally, there are a few changes that will be necessary for running a multi-computer co-simulation:

- Adding the `--ipv4` flag to the broker initialization string. This opens an external port for federates on other computers to use when connecting to the broker; something like `helics_broker -f 3 --loglevel=warning --ipv4` (NOTE: –ipv4 is a shortcut to open up the ports on all external network interfaces with ipv4 addresses, other options can do similar things on specific interfaces or with ipv6).
- Define `broker_address` for each of the federates that are running on another computer. This will look something like `"broker_address": "tcp://10.211.55.23"` if using a JSON configuration file.
- The default port for the HELICS broker is 23405 and if that works in your networking environment then you don't need to do anything. To use another port, each federate must change the value for `broker_port` (*e.g.* `"broker_port": "23500"`)and the broker must set its own `local_port` option to the same value (*e.g.* `helics_broker -f 3 --loglevel=warning --ipv4 --port=23500`).

Which is not to say there can't be other networking complications. Once running on multiple computers the network configuration and configuration can create new challenges. Handling these is beyond the scope of this document but take a look at the some of the other examples to get an idea of how you might be able to handle this. There's also the *Configuration Options Reference* that has a more comprehensive list of the *network configurations available*.

For those that are doing configuration via APIs, the "broker_address" and "broker_port" options can be included as part of the *"core_init_string"* for the federates and the *"broker_init_string"* if instantiating the broker via APIs.

Example

A full co-simulation example showing how to implement a multi-computer co-simulation *has been documented over here* (and the source code is in the HELICS Examples repository). Note that this example will require adjusting the `broker_address` option in the Charger and Controller federate and, obviously, will require more than one computer (or virtual machine) to run.

Multi-Source Inputs

On occasion it is useful to allow multiple source to feed an input, creating an *n*-to-one relationship (publications to input). This could occur in situations like a summing junction, or a switch that can be turned on or off from multiple other federates. Alternatively, a multi-source input can be a convenient way to collecting multiple inputs into a vector for processing by the federate. While technically supported prior to 2.5.1 the control and access to this feature of HELICS was not well thought through or straightforward. The developments in 2.5.1 made it possible to specify a mathematical reduce operation on multiple inputs to allow access to them as a single value or vector.

Mechanics of multi-input handling

Internally, HELICS manages input data in a queue and when a federate is granted a time, the values are read and placed in a holding location by source. In many cases there is likely only to be a single source. But if multiple publications link to a single source the results are placed in a vector. The order of the values in that vector is determined by the order of linking when the federate with the multi-source input is created. If a single publication value is retrieved from the input, the newest value is given as if it were a single source. In case of ties (multiple publishing federates publishing values on the same timestep), the publication that connected first is given priority.

Controlling the behavior

A few flags are available to control or modify this behavior including limiting the number of connections and adjusting the priority of the different inputs sources. The behavior of inputs is controlled via flags using `helicsInputSetOption()` method.

The number of connections

There are several flags and handle options which can control this for Inputs

- `helics_handle_option_single_connection_only` : If set to true specified that an input may have only 1 connection
- `helics_handle_option_multiple_connections_allowed`: if set to true then multiple connections are allowed
- `helics_handle_option_connections`: takes an integer number of connections specifying that an input must have N number of connections or an error will be generated.

Controlling priority

The default priority of the inputs if they are published at the same time and only a single value is retrieved is in order of connection. This may not be desired so a few handle options are available to manipulate it.

- `helics_handle_option_input_priority_location` takes a value specifying the input source index to which it will give priority. If given multiple times it establishes an ordering of the input priority with the most recent call being highest priority. This allows signals that are received at the same time to be prioritized. For example, let's say the option is called first with a given value of "2" then again with a value of "1". If all signals are sent at the same simulated time, the source with index 1 will have highest priority, and in the case of a tie between sources 0 and 2, source 2 will have priority.
- `helics_handle_option_clear_priority_list` will erase the existing priority list.

Reduction operations on multiple inputs

The priority of the inputs is only applicable if the default operation to retrieve a single value is used. The option `helics_handle_option_multi_input_handling_method` can be used to specify a reduction operation on all the inputs to process them in some fashion a number of operations are available.

method	Description
helics_multi_input_no_op	default operation to pick the highest priority value
helics_multi_input_vecto	take all the values and collapse to a single vector, converts strings into a JSON string vector
helics_multi_input_and_o	treat all inputs as booleans and perform an *and* operation
helics_multi_input_or_op	treat all inputs as boolean and perform an *or* operation
helics_multi_input_sum_o	sum all the inputs after converting to double vector, except if input type is string then concatenate all inputs as a single string
helics_multi_input_diff_	if the input type is specified as a double subtract the sum of remaining values from the first, if it is a vector do a vector diff operation
helics_multi_input_max_o	pick the biggest value
helics_multi_input_min_o	pick the smallest value
helics_multi_input_avera	take a numerical average of all values

The handling method specifies how the reduction operation occurs the value can then retrieved normally via any of the `getValue` methods on an input.

Configuration

Multi Input handling can be configured through the programming API or through a file based configuration.

C++

```
auto& in1 = vFed1->registerInput<double>("");
    in1.addTarget("pub1");
    in1.addTarget("pub2");
    in1.addTarget("pub3");
    in1.setOption(helics::defs::options::multi_input_handling_method,
                  helics::multi_input_handling_method::average);
```

C

```
/*errors are ignored here*/
helics_input in1 = helicsFederateRegisterInput("",helics_data_type_double,"",nullptr);
helicsInputAddTarget(in1,"pub1",nullptr);
helicsInputAddTarget(in1,"pub2",nullptr);
helicsInputAddTarget(in1,"pub2",nullptr);
helicsInputSetOption(in1,helics_handle_option_multi_input_handling_method,helics_multi_
↪input_average_operation, nullptr);
```

Python

```
in1 = h.helicsFederateRegisterInput("", h.helics_data_type_double, "")
h.helicsInputAddTarget(in1, "pub1")
h.helicsInputAddTarget(in1, "pub2")
h.helicsInputAddTarget(in1, "pub2")
h.helicsInputSetOption(
    in1,
    helics_handle_option_multi_input_handling_method,
    helics_multi_input_average_operation,
)
```

The handling can also be configured in the configuration file for the federate

TOML

```
inputs=[
{key="ipt2",  type="double", targets=["pub1","pub2"], connections=2, multi_input_
↪handling_method="average"}
]
```

JSON

```
"inputs": [
    {
      "key": "ipt2",
      "type": "double",
      "connections":2,
      "multi_input_handling_method":"average",
      "targets": ["pub1","pub2"]
    }
 ]
```

The priority of the inputs in most cases determined by the order of adding the publications as a target. This is not strictly guaranteed to occur but is a general rule and only applies in the default case, and possibly the diff operation.

Example

An *explanation of a full co-simulation example* showing how a multi-source input might be used in a federation is provided in the HELICS Examples repository.

Configuration in Complex Networks

Default Operation using ZMQ core

The starting place for many HELICS-based co-simulation is running on a single computer using the default core, ZMQ. The ZMQ core provides a lot of advantages but its behavior on the localhost network is more complex and this complexity is often hidden when a user is only running the co-simulation on a single computer.

When starting up a HELICS-based co-simulation using the ZMQ core, HELICS opens two ports: one for high-priority traffic (default to port number 23405) and the other for low priority traffic (defaults to 23406). As federates join the co-simulation they connect to the broker on the high-priority port and the broker assigns the federate a dedicated port that all further communication between the broker and federate takes place.

Again, when all federates are running on a single computer, this prolific use of port numbers (at high federate counts) is generally not a problem. There may need to be permission granted on the local computer to open those ports in a firewall for localhost traffic but no traffic ever leaves the local machine.

Speciality Cores for Complex Networks

When a co-simulation grows to the point where it starts spanning multiple compute nodes, the default ZMQ core may start running into networking problems. In situations where all the compute nodes are still in the same subnet or administered by the same organization, IT policies may easily accommodate the ZMQ core's need for ports. This may not always be the case, though, and to allow HELICS to operate in these environments, the HELICS developers have created two speciality cores to simplify the impact of HELICS in a networked environment.

`zmq_ss` core

The zmq_ss core is a version of the ZMQ core with modified behavior to only use a single socket. (A socket is the combination of ip address and port.) The core has been designed to accommodate a large number of federates where there is a possibility of running out of available ports on a single compute node (ip address). By using a single socket, it has the side-affect of also simplifying the required networking and/or firewall configuration.

`tcp_ss` core

The tcp_ss core is similar in nature to the zmq_ss core in that it uses a single socket but is based on the tcp core. This core removes the extra complexity of the zmq core and just uses the tcp protocol directly and has been designed as the go-to core when needing to work in complex networking environments. In addition to only using a single socket, the tcp core allows the broker to initiate connections with federates which can be important when trying to work in networking environments when firewalls prevent connections to be initiated in particular directions.

`broker_address`, `broker_port`, `local_interface` and `local_port`

Regardless of which core you're using, there are a few specific networking options that allow for changes to default values to enable working in a more restrictive networking environment.

- **`broker_address`** and **`broker_port`** - for sub-brokers or federates, defines the IP address (`broker_address`) and port (`broker_port`) which should be used to connect to a parent broker
- **`local_interface`** and **`local_port`** - defines the IP address (`local_interface`) and port (`local_port`) where a broker, sub-broker, or federate will look for connections to the federation.

The `broker_port` is typically defined as a command line switch when instantiating the broker. `broker_address` and `broker_port` are also used as options when configuring federates to define the socket they should connect to. These options can be set as part of the `fed_init_string`, `core_init_string` or part of the JSON configuration for that federate. The `local_port` options can be similarly configured. Further details on these configuration methods can be found in the *Configuration Options Reference*.

It is also possible to include the port number when defining `broker_address` (effectively defining the broker's socket) in addition to the IP address such as in the format: `192.169.0.1:23400`. Doing so would then not require the `broker_port` option to be defined.

As of this writing there is a generic `port` option supported in HELICS that tries to be all things to all people. Experience has shown that though well-intentioned, the feature of it being generic has become a bug in that it causes confusion among users. You may still see it lurking in examples or documentation but it is recommended that its use be avoided and the more explicit `broker_port` and `local_port` be used instead.

Broker Hierarchies and Multi-Computer Federations

Once a federation reaches a certain size, it is not unusual for it to end up deployed across multiple compute nodes and often this results in establishing a broker hierarchy to reduce traffic between compute nodes (and help the co-simulation to run faster). In complex networking environments, this will likely entail the use of the `tcp_ss` core and the specification of the broker and sub-broker sockets.

The good news is that we already have two other pages of documentation devoted to this and both include running examples that show how these features can be put to use. Here's the *documentation on broker hierarchies* and here's the one on *running across multiple compute nodes*.

Orchestration for HPC systems

The goal of this guide is to show and guide you on how to handle co-simulation orchestration on high-performance computing systems. We will walk through using a specific tool (Merlin) that has been tested with HELICS co-simulations. This is not the only tool that exists that has this capability and is not a requirement for co-simulation orchestration. One advantage that Merlin has is its ability to interface with HPC systems that have SLURM or Flux as their resource managers.

Definition of "Orchestration" in HELICS

We will define the term "orchestration" within HELICS as workflow and deployment in an HPC environment. This will allow users to define a co-simulation workflow they would like to execute and deploy the co-simulation either on their own machine or in an HPC environment.

Orchestration with Merlin

First you will need to build and install Merlin. This guide will walk through a suggested co-simulation spec using Merlin to launch a HELICS co-simulation. This is not a comprehensive guide on how to use Merlin, but a guide to use Merlin for HELICS co-simulation orchestration. For a full guide on how to use Merlin, please refer to the Merlin tutorial.

Merlin is a distributed task queuing system, designed to allow complex HPC workflows to scale to large numbers of simulations. It is designed to make building, running, and processing large scale HPC workflows manageable. It is not limited to HPC; it can also be set up on a single machine.

Merlin translates a command-line focused workflow into discrete tasks that it will queue up and launch. This workflow is called a specification, spec for short. The spec is separated into multiple sections that is used to describe how to execute the workflow. This workflow is then represented as a directed acyclic graph (DAG) which describes how the workflow executes.

Once the Merlin spec has been created, the main execution logic is contained in the Study step. This step describes how the applications or scripts need to be executed in the command line in order to execute your workflow. The study step is made up of multiple run steps that are represented as the nodes in the DAG.

For a more in-depth explanation on how Merlin works, take a look at their documentation here

Why Merlin

The biggest feature that Merlin will give HELICS users is its ability to deploy co-simulations in an HPC environment. Merlin has the ability to interface with both FLUX and SLURM workload managers that are installed on HPC machines. Merlin will handle the request for resource allocation, and take care of job and task distribution amongst the nodes. Users will not need to know how to use SLURM or FLUX because Merlin will handle all resource allocation calls to the workload manager, the user will only need to provide the number of nodes they need for their study.

Another benefit of using Merlin for HELICS co-simulation is its flexibility to manage complex co-simulations. `pyhelics` includes functionality to launch HELICS co-simulations using a JSON file that defines the federates of the co-simulation. As of this writing it does not have the ability to analyze the data and launch subsequent co-simulations. In this type of scenario a user could use Merlin to setup a specification that included an analyze step in the Study step of Merlin. The analysis step would determine if another co-simulation was needed and the input to the next co-simulation, and would then proceed to launch the co-simulation with the input generated by the analysis step.

Merlin Specification

A Merlin specification has multiple parts that control how a co-simulation may run. Below we describe how each part can be used in a HELICS co-simulation workflow. For the sake of simplicity we are using the pi-exchange python example that can be found here. The goal will be to have Merlin launch multiple pi-senders and pi-receivers.

Merlin workflow description and environment

Merlin has a description and an environment block. The `description` block provides the name and a short description of the study.

```
description:
  name: Test helics
  description: Juggle helics data
```

The env block describes the environment that the study will execute in. This is a place where you can set environment variables to control the number of federates you may need in your co-simulation. In this example, `N_SAMPLES` will be used to describe how many pi-senders and pi-receivers (total federates) we want in our co-simulation.

```
env:
  variables:
    OUTPUT_PATH: ./helics_juggle_output
    N_SAMPLES: 8
```

Merlin Step

The Merlin step is the input data generation step. This step describes how to create the initial inputs for the co-simulation so that subsequent steps can use this input to start the co-simulation. Below is how we might describe the Merlin step for our pi-exchange study.

```
merlin:
  samples:
    generate:
      cmd: |
        python3 $(SPECROOT)/make_samples.py $(N_SAMPLES) $(MERLIN_INFO)
        cp $(SPECROOT)/pireceiver.py $(MERLIN_INFO)
        cp $(SPECROOT)/pisender.py $(MERLIN_INFO)
    file: samples.csv
    column_labels: [FED]

NOTE: samples.csv is generated by make_samples.py. Each line in
samples.csv is a name of one of the json files that is created.
```

There is a python script called `make_samples.py` located in the HELICS repository that generates all helics-cli json configs that will be executed by helics-cli that will be used to execute the co-simulations. `N_SAMPLES` is an environment variable that is set to 8, so in this example 8 pireceivers and 8 pisenders will be created and used in this co-simulations. `make_samples.py` also outputs the name of each json file to a csv file called `samples.csv`. `samples.csv` contains the names of the json files that were generated. The `column_labels` tag tells Merlin to set each column in `samples.csv` to `[FED]`. This means we can use `FED` as a variable in the study step. Below is an example of one of the json files that is created.

```
{
  "federates": [
    {
      "directory": ".",
      "exec": "python3 -u pisender.py 0",
      "host": "localhost",
      "name": "pisender0"
    }
  ],
```

(continues on next page)

(continued from previous page)

```
  "name": "pisender0"
}
```

This json file will then be used as the input file for helics-cli. The helics-cli will be executed in the study step in Merlin which we will go over next.

Study Step

The study step is where Merlin will execute all the steps specified in the block. Each step is denoted by a name and has a run segment. The run segment is where you will tell Merlin what commands need to be executed.

```
- name: start_federates <-- Name of the step
  description: say Hello
  run:
    cmd: |
      helics run --path=$(FED) <-- execute the HELICS runner for each column in samples.
↪csv
      echo "DONE"
```

In the example snippet we ask Merlin to execute the json file that was created in the Merlin step. Since the `FED` variable is a list, this command will get executed for each index in `FED`.

Full Spec

Below is the full Merlin spec that was created to make 8 pi-receivers and pi-senders and execute it as a Merlin workflow.

```
description:
  name: Test helics
  description: Juggle helics data

env:
  variables:
    OUTPUT_PATH: ./helics_juggle_output
    N_SAMPLES: 8

merlin:
  samples:
    generate:
      cmd: |
        python3 $(SPECROOT)/make_samples.py $(N_SAMPLES) $(MERLIN_INFO)
        cp $(SPECROOT)/pireceiver.py $(MERLIN_INFO)
        cp $(SPECROOT)/pisender.py $(MERLIN_INFO)
    file: samples.csv
    column_labels: [FED]

study:
  - name: start_federates
    description: say Hello
    run:
      cmd: |
        spack load helics
        helics run --path=$(FED)
```

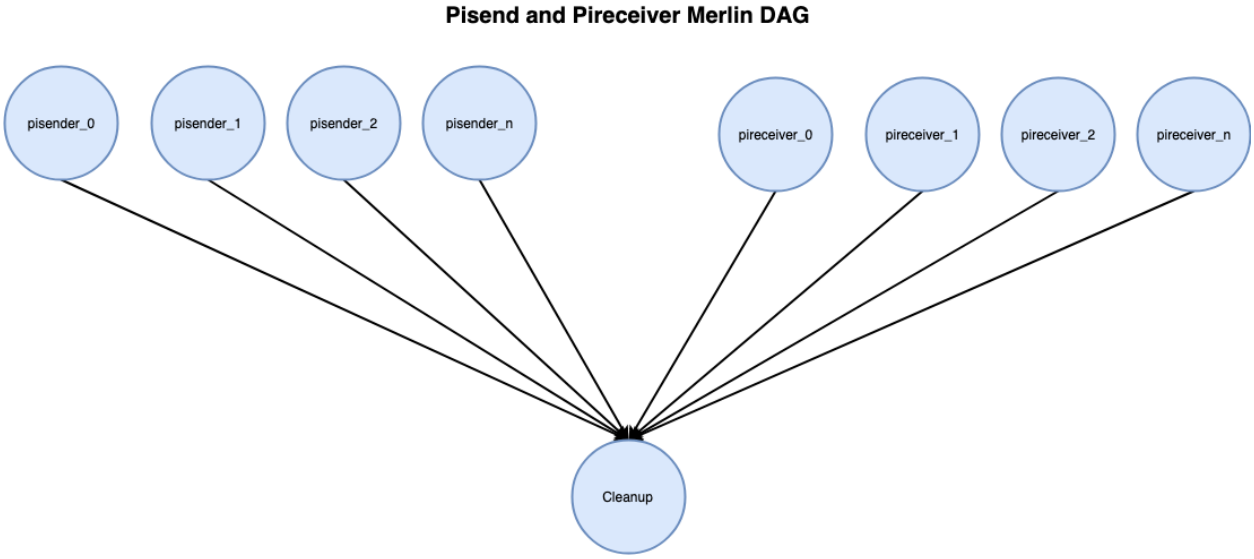
(continues on next page)

(continued from previous page)

```
      echo "DONE"
  - name: cleanup
    description: Clean up
    run:
      cmd: rm $(SPECROOT)/samples.csv
      depends: [start_federates_*]
```

DAG of the spec

Finally, we can look at the DAG of the spec to visualize the steps in the Study.



Orchestration Example

An example of orchestrating multiple simulation runs (e.g. Monte Carlo co-simulation) is given in the *Advanced Examples* Section.

Terminating HELICS

If executing from a C or C++ based program. Ctrl-C should do the right thing, and terminate the local program. If the co-simulation is running across multiple machines then the remaining programs won't terminate properly and will either timeout or if that was disabled potentially deadlock.

Signal handler facilities

The C shared library has some facilities to enable a signal handler.

```
/** Load a signal handler that handles Ctrl-C and shuts down all HELICS brokers, cores,
and federates then exits the process.*/
void helicsLoadSignalHandler();

/** clear HELICS based signal Handlers*/
void helicsClearSignalHandler();

/** Load a signal handler that handles Ctrl-C and shuts down all HELICS brokers, cores,
```

(continues on next page)

(continued from previous page)

```
and federates then exits the process.  This operation will execute in a newly created␣
↪and detached thread returning control back to the
calling program before completing operations.*/
void helicsLoadThreadedSignalHandler();
```

This function will insert a signal handler that generates a global error on known objects and waits a certain amount time, clears the print buffer, and terminates.

There are also threaded versions of these signal handlers. These generate a separate thread and detach it before executing the HELICS callback operations. This is useful in contexts where the main thread in the program needs to continue in order to close out operations. This is used in the python API to properly (most of the time) handle Ctrl-C operations and terminate the co-simulation.

NOTE : the signal handlers use unsafe operations, so there is no guarantee they will work, or that they will work as expected. Testing indicates they work in most situations and improve operations where needed but it is not 100% reliable or safe code. They make use of atomic variables, mutexes, and other constructs that are not technically safe in signal handlers. The primary use case is program termination so the effects are minimized and they usually work, but the unsafe nature of them should be kept in mind.

```
/** Load a custom signal handler to execute prior to the abort signal handler.
@details  This function is not 100% reliable it will most likely work but uses some␣
↪functions and
techniques that are not 100% guaranteed to work in a signal handler
and in worst case it could deadlock.  That is somewhat unlikely given usage patterns
but it is possible.  The callback has signature HelicsBool(*handler)(int) and it will␣
↪take the SIG_INT as an argument
and return a boolean.  If the boolean return value is HELICS_TRUE (or the callback is␣
↪null) the default signal handler is run after the
callback finishes; if it is HELICS_FALSE the default callback is not run and the default␣
↪signal handler is executed. If the second
argument is set to HELICS_TRUE the default signal handler will execute in a separate␣
↪thread(this may be a bad idea). */
void helicsLoadSignalHandlerCallback(HelicsBool (*handler)(int), HelicsBool␣
↪useSeparateThread);
```

It is also possible to insert a custom callback into the signal handler chain. Again this is not 100% reliable. But is useful for some language API's that do other things to signals. This allows for inserting a custom operation that does some other cleanup. The callback has a helics_boolean return value. If the value is set to `HELICS_TRUE`(or any positive value) then the normal or threaded Signal handler is called which aborts ongoing federations and exits. If it is set to `HELICS_FALSE` then the default callback is not executed. If the `useSeparateThread` call was set to true in the `helicsLoadSignalHandlerCallback` method then the HELICS portion of the callback is executed in a new thread. The signal handler will return before this portion of the handler is completed to all control to return to the main program to complete operations. Once again these operations are not guaranteed to be safe in a signal handler. In most cases they work and can be useful for usability. And in most cases they are used when desiring to terminate a program so consequences are minimal.

Signal handlers in C++

Facilities for signal handling in C++ were not put in place since it is easy enough for a user to place their own handlers which would likely do a better job than any built in ones, so a default one was not put in place at present though may be at a later time.

Generating an error

A global error generated anywhere in a federation will terminate the co-simulation.

```
/**
 * generate a global error through a broker  this will terminate the federation
 *
 * @param broker The broker to set the time barrier for.
 * @param errorCode the error code to associate with the global error
 * @param errorString an error message to associate with the error
 * @param[in,out] err An error object that will contain an error code and string if any␣
↪error occurred during the execution of the function.
 */
void helicsBrokerGlobalError(HelicsBroker broker, int errorCode, const char *errorString,
↪ HelicsError* err);

void helicsCoreGlobalError(HelicsCore core, int errorCode, const char* errorString,␣
↪HelicsError* err);

/**
 * Generate a global error from a federate.
 *
 * @details A global error halts the co-simulation completely.
 *
 * @param fed The federate to create an error in.
 * @param errorCode The integer code for the error.
 * @param errorString A string describing the error.
 */
HELICS_EXPORT void helicsFederateGlobalError(HelicsFederate fed, int errorCode, const␣
↪char* errorString);
```

Corresponding functions are available in the C++ API as well. Any global error will cause a termination of the co-simulation.

Some modifying flags

Setting the `HELICS_TERMINATE_ON_ERROR` flag to true will escalate any local error into a global one and terminate the co-simulation. This includes any mismatched configuration or other local issues.

Comments

Generally it isn't a wise idea to just terminate the co-simulation without letting everyone else know. If you control everything it probably works fine but as co-simulations get larger more care needs to be taken to prevent zombie processes and hung federates and brokers, which can cause issues on the next one. This is an evolving area of how best to handle terminating large co-simulations in abnormal conditions and hopefully the best practices will make it easier for users.

Profiling

As of versions 2.8 or 3.0.1 HELICS includes a basic profiling capability. This is simply the capability to generate timestamps when entering or exiting HELICS blocking call loops where a federate may be waiting on other federates.

Output

The profiling output can be either in the other log files or a separate file, and can be enabled at the federate, core, or broker levels. There are 3 messages which may be observed:

```
<PROFILING>test1[131072](created)MARKER<138286445040200|1627493672761320800>[t=-
↪9223372036.854776]</PROFILING>
<PROFILING>test1[131072](initializing)HELICS CODE ENTRY<138286445185500>[t=-1000000]</
↪PROFILING>
<PROFILING>test1[131072](executing)HELICS CODE EXIT<138286445241300>[t=0]</PROFILING>
<PROFILING>test1[131072](executing)HELICS CODE ENTRY<138286445272500>[t=0]</PROFILING>
```

The messages all start and end with and to make an xml-like tag. The message format is `FederateName[FederateID](federateState)MESSAGE<wall-clock time>[simulation time]`

The federate state is one of `created`, `initializing`, `executing`, `terminating`, `terminated`, or `error`.

The three possible `MESSAGE` values are:

- `MARKER` : A time stamp matching the local system up time value with a global time timestamp.
- `HELICS CODE ENTRY` : Indicator that the executing code is entering a HELICS controlled loop
- `HELICS CODE EXIT` : Indicator that the executing code is returning control back to the federate.

For `HELICS CODE ENTRY` and `HELICS CODE EXIT` messages the time is a steady clock time, usually the time the system on which the federate is running has been up. The `MARKER` messages have two timestamps for global coordination, `<steady clock time|system time>`. The system time is the wall clock time as available by the system, which is usually with reference to Jan 1, 1970 and in GMT.

The timestamp values are an integer count of nanoseconds. For all 3 message types they refer to the system uptime which is monotonically non-decreasing and steady. This value will differ from each computer on which federates are running, though. To calibrate for this there is a marker that gets triggered when the profiling is activated, indicating the local uptime that is synchronous across compute nodes. This matches a system uptime, with the global system time. The ability to match these across multiple machines will depend on the latency associated with time synchronization across the utilized compute nodes. No effort is made in HELICS to remove this latency or even measure it; that is, though the marker time is measured in nanoseconds it could easily differ by microseconds or even milliseconds depending on the networking conditions between the compute nodes.

Enabling profiling

Profiling can be enabled at any level of the hierarchy in HELICS and when enabled it will automatically enable profiling on all the children of that object. For example, if profiling is enabled on a broker, all associated cores will enable profiling and all federates associated with those cores will also have profiling enabled. This propagation will also apply to any child brokers and their associated cores and federates.

Broker profiling

Profiling is enabled via the command prompt by passing the `--profiler` option when calling `helics_broker`.

- `--profiler=save_profile2.txt` will clear save_profile2.txt and save new profiling data to a text file `save_profile2.txt`
- `--profiler_append=save_profile2.txt` will append profiling data to the text file `save_profile2.txt`
- `--profiler=log` will capture the profile text output to the normal log file or callback
- `--profiler` is the same as `--profiler=log`

Enabling this flag will pass in the appropriate flags to all children brokers and cores.

Core profiling

Profiling is enabled via the `coreinitstring` by adding a `--profiler` option.

- `--profiler=save_profile2.txt` will clear save_profile2.txt and save new profiling data to a text file `save_profile2.txt`
- `--profiler_append=save_profile2.txt` will append profiling data to the text file `save_profile2.txt`
- `--profiler=log` will capture the profile text output to the normal log file or callback
- `--profiler` is the same as `--profiler=log`

Enabling this flag will pass in the appropriate flags to all children federates.

Federate profiling

Profiling on a federate will recognize the same flags as a core, and pass them as appropriate to the core. However a federate also supports passing the flags and a few additional ones into the federate itself.

```
helicsFederateSetFlagOption(fed,HELICS_FLAG_PROFILING, HELICS_TRUE, &err);
```

can directly enable the profiling. If nothing else is set this will end up generating a message in the log of the root broker.

```
helicsFederateSetFlagOption(fed,HELICS_FLAG_PROFILING_MARKER, HELICS_TRUE, &err);
```

can generate an additional marker message if logging is enabled.

```
helicsFederateSetFlagOption(fed,HELICS_FLAG_LOCAL_PROFILING_CAPTURE, HELICS_TRUE, &err);
```

captures the profiling messages for the federate in the federate log instead of forwarding them to the core or broker.

Some can be set through the flags option for federate configuration. `--flags=profiling`, `local_profiling_capture` can be set through command line or configuration files. If enabling the `local_profiling_capture`, `profiling` must also be enabled; that is, just setting `local_profiling_capture` does not enable profiling. The profiling marker doesn't make sense anywhere but through the program call.

Notes

This capability is preliminary and subject to change based on initial feedback. In HELICS 3 there will probably be some additional command infrastructure to handle profiling as well added in the future.

If timing is done in the federate itself as well there will be a time gap; there is some processing code between the profiling message, and when the actual function call returns, but it is not blocking and should be fairly short, though dependent on how much data is actually transferred in the federate. Profiling does not work with callback federates.

Queries

Queries are an asynchronous means within a HELICS federation of asking for and receiving information from other federate components. A query provides the ability to evaluate the current state of a federation and typically addresses the configuration and architecture of the federation. Brokers, Federates, and Cores all have query functions. Federates are also able to define a callback for answering custom queries.

The general function looks like this:

C++

```
std::string query(const std::string& target, const std::string& queryStr)
```

Python

```
query_result = h.helicsCreateQuery(traget_string, query_string)
```

Targets

Each query must define a "target", the component in the federation that is being queried. The target is either specified in terms of the relationship to the querying federate (*e.g.* "broker", "core") or by name of the federation component (*e.g.* "dist_system_1_fed"). The table below lists the valid query targets; if a federate happens to be named one of the target names listed below, it can not be queried by that name. For example, naming one of your brokers "broker" will prevent it being a valid target of a query by name. Instead, any federate that queries "broker" will end up targeting their broker.

target	Description
`broker`	The first broker encountered in the hierarchy from the caller
`root`, `federation`, `rootbroker`	The root broker of the federation
`global`	Retrieve the data associated with a global variable
`parent`	The parent of the caller
`core`	The core of a federate. This is not a valid target if called from a broker
`federate`	A query to the local federate or the first federate of a core
`<object name>`	any named object in the federation can also be queried, brokers, cores, and federates

Query String

The `queryStr` is the specific data being requested; the tables below show the valid data provided by each queryable federation component. All queries return a valid JSON string with invalid queries returning a JSON with an error code and error message. (The only exception is the `global_value` query which just returns a string containing global value.)

As of HELICS 2.7.0 Queries have an optional parameter to describe a sequencing mode. There are currently two modes, `HELICS_SEQUENCING_MODE_FAST` which travels along priority channels and is identical to previous versions in which all queries traveled along those channels. The other mode is `HELICS_SEQUENCING_MODE_ORDERED` which travels along lower priority channels but is ordered with all other messages in the system. This can be useful in some situations where you want previous messages to be acknowledged as part of the federation before the query is run. The `global_flush` query is forced to run in ordered mode at least until after it gets to the specified target.

Federate Queries

The following queries are defined for federates. Federates may specify a callback function which allows arbitrary user defined Queries. The queries defined here are available inside of HELICS.

queryString	Description
name	the identifier of the federate [string]
exists	basic query if the federate exists in the Federation [T/F]
isinit	federate has entered init mode? [T/F]
state	current state of the federate as a string [string]
global_state	current state of the federate as a string [structure]
publications	current publications of a federate [sv]
publication_details	details of current publications of a federate [sv]
subscriptions	current subscriptions of a federate [sv]
inputs	current inputs of a federate [sv]
endpoints	current endpoints of a federate [sv]
input_details	details of current inputs of a federate [structure]
endpoint_details	details of current endpoints of a federate [structure]
dependencies	list of the objects this federate depends on [sv]
dependents	list of dependent objects [sv]
current_time	the current time of the federate [structure]
endpoint_filters	data structure with the filters for endpoints[structure]
dependency_graph	a graph of the dependencies in a federation [structure]
data_flow_graph	a structure with all the data connections [structure]
queries	list of available queries [sv]
version	the version string of the helics library [string]
tags	a JSON structure with the tags and values [structure]
barriers	a JSON structure with current time barriers [structure]
logs	any log messages stored in the log buffer [structure]
tag/<tagname>	the value associated with a tagname [string]
<tagname>	the value associated with a tagname [string]

The global_time_debugging and global_flush queries are also acknowledged by federates but it is not usually recommended to run those queries on a particular federate as they are more useful at higher levels. See the Core and Broker queries for more description of them. The difference between tag/<tagname> and <tagname> is that using the tag/ prefix can retrieve any tag and will return an empty string if the tag doesn't exist. Just using the tag name will not return tags of the same name as other queries and will generate an error response if the tag doesn't exist. The logs query will only contain information if log buffer size is set to greater than 0 by property or command. The logs query also works on cores and brokers that have been disconnected to retrieve buffered logs after the co-simulation has completed. This of course only works with the local instance.

Local Federate Queries

The following queries are defined for federates but can only be queried on the local federate, that is, the federate making the query. Federates may specify a callback function which allows arbitrary user defined Queries.

queryString	Description
updated_input_indices	vector of number of the inputs that have been updated [sv]
updated_input_names	names or targets of inputs that have been updated [sv]
updates	values of all currently updated inputs [structure]
values	current values of all inputs [structure]
time	the current granted time [string]

Core queries

The following queries will be answered by a core:

queryString	Description
`name`	the identifier of the core [string]
`address`	the network address of the core [string]
`isinit`	If the core has entered init mode [T/F]
`isconnected`	If the core has is connected to the network [T/F]
`publications`	current publications defined in a core [sv]
`inputs`	current named inputs defined in a core [sv]
`endpoints`	current endpoints defined in a core [sv]
`filters`	current filters of the core [sv]
`publication_details`	details of current publications defined in a core [structure]
`input_details`	details of current named inputs defined in a core [structure]
`endpoint_details`	details of current endpoints defined in a core [structure]
`filter_details`	details of current filters of the core [structure]
`federates`	current federates defined in a core [sv]
`dependenson`	list of the objects this core depends on [sv]
`dependents`	list of dependent objects [sv]
`dependencies`	structure containing dependency information [structure]
`federate_map`	a Hierarchical map of the federates contained in a core [structure]
`federation_state`	a structure with the current known status of the brokers and federates [structure]
`current_time`	if a time is computed locally that time sequence is returned [structure]
`global_time`	get a structure with the current time status of all the federates/cores [structure]
`current_state`	The state of all the components of a core as known by the core [structure]
`global_state`	The state of all the components from the components [structure]
`dependency_graph`	a representation of the dependencies in the core and its federates [structure]
`data_flow_graph`	a representation of the data connections from all interfaces in a core [structure]
`filtered_endpoints`	data structure containing the filters on endpoints for the core[structure]
`barriers`	a data structure with current time barriers [structure]
`queries`	list of dependent objects [sv]
`version_all`	data structure with the version string and the federates[structure]
`version`	the version string for the helics library [string]
`counter`	A single number with a code, changes indicate core changes [string]
`global_time_debugging`	return detailed time debugging state [structure]
`global_flush`	a query that just flushes the current system and returns the id's [structure]
`tags`	a JSON structure with the tags and values [structure]
`logs`	any log messages stored in the log buffer [structure]
`tag/<tagname>`	the value associated with a tagname [string]
`<tagname>`	the value associated with a tagname [string]

The `version` and `version_all` queries are valid but are not usually queried directly, but instead the same query is used on a broker and this query in the core is used as a building block.

Broker Queries

The following queries will be answered by a broker:

queryString	Description
name	the identifier of the broker [string]
address	the network address of the broker [string]
isinit	If the broker has entered init mode [T/F]
isconnected	If the broker is connected to the network [T/F]
publications	current publications known to a broker [sv]
endpoints	current endpoints known to a broker [sv]
inputs	current inputs known to a broker [sv]
filters	current filters known to a broker [sv]
federates	current federates under the brokers hierarchy [sv]
publication_details	details of current publications defined in a core [structure]
input_details	details of current named inputs defined in a core [structure]
endpoint_details	details of current endpoints defined in a core [structure]
filter_details	details of current filters of the core [structure]
brokers	current cores/brokers connected to a broker [sv]
dependson	list of the objects this broker depends on [sv]
dependencies	structure containing dependency information for the broker [structure]
dependents	list of dependent objects [sv]
barriers	a data structure with current time barriers [structure]
counts	a simple count of the number of brokers, federates, and interfaces [structure]
current_state	a structure with the current known status of the brokers and federates [structure]
global_state	a structure with the current state all system components [structure]
status	a structure with the current known status (true if connected) of the broker [structure]
current_time	if a time is computed locally that time sequence is returned, otherwise #na [string]
global_time	get a structure with the current time status of all the federates/cores [structure]
federate_map	a Hierarchical map of the federates contained in a broker [structure]
dependency_graph	a representation of the dependencies connections in all objects connected to a broker [structure]
data_flow_graph	a representation of the data connections from all interfaces in a federation [structure]
queries	list of dependent objects [sv]
version_all	data structure with the version strings of all broker components [structure]
version	the version string for the helics library [string]
counter	A single number with a code, changes indicate federation changes [string]
logs	any log messages stored in the log buffer [structure]
global_time_debugging	return detailed time debugging state [structure]
global_flush	a query that just flushes the current system and returns the id's [structure]
global_status	an aggregate query that returns a combo of global_time and current_state [structure]
time_monitor	get the current time as recorded from the current monitor federate [structure]
monitor	The name of the object used as a time monitor [string]

`federate_map`, `dependency_graph`, `global_time`,`global_state`,`global_time_debugging`, `barriers`, and `data_flow_graph` when called with the root broker as a target will generate a JSON string containing the entire structure of the federation. This can take some time to assemble since all members must be queried. `global_flush` will also force the entire structure along the ordered path which can be quite a bit slower. Error codes returned by the query follow http error codes for "Not Found (404)" or "Resource Not Available (400)" or "Server Failure (500)".

Usage Notes

Queries that must traverse the network travel along priority paths unless specified otherwise with a sequencing mode. The calls are blocking, but they do not wait for time advancement from any federate and take priority over regular communication.

The difference between `current_state` and `global_state` is that `current_state` is generated by information contained in the component so doesn't generate secondary queries of other components. Whereas `global_state` will

reach out to the other components to get up to date information on the state.

Error Handling

Queries that can't be processed or are not recognized return a JSON error structure. The structure will contain an error code and message such as:

```
{
  "error": {
    "code": 404,
    "message": "target not found"
  }
}
```

The error codes match with HTTP error codes to the extent possible.

Application API

There are two basic calls in the application API as part of a federate object In addition to the call described above a second version omits the "target" specification and always queries the local federate.

```
std::string    query(const std::string& queryStr)
```

There is also an asyncrhonous version (that is, non-blocking) that returns a `query_id_t` that can be use in `queryComplete` and `isQueryComplete` functions.

```
query_id_t     queryAsync(const std::string& target, const std::string& queryStr)
```

In the header `<helics\queryFunctions.hpp>` a few helper functions are defined to vectorize query results and some utility functions to wait for a federate to enter init, or wait for a federate to join the federation.

C API and interface API's

Queries in the *C API* have the same valid targets and properties that can be queried but the construction of the query is slightly different. The basic operation is to create a query using `helicsQueryCreate(target,query)`. Once created, the target or query string can be changed with `helicsQuerySetTarget()` and `helicsQuerySetQueryString()`, respectively.

This function returns a query object that can be used in one of the execute functions (`helicsQueryExecute()`, `helicsQueryExecuteAsync()`, `helicsQueryBrokerExecute()`, `helicsQueryCoreExecute()`, to perform the query and receive back results. The query can be called asynchronously on a federate. The target field may be empty if the query is intended to be used on a local federate, in which case the target is assumed to be the federate itself. A query must be freed after use `helicsQueryFree()`.

Timeouts

As long as timeouts are enabled in the library itself, queries have a timeout system so they don't block forever if a federate fails or some other condition occurs. The current default is 15 seconds. It can be changed by using the command line option `--querytimeout` on cores or brokers (or in `--coreinitstring` on cores). In a later version an ability to set this and some other timeout values through properties will likely be added (HELICS 3.1). If the query times out a value of #timeout will be returned in the string.

Example

A full co-simulation example showing how queries can be used for *dynamic configuration can be found here* (with the source code in the HELICS Examples repository).

Simultaneous Co-simulations

Sometimes it is necessary or desirable to be able to execute multiple simultaneous simulations on a single computer system. Either for increased parallelism or from multiple users or as part of a larger coordinated execution for sensitivity analysis or uncertainty quantification. HELICS includes a number of different options for managing this and making it easier.

General Notes

HELICS starts with some default port numbers for network communication, so only a single broker (per core type) with default options is allowed to be running on a single computer at a given time. This is the general restriction on running multiple simultaneous co-simulations. It is not allowed to have multiple default brokers running at the same time, the network ports will interfere and the co-simulation will fail.

There are a number of ways around this and some tools to assist in checking and coordinating.

Specify port numbers

The manual approach works fine. All the network core types accept user specified port numbers. The following script will start up two brokers on separate port numbers:

```
helics_broker --type=zmq --port=20200 &
helics_broker --type=zmq --port=20400 &
```

Federates connecting to the broker would need to specify the `--brokerport=X` to connect with the appropriate broker. These brokers operate independently of each other. The port numbers assigned to the cores and federates can also be user assigned but if left to default will be automatically assigned by the broker and should not interfere with each other.

An example of configuring multiple federations to run on a single compute node using port numbers *has been written up over here* (and the source code can be found in the HELICS Examples repo).

Use Broker server

For the zmq, zmqss, tcp, and udp core types it is possible to use the broker server.

```
helics_broker_server --zmq
helics_broker_server --zmqss
helics_broker_server --tcp
helics_broker_server --udp
```

multiple broker servers can be run simultaneously

```
helics_broker_server --zmq --tcp --udp
```

The broker server currently has a default timeout of 30 minutes on the default port and will automatically generate brokers on separate ports and direct federates which broker to use. The duration of the server can be controlled via

```
helics_broker_server --zmq --duration=24hours
```

It will also generate brokers as needed so the `helics_broker` does not need to be restarted for every run.

By default the servers will use the default ports and all interfaces. This can be configured through a configuration file

```
helics_broker_server --zmq --duration=24hours --config=broker_config.json
```

this is a json file. The sections in the json file include the server type For example

```
{
  "zmq": {
    "interface": "tcp://127.0.0.1"
  },
  "tcp": {
    "interface": "127.0.0.1",
    "port": 9568
  }
}
```

There is also a *webserver* that can be run with the other broker servers.

Use of keys

If there are multiple users and you want to verify that a specific broker can only be used with federates you control. It is possible to add a key to the broker that is required to be supplied with the federates to connect to the broker. **NOTE:** *this is not a cryptographic key, it is just a string that is not programmatically accessible to others.*

```
helics_broker --type=zmq --key=my_broker_key
```

Federates then need to supply the key as part of the configuration string otherwise the broker will return an error on connection. This is like a fence that prevents some accidental interactions. The rule is that both the federate and broker must provide no key or the same key.

Targeted Endpoints

Endpoints can be configured to send messages to other specific endpoints be two means, `default` and `targeted`.

- `default` - Convenient way of specifying the intended destination of a message from the configuration file. Allows the API call that sends the message to leave the destination field blank. DOES NOT create a specific dependency between the sending and receiving endpoint for the HELICS core library to use when figuring out which federate should be granted what time.
- `targeted` - Creates a dependency link between the sending and receiving endpoints, just like in value exchanges. This allows the HELICS core library to make more efficient decisions about when to grant times to federates and can lead to more efficient co-simulation execution. Targeted endpoints can specify a list of federates that all their messages go to.

Targeted endpoint configuration example:

```
{
  "name": "EV_Controller",
  "coreType": "zmq",
  "timeDelta": 1.0,
  "endpoints": [
    {
      "name": "EV_Controller/EV6",
      "global": true,
      "destinationTarget": "charger/ep"
    }
```

(continues on next page)

(continued from previous page)

```
  ]
}
```

Timeouts

HELICS has a number of properties to detect things going wrong in the co-simulation. Many of these center around detecting a deadlock condition in federates or even potential bugs in HELICS itself.

Timeouts are the main way this is done. What timeouts make the most sense and how to use them is an ongoing effort and will evolve as the use of HELICS grows and we gain more experience with larger co-simulations.

There are 2 main categories of timeout operations. The first has to do with the connection phase, about how long federates should wait for the broker to be available or network resources to be available before generating an error. The second category deals more with a co-simulation getting stuck and unable to proceed in time either from a single federate hanging or a lost network connection or something of that nature. For small co-simulations it is easy enough to just kill things manually if something goes wrong, but it can be potentially problematic as the co-simulations get larger and more complicated.

Background

Typical brokers and cores in HELICS operate on a separate thread processing messages in the background from the user code. They also make use of system timers and can generate messages for processing independent of the main loop. The foundation of this is a heartbeat timer on the core and brokers. All options in HELICS that take a time are defaulted to milliseconds if only a number is supplied in the argument field. They also can take a time unit with the number such as `5 s` or `2 min` or `265234 ms`. Using `--tick` sets up the heartbeat which is fundamental to timeouts at the broker and core level. Enabling this requires HELICS be built with ASIO, so in situations where the CMake option `-DHELICS_DISABLE_ASIO=ON` was given all timeouts are non-functional. The tick timer can also be disabled using `--disable_timer`,`--no_tick` or `--debugging`. (NOTE: debugging also changes a few other things). The tick effectively establishes a timing resolution for the timeouts, if the tick is set to 1 second timeouts will have to be exceeded by at least 1 tick for actions to trigger. The default tick is 5 seconds so generally timeouts should be specified in multiples of the tick timer or slightly less than an integer multiple if desired to trigger earlier.

Co-simulation startup

Starting up a co-simulation often depends on a number of factors. There are multiple pieces in a co-simulation: at least 1 broker and often several federates. These communicate via network resources. And things can go wrong, so you don't want to have the co-simulation just hang forever waiting on certain pieces, so the most general option is `--timeout`. This is the timeout for establishing a broker connection, and it is also the default for some other more specific timeouts. The default is 30 seconds. If you are setting up a large co-simulation which may have a lot of contention during the setup phase, it may be necessary to lengthen this, or if you are not starting up the broker before the federates it can be necessary to change to a longer time. Currently 30 seconds is used as a compromise that works in most settings where HELICS is used.

When establishing a network connection sometimes the necessary ports and socket resources are not available in which case the `--network_timeout` applies. This defaults to the `--timeout` but it can be specified independently if desired. The core will use this timeout for retries on network resources before generating an error. The network timeouts are not dependent on the tick timer so will apply even if the timers have been disabled.

Co-simulation operation

While a co-simulation is executing, bad things can happen; networks can go down, federates can crash. If left on its own this may result in a co-simulation hanging and doing nothing while waiting for the failed federate. Detecting and handling this is the primary role of the heartbeat timer. If a broker has not received any communication in a tick, it sends out a ping to its parent or children. If the parent responds then no action is taken. If no response is received in a

certain period of time an error is generated and the process is terminated. Sometimes cores are not able to respond to pings, in which case the `--slowresponding` flag should be used on that federate/core/broker to prevent co-simulation termination from lack of ping responses. The `--debugging` option specified early is really a macro flag which turns on `--slowresponding` and `--disable_timer`. The primary use case is for doing stepwise debugging of a federate in a code debugger; otherwise, the ticks and other federates could cause issues from the timeouts and pings to the federate being debugged.

At the federate level a `--granttimeout` option is available. This depends on ASIO being compiled in and works similar to the real time mode of federates. At present it is primarily for diagnostics as it cannot generate an error, though this may change at some point in the future. It is disabled by default since some federates could legitimately take a long time to execute so having a timeout on others doesn't make sense. So any grant timeout needs to be handled based on the user knowledge of how the different federates operate. This operation is subject to change but at present the operation is handled in 4 stages.

- 1X timeout print warning message
- 3X timeout request a resend of timing message from blocking dependencies (in case of partial communications failure)
- 6X timeout print timing diagnostic information and send message to parent for additional debugging information
- 10X print additional warning – this is when additional corrective action may be taken but that is a work in progress

It is expected this process will change as more experience with it is gained.

At the broker level a `--maxcosimduration` option is available. By default it is disabled. If specified on a broker or core the co-simulation will terminate if running for more than the specified total time. This can be useful if running a large number of co-simulations in an automated fashion and the expected time is known. This can prevent a situation where a single co-simulation hangs indefinitely blocking others and doing no useful work.

Other timeouts

There are two other timeouts that can be specified. The first is a query timeout (`--querytimeout=X`). The default is 15 seconds. This applies to queries which are blocking calls, and if a federate is shutting down or fails as a query is in flight it is possible (not likely) that it could result in a non-response and the query never completing. In that rare situation the query timeout will trigger and result in an error on the query. In very large co-simulations it is possible a complex query could exceed the timeout without there being a problem so it may be necessary to lengthen it.

The `--errortimeout` will specify a certain amount of time to wait after a global error is generated to shutdown the co-simulation network. The default is 10s. This would potentially give time for additional queries and diagnostics in the event of an error. This would trigger if a global error is generated in a federate or core.

Final notes

The area of timeouts is subject to active development and changes are expected in the future. The operation of timeouts is subject to operating system constraints and timeouts may operate more as a minimum bound on heavily loaded systems with the times exceeding the specified value before action is taken.

Introduction

Generally, most federates have both inputs they depend on and outputs on which other federates depend and configuring federates so that each one updates its own model under the correct circumstances is challenging. HELICS largely handles this through its own timing algorithm and also provides many configuration flags and settings to allow the user to optimize the timing and sequencing of federates to produce the most optimal performance. This document provides a few hints that, based on experience, we have found helpful in getting the most out of a HELICS-based co-simulation.

uninterruptible

By default, a federate in HELICS is granted time when either of two conditions are met:

1. HELICS has determined it is safe for the federate to simulate its requested time based on the simulated times of all the federates from which it has inputs.
2. The federate has received an update on one of its value or message interfaces.

The latter logic is implemented under the philosophical assumption that a federate MAY want to produce a new output if one of its inputs has changed. By granting a federate an earlier time it gives the federate the ability to evaluate its new input and decide what, if any, actions it needs to take. There is some computational cost, though, to granting an earlier time as the federate must perform some kind of minimal computation to evaluate the new input. For a federate with a large number of inputs or inputs that receive changes frequently, these early time grants can add up to a relatively large amount of unnecessary computation.

To prevent this, HELICS provides for federates a flag called "uninterruptible" which removes this second option for determining the time to grant and only allows a federate to be granted the time it requests.

The `uninterruptible` flag had a significant effect on the total number of calls into the `federateRequestTime` method, the most significant source of time delay found in our testing. By reducing the number of times this method is called, especially where it may not be necessary, moderate performance gains were observed in communication-heavy federates.

Setting the `uninterruptible` via the JSON configuration is trivial:

```
{
  "name": "sub_7",
  "period": 15,
  "logging": "warning",
  "uninterruptible": true,
  "publications": [
    {
      "global": false,
      "key": "unresponsive_mw",
      "type": "double"
    },
  ]
  "subscriptions": [
    {
      "key": "pypower/LMP_7",
      "type": "double"
    },
    {
      "key": "gld_7/distribution_load",
      "type": "complex"
    },
  ]
}
```

Input `isUpdated`

HELICS provides a method which allows the federate to check if a specific endpoint or subscription has been updated since it was last read. In this context, "update" simply means the input has received a value since the last time it was checked. This new input may be identical in value to the last time the interface was updated and the flag simply indicates the publisher has sent another value to the subscriber's input.

If a value change is not guaranteed every time the simulator is granted time, or if multiple values are being updated in lockstep with a change on one indicating many have changed, a call to the `isUpdated` method prior to asking for the latest value can significantly reduce total computation time, and net substantial performance benefits.

```
...
hFed = helics.helicsCreateValueFederateFromConfig(helicsConfig)
subLMP = helics.helicsFederateGetInputByTarget(hFed, tso_federate + "/LMP_" + bus)
...

# ==================== Time step looping under HELICS ==========================
helics.helicsFederateEnterExecutingMode(hFed)

while time_granted < time_stop:
    ...
    time_granted = int(helics.helicsFederateRequestTime(hFed, nextHELICSTime))
    # Faster to check than to always grab the input every time the federate
    #   is granted time
    if helics.helicsInputIsUpdated(subLMP):
        LMP = helics.helicsInputGetDouble(subLMP)
        aucObj.set_lmp(LMP)
    ...
```

onlyUpdateOnChange

Similar to what the `isUpdated()` API call achieves, there is also a configuration flag that can be set to force HELICS to prevent signals from being sent (if set on the publication) or from being received (if set on the subscription) if the value of the signal has not changed. This adds a slight burden to HELICS as it now compares each published or subscribed value to the previous one to decide whether the signal will appear on the receivers interface. In cases where the computational cost of federate is large, though, this can end up being a net increase in performance.

Furthermore, a `tolerance` value can be set for each interface to define a degree of change that can be considered no-change. This allows very small changes in the published value to be ignored and treated as no change to the signal.

Configuring this in the JSON is not difficult:

```
{
  "name": "sub_7",
  "period": 15,
  "logging": "warning",
  "publications": [
    {
      "global": false,
      "key": "unresponsive_mw",
      "type": "double",
      "onlyUpdateOnChange": true,
      "tolerance": 0.01
    },
  ]
  "subscriptions": [
    {
      "key": "pypower/LMP_7",
      "type": "double"
    },
    {
```

(continues on next page)

(continued from previous page)

```
      "key": "gld_7/distribution_load",
      "type": "complex"
    },
  ]
}
```

Translators

Translators are HELICS objects that can be used to bridge the gap between values (publications, subscriptions, inputs) and messages (endpoints). These two types of signals come from the need to model different data: physical data (*e.g.* voltage, speed, energy) use value interfaces and abstract data (*e.g.* control signals, measurements) use message interfaces. Each type of interface has unique properties with advantages and disadvantages so it pays to think through which type of interface best suits a particular piece of data being exchanged.

But the world is messy and not ideal. Maybe you are working with a federation you wrote before you were a HELICS Master. Maybe you're inheriting somebody else's messy federation. Maybe the tools you have to work with only implemented value interfaces but you really need to use a message interface. Translators exist to bridge these two interface types and allow data exchanges that would not otherwise be possible.

Translator Basics

To HELICS, a translator is a specialized federate with one input, one publication (output) and one endpoint. A value sent to the input will result in a message sent to any targets of the translator's endpoint and any messages sent to the endpoint will result in a publication to any subscribing federates. As a federate, there will always be a one simulated nanosecond delay between when the data arrives at the input or endpoint and is sent out on the endpoint or publication, respectively. That is, the work of the translator takes a non-zero amount of simulated time.

Translators come in three types that determine how the data is translated between values and messages: JSON, custom, and binary. The binary option is unlikely to be useful in virtually all cases and won't be discussed. The custom option utilizes the `helicsTranslatorSetCustomCallback()` function to allow a user to define a translation process directly via a custom function that gets called anytime the translator needs to do its work. Using a custom translator may be required for special translation needs but is not expected to be needed for the typical use cases.

That leaves the JSON translator type, where the message coming out of or going into the endpoint is structured as simple JSON formatted as:

```
{
    "type": <HELICS data type>,
    "value": <value>
}
```

for example:

```
{
  "type": "double",
  "value": 56.78943
}
```

In this example, if the translator's value input received data from a publication that was formatted as a double of 56.78943, it would produce the above JSON on the endpoint output. Similarly, if the translator's endpoint received the above JSON, it would publish out a double value of 56.78943.

Translator Configuration

Translators can be defined by any federate and either linked into the federation via a core or a federate, with the latter creating a global translator which simplifies the addressing of the input, publication, and endpoint of the translator.

Translators can add their own sources and destinations for their interfaces but can also be targeted by configuration of other federates. That is, both the translator itself and the federates that want to use the translator have the ability to connect to its inputs, publications, and outputs. Because all federates connected to the output and endpoint will receive the translated data, the data-passing topology of the federation should be designed carefully. To keep certain data from going to certain federates, multiple translators may be required.

Just like inputs, publications, endpoints, and filters, translators can be defined via JSON configuration by any federate as follows:

```
"translators": [
   {
     "name": "test translator",
     "type": "JSON",
     "global": true,
     "info": "",
     "source_target": <name or key, (string, can be lists)>
     "destination_target: <name or key, (string, can be lists)>
   }
]
```

APIs exist for setting all of these parameters and are further defined in the *Configuration Options Reference*.

DIY Translator

If for some reason the HELICS provided translators are not able to meet your needs, you can always implement similar but more highly customized functionality through a custom federate. Though this is the use case intended to be supported via setting a custom callback for a translator, there may be other limitations that don't allow the use of a custom callback function. With such a federate, it would be possible to, for example, translate values on the same interface differently based on who sent them, allow message interfaces to conform to a non-JSON format, and implement data type translation. You would use the same APIs used for writing any other federates, so in this regard it is just another federate. In this case, though, this custom translator federate doesn't model any system and just acts as a data translator.

Example

Interacting with a Running Simulation

Starting in HELICS 2.4 there is a webserver that can be run with the helics_broker or helics_broker_server. This requires building with Boost enabled. The Webserver can be disabled by the `HELICS_DISABLE_BOOST=ON` or `HELICS_DISABLE_WEBSERVER=ON` options being set.

Startup

The webserver can be started with the option `--http` to start a restful interface using HTTP, or `--web` to start a server using websockets. For example to run a broker server with zmq and the webserver active for 30 minutes, you can use the following:

```
helics_broker_server --http --zmq --duration 30minutes
```

The `--duration` is optional and the default is 30 minutes but any time can be specified.

The web server is configured by default on the localhost address port 43542 to prevent interference with the helics runner or other tools running on standard ports. If you want to configure this it can be done through a configuration

file. The format is json.

```
{
  "http": {
    "port": 8080,
    "interface": "0.0.0.0"
  },
  "websocket": {
    "port": 8008,
    "interface": "0.0.0.0"
  }
}
```

Then it can be specified on the command line like so:

```
$ helics_broker_server --web --zmq --config broker_server_config.json
```

The configuration will then make the REST web server accessible on any interface on port 8080 and a WebSocket server on port 8080. The port in use can be specified in a configuration file, or via command line such as

```
$ helics_broker_server --web --zmq --http_server_args="--http_port=80"
```

Arguments are passed to servers using an option in the form `--<server>_server_args`, and in that arg field `--<server>_port` and `--<server>_interface` are valid arguments. Valid server names are `http`, `websocket`, `zmq`, 'tcp', and `udp`, and eventually `mpi`. The http web server also acknowledges `HELICS_HTTP_PORT` as an environment variable. The websocket server acknowledges `HELICS_WEBSOCKET_PORT` for the port numbers of the respective servers.

By default the webserver only listens for connections on the localhost network interface, this can be modified by specifying the external address directly or by declaring `--external` in the arguments.

```
$ helics_broker_server --web --zmq --http_server_args="--http_port=8080 --external"
```

REST API

The running webserver will start a process that can respond to HTTP requests.

HTTP actions

HTTP VERB	Description
GET	Make a query, usually with nothing in the message body
PUSH	most general command, usually for creating a broker, but other actions are possible
SEARCH	make a query mostly with data in the body
PUT	create a broker
DELETE	remove a broker

Parameters

parameter	Description
`command`	specify the command to use if not implied from the HTTP action, primarily PUSH
`broker`	The broker to target a focused request, or the name of a broker to create or delete
`type`	For commands that create a broker, this is the type of the broker to create
`target`	The actual object to target in a query
`query`	The query to execute on the specified target
`args`	The command line args to pass into a created broker

Valid commands for the `command` parameter in either JSON or the URI:

- `query`, `search` : run a query
- `create` : create a broker
- `delete`, `remove` : remove a broker

Websocket API

The websocket API will always respond in a JSON packet. For search/get operations where the response is a JSON value that JSON will be returned. for other responses, they are converted to a JSON.

For create/delete commands the response will be:

```
{
  "status": 0
}
```

```
{
  "status": 0,
  "value": "<query result>"
}
```

For queries that are not a json value the response will be:

```
{
  "status": 401,
  "error": "error message"
}
```

For queries that did not result in a valid response the response will be:

```
{
  "status": 404,
  "error": "error message"
}
```

The status code corresponds to the most appropriate html error codes.

Making queries

As a demo case there is a `brokerServerTestCase` executable that can be built when compiling the main HELICS library from source. Running this example starts a webserver on the localhost using port 80.

The response to queries is a string either in plain text or json. For example:

```
localhost/brokers
```

will return

```
{
  "brokers": [
    {
      "address": "tcp://127.0.0.1:23408",
      "isConnected": true,
      "isOpen": false,
      "isRoot": true,
      "name": "brokerA"
    },
    {
      "address": "tcp://127.0.0.1:23410",
      "isConnected": true,
      "isOpen": true,
      "isRoot": true,
      "name": "brokerB"
    }
  ]
}
```

Other queries should be directed to a specific broker such as:

```
http://localhost/brokerA/brokers
```

which will produce a string vector:

```
[41888-wfQ8t-GIGjS-dndI3-e7zuk;41888-e9KF2-HAfm8-Rft0w-JLV4a]
```

The following:

```
http://localhost/brokerA/federate_map
```

will produce a map of the federates in the federation:

```
{
  "brokers": [],
  "cores": [
    {
      "federates": [
        {
          "id": 131072,
          "name": "fedA_1",
          "parent": 1879048192
        }
      ],
```

(continues on next page)

(continued from previous page)


```
      "id": 1879048192,
      "name": "41888-wfQ8t-GIGjS-dndI3-e7zuk",
      "parent": 1
    },
    {
      "federates": [
        {
          "id": 131073,
          "name": "fedA_2",
          "parent": 1879048193
        }
      ],
      "id": 1879048193,
      "name": "41888-e9KF2-HAfm8-Rft0w-JLV4a",
      "parent": 1
    }
  ],
  "id": 1,
  "name": "brokerA"
}
```


Making an invalid query will produce:

`http://localhost/brokerA/i_dont_care` -> `#invalid`

Queries can be make in a number of different formats, the following are equivalent:

- `http://localhost/brokerA/publications`
- `http://localhost/brokerA`
- `http://localhost/brokerA?query=publications`
- `http://localhost/brokerA/publications`
- `http://localhost/brokerA/root/publications`
- `http://localhost?broker=brokerA&query=publications&target=root`

In the example, these will all produce `[pub1;fedA_1/pub_string]` which is a list of the publications. POST requests can also be made using a similar format.

Queries

Currently any query is accessible through this interface. Queries have a target and a query. The target is some named object in the federation and the query is a question. The available queries are listed *here*. More are expected to be added.

Json

For both the websockets and REST API they can accept arguments in JSON format. For the REST API the parameters can be a combination of arguments in the URI and JSON in the body of the request For example:


```
{
  "command": "search",
  "broker": "broker1",
  "target": "federate0",
```


(continues on next page)

(continued from previous page)

```
  "query": "current_state"
}
```

The most likely use case for this will be as a component for a more sophisticated control interface, so a more user friendly setup will be using the webserver as a back-end for control, debugging, information, and visualization of a running co-simulation.

Whereas the *Fundamental Topics* provided a broad overview of co-simulation and a good step-by-step introduction to setting up a HELICS co-simulation, the Advanced Topics section assumes you, the reader, have a familiarity and experience with running HELICS co-simulations. If that's not the case, it's well worth your while to go review the *Fundamental Topics* and *corresponding examples*. In this section it will be assumed you know things like:

- The difference between value and message passing in HELICS
- How to configure HELICS federate appropriately
- Familiarity with the common HELICS APIs (*e.g.* requesting time, getting subscribed values, publishing values)
- Experience running HELICS co-simulations

The Advanced Topics section will dig into specific features of HELICS that are less commonly used but can be very useful in particular situations. Each section below provides a description of the feature, what it does, the kind of use case that might utilize it, and then links to examples that demonstrate an implementation. It's important to note that there are many other HELICS features and APIs not demonstrated here that can also be useful. As they say in academia, we'll leave it as an exercise to the reader to discover these. (Hint: The *API references* and the *Configuration Options Reference* are good starting points to see what's out there in the broader HELICS world.)

The Advanced Topics will cover:

- ***Aliases*** - HELICS 3.3 introduced the notion of aliases. Aliases allow a mapping of an interface key to a different string.
- ***Architectures*** - Introduction to different ways to connect federates, cores, and brokers to manage efficient passing of signals in a co-simulation.
- ***Broker Hierarchies*** - Purpose of broker hierarchies and how to configure a HELICS co-simulation to implement one.
- ***Callbacks*** - Over time a number of callbacks have been added for various operations and stages of the life cycle of a federate. This document describes the different callbacks available.
- ***Callback Federates*** - HELICS 3.3 introduced a beta test for callback federates which allow a federate to operate purely inline with a core based solely on callbacks. This can allow a much higher number of federates on a given system than was previously possible.
- ***Command Interface*** - HELICS v3 introduced the command interface as a method of asynchronously communicating between federates.
- ***helics_connector*** - App that creates connections between federates as a separate process from the declaration of said interfaces.
- ***Cores*** - Discussion of the different types of message-passing buses and their implementation as HELICS cores.
- ***Dynamic Federations*** - Sometimes it is useful to have a federate that is not ready at the beginning of co-simulation. This is a dynamic federation. There are various levels of this (not all are available yet) and this document discusses some aspects of dynamic co-simulation.
- ***Encrypted Communication*** - How to encrypt communication between HELICS brokers/federates.
- ***Environment Variables*** - HELICS supports some environment variables for configuration of a federate or broker.

- ***Iteration*** - Setting up federates so that they can iterate without advancing simulation time to achieve a more consistent state.
- ***Multi-compute-node Co-simulation*** - Executing a co-simulation across multiple compute nodes.
- ***Multi-Protocol Brokers (Multi-broker) for Multiple Core Types)*** What to do when one type of communication isn't sufficient.
- ***Multi-Source Inputs*** - Using inputs (rather than subscriptions), it is possible to accept value signals from multiple sources. This section discusses the various tools HELICS provides for managing how to handle/resolve what can be conflicting or inconsistent signal data.
- ***Networking*** - How to configure HELICS to run in more challenging networking environments.
- ***Orchestration Tool (Merlin)*** - Brief guide on using Merlin to handle situations where a HELICS co-simulation is just one step in an automated analysis process (*e.g.* uncertainty quantification) or where assistance is needed deploying a large co-simulation in an HPC environment.
- ***Program termination*** - Some additional features in HELICS related to program shutdown and co-simulation termination.
- ***Profiling*** - Some profiling capability for co-simulations.
- ***Queries*** - How queries can be used to get information on HELICS brokers, federates, and cores.
- ***Simultaneous co-simulations*** - Options for running multiple independent co-simulations on a single system.
- ***Targeted Endpoints*** - details on the new targeted endpoints in HELICS 3.
- ***Timeouts*** - HELICS includes a number of timeouts to prevent failed operations from continuing indefinitely, the various timeout options are discussed in this document.
- ***Timing Optimization*** - Guidance and recommendation on how to configure and set-up timing to optimize federation performance.
- ***Translators*** - Translators provide a means of HELICS message interfaces to communicate with HELICS value interfaces and vice versa.
- ***Webserver API*** - How to interact with a running co-simulation using a REST-based web API.

2.4 Examples

The examples provided in the user guide begin with a simple co-simulation that you should be able to execute with only python and HELICS installed. If you have not installed HELICS yet, navigate to the ***installation page***.

What are we modeling?

The model for the examples is a co-simulation of a number of electric vehicle (EV) batteries and a charging port. A researcher may pose the question, "What is the state of charge of batteries on board EVs as they charge connected to a charging port?"

This can be addressed with a simple two-federate co-simulation, as demonstrated in the Fundamental Examples, or with a more complicated multi-federate co-simulation modeled in the Advanced Examples. In each learning path, modules are provided to the user to demonstrate a skill. The Advanced Examples build on the basics to make the co-simulation better emulate reality.

Learning Tracks

There are two learning tracks available to those hoping to improve their HELICS skills. The Fundamental Examples are designed for users with no experience with HELICS or co-simulation. The Advanced Examples are geared towards users who are familiar with HELICS and feel confident in their abilities to build a simple co-simuluation. The Advanced Examples harness the full suite of HELICS capabilities, whereas the Fundamental Examples teach the user the basics.

These two learning tracks each start with a "base" model, which should also be considered the recommended default settings. Examples beyond the base model within a track are modular, not sequential, allowing the user to self-guide towards concepts in which they want to gain skill.

A Word on HELICS CLI

All (or almost all) of the HELICS User Guide examples utilize a common tool and command for launching the co-simulation:

```
helics run --path=runner.json
```

This utilizes a tool we call `helics_cli` that is (optionally) *installed with the Python language binding*. Not only does HELICS CLI take care of launching the co-simulation, it also manages the logging and error/warning messages that are often printed to console when running code in a stand-alone manner. Generally, using HELICS CLI is the recommended way to run a federation but if there are particular needs you have that HELICS CLI can't meet, all federations can simply be run by launching each individual federate sequentially. The contents of the HELICS CLI runner.json shows the commands it uses to launch each federate and those can simply be copied and pasted into a set of command-line prompts manually launch the co-simulation.

2.4.1 Fundamental Examples

The Fundamental Examples teach three concepts to build on a default setup:

Base Example Co-Simulation

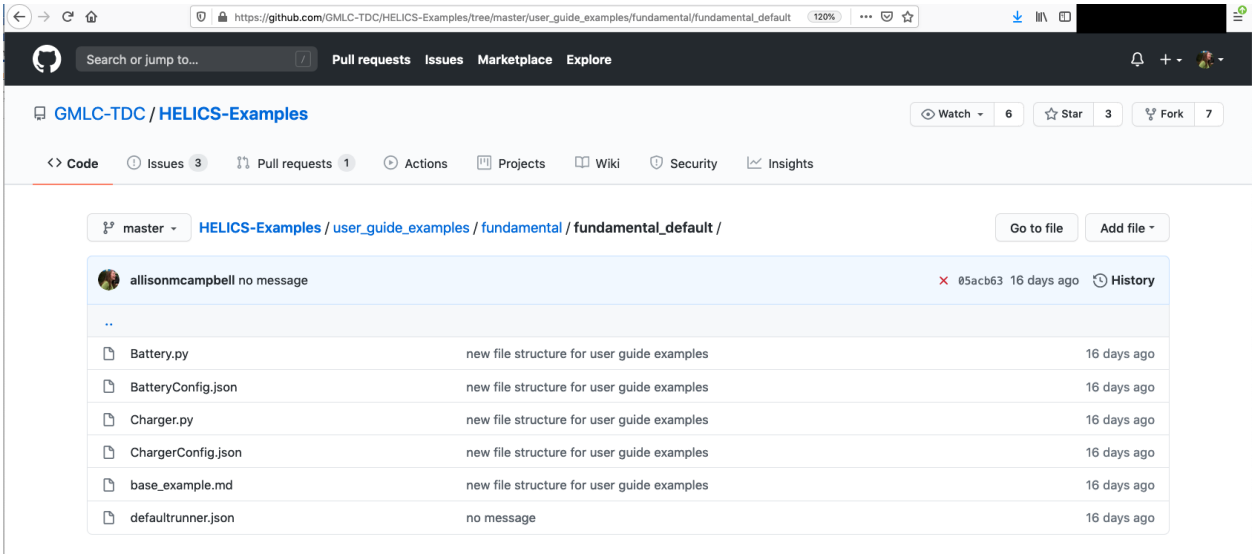
The Base Example walks through a simple HELICS co-simulation between two python federates. This example also serves as the recommended defaults for setting up a co-simulation. The base example described here will go into detail about the necessary components of a HELICS program. Subsequent examples in the Fundamental Examples section will change small components of the system.

The Base Example tutorial is organized as follows:

- *Example files*
- *Default Setup*
  - *Messages and Communication: pub/sub*
  - *Simulator Integration: External JSON*
  - *Co-simulation Execution*
- *Questions and Help*

Example files

All files necessary to run the Base Example can be found in the Fundamental examples repository:



The files include:

- Python program and configuration JSON for Battery federate
- Python program and configuration JSON for Charger federate
- HELICS runner JSON to enable execution of the co-simulation

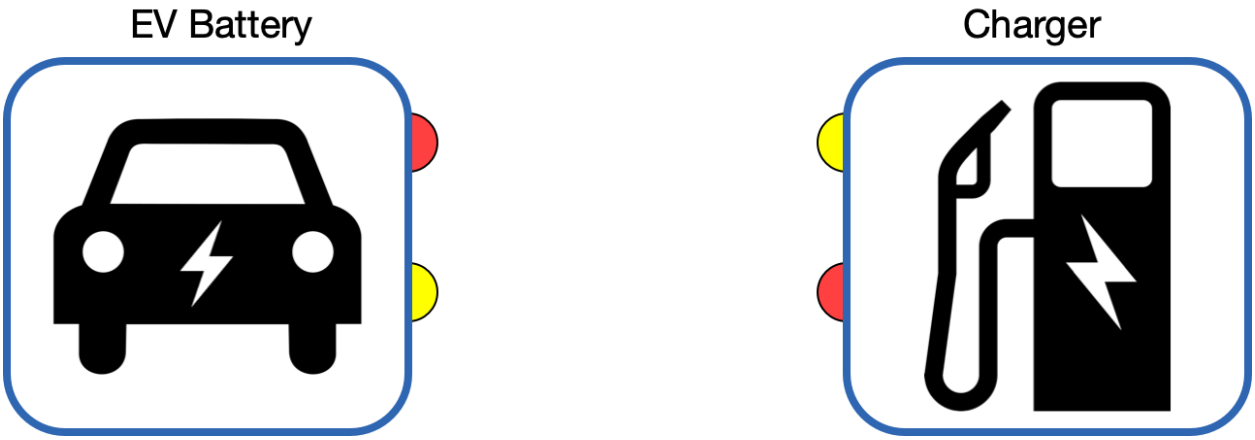
Default Setup

The default setup, used in the Base Example, integrates the federate configurations with external JSON files. The message and communication configurations are publications and subscriptions. This section introduces federate configuration of publications (pubs) and subscriptions (subs) with JSON files and how to launch the co-simulation with the HELICS runner.

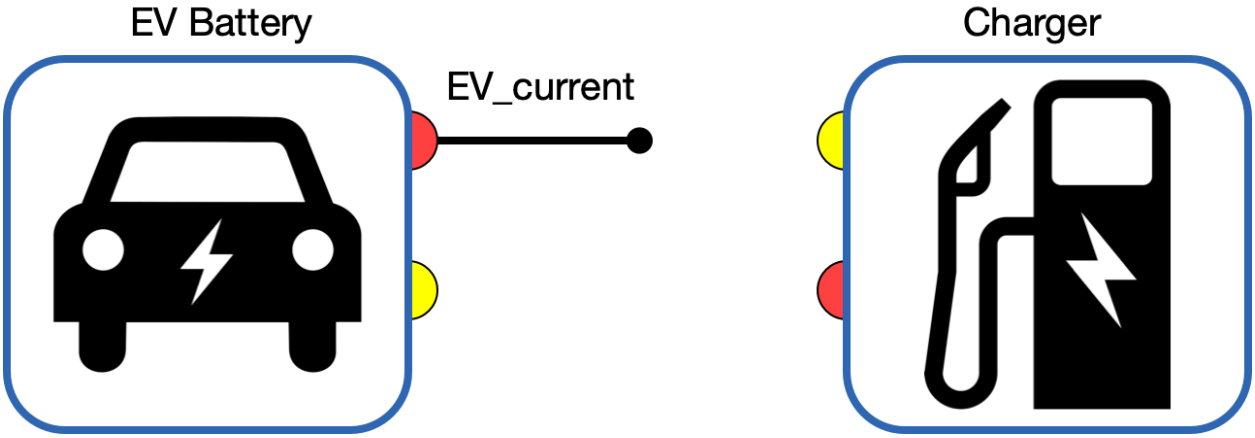
Messages and Communication: pub/sub

In the Base Example, the information being passed between the `Battery.py` federate and the `Charger.py` federate is the **voltage** applied to the battery, and the **current** measured across the battery and fed back to the charger. Voltage and current are both physical quantities, meaning that unless we act on these quantities to change them, they will retain their values. For this reason, in HELICS, physical quantities are called **values**. Values are sent via publication and subscription – a federate can publish its value(s), and another federate can subscribe this value(s).

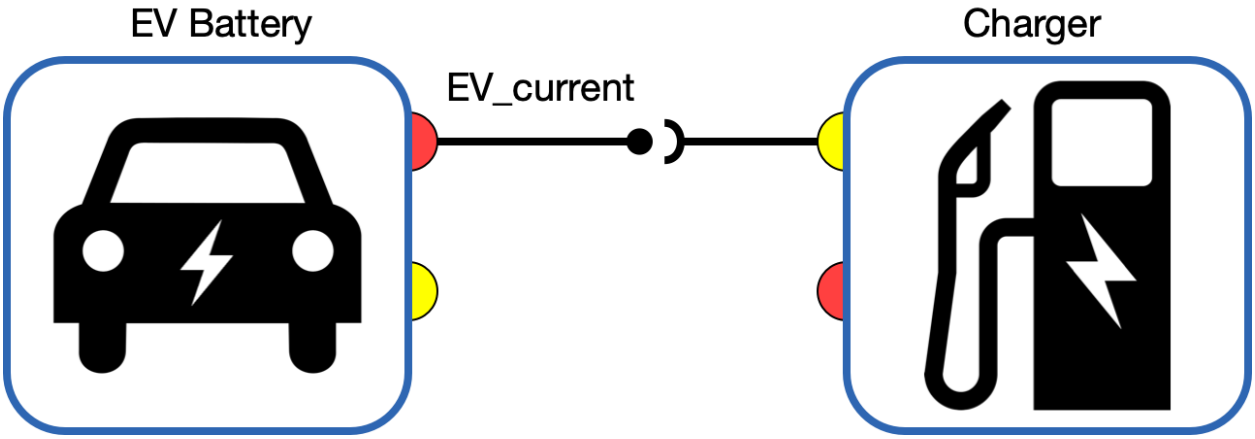
When configuring the communication passage between federates, it is important to connect the federate to the correct **interface**. In the image below, we have a Battery federate and a Charger federate. Each federate has a **publication** interface (red) and a **subscription** interface (yellow). The publication interface is also called the **output**, and the subscription interface the **input**. How are values passed between federates with pubs and subs?



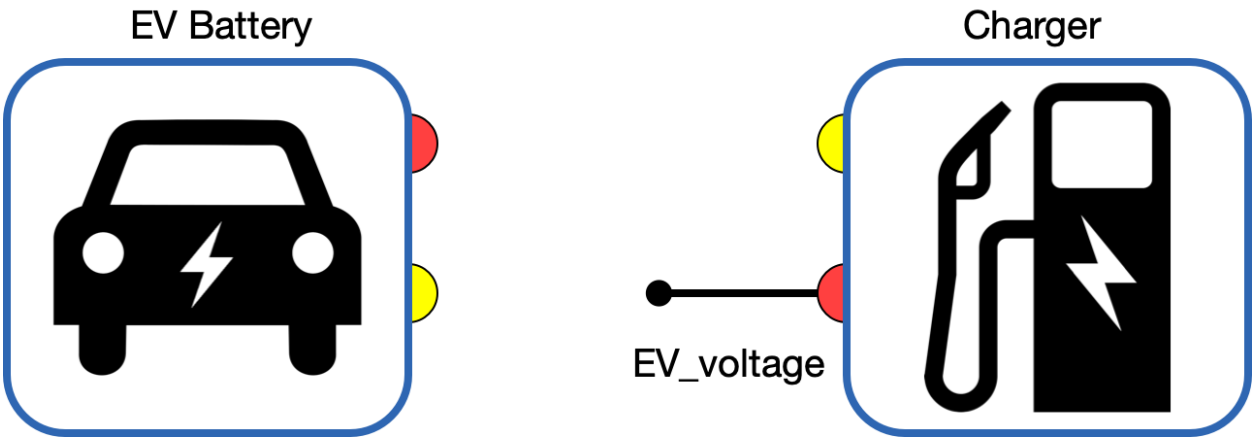
We have **named** the publication interface for the `Battery` federate `EV_current` to indicate the information being broadcast – we can also call the publication interface the **named output**. This is what the publication is doing – we are telling the Battery federate that we want to publish the EV_current. The full interface designation for the current is `Battery/EV_current` (within the JSON, this is also called the `key`).



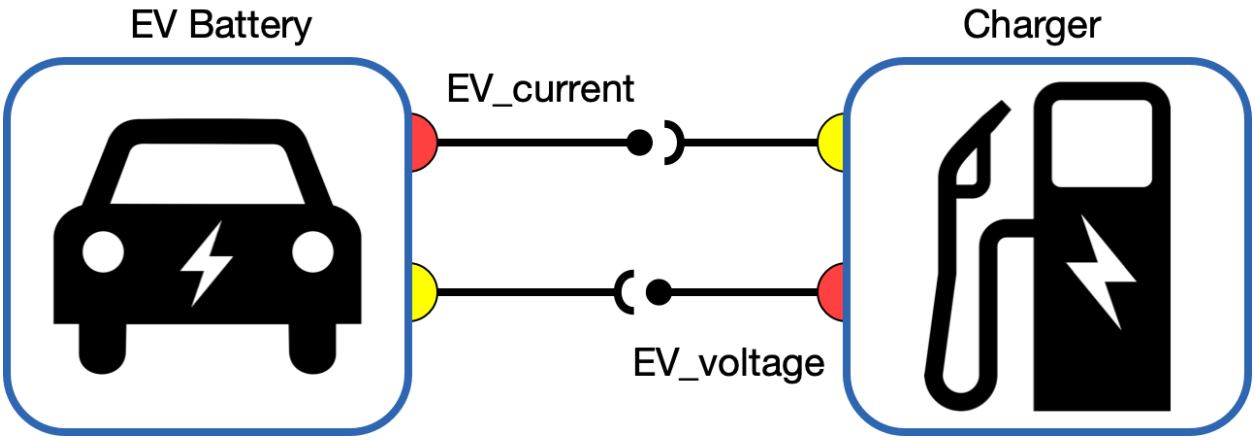
How does the current value get from the Battery federate's publication to the Charger federate? The Charger must subscribe to this publication interface – the Charger will subscribe to `Battery/EV_current`. The Charger subscription interface has not been given a name (e.g., `Charger/EV_current`), but it will receive **input** – the Charger subscription is a defined unnamed input with a targeted publication. In this example, we configure the target of the Charger subscription in the JSON to the publication interface name `Battery/EV_current`.



Thus far we have established that the Battery is publishing its current from the named interface `Battery/EV_current` and the Charger is subscribing to this named interface. The Charger is also sending information about values. The Charger federate will be publishing the voltage value from the `Charger/EV_voltage` interface (a named output).



In order to guarantee that the Battery federate receives the voltage value from the Charger federate, the Battery will have an unnamed input subscription which targets the `Charger/EV_voltage` interface.



With a better understanding of how we want to configure the pubs and subs, we can now move on to the mechanics of integrating the two simulators.

Simulator Integration: External JSON

Configuration of federates may be done with JSON files. Each federate will have its own configuration ("config") file. It's good practice to mirror the name of the federate with the config file. For example, the `Battery.py` federate will have a config file named `BatteryConfig.json`.

There are *extensive ways* to configure federates in HELICS. The `BatteryConfig.json` file contains the most common as defaults:

```
{
  "name": "Battery",
  "log_level": 1,
  "core_type": "zmq",
  "period": 60,
  "uninterruptible": false,
  "terminate_on_error": true,
```

(continues on next page)

(continued from previous page)

```
  "wait_for_current_time_update": true,
  "publications": [],
  "subscriptions": []
}
```

In this configuration, we have named the federate `Battery`, set the `log_level` to 1 (*what do loglevels mean and which one do I want?*), and set the `core_type` to `zmq` (*the most common*). The next four options control timing for this federate. The final options are for message passing.

This federate is configured with pubs and subs, so it will need an option to indicate the publication and the subscription configurations (for brevity, only the first pub and sub are printed below):

```
"publications":[
  {
    "key":"Battery/EV1_current",
    "type":"double",
    "unit":"A",
    "global": true
  },
],
"subscriptions":[
  {
    "key":"Charger/EV1_voltage",
    "type":"double",
    "unit":"V",
    "global": true
  },
]
```

This pub and sub configuration is telling us that the `Battery.py` federate is publishing in units of amps (`A`) for current from the named interface (`key`) `Battery/EV1_current`. This federate is also subscribing to information from the `Charger.py` federate. It has subscribed to a value in units of volts (`V`) at the named interface (`key`) `Charger/EV1_voltage`.

As discussed in *"Simulator Integration: External JSON""*, the federate registration and configuration with JSON files in the python federate is done with one line of code:

```
fed = h.helicsCreateValueFederateFromConfig("BatteryConfig.json")
```

Recall that federate registration and configuration is typically done **before** entering execution mode.

Co-simulation Execution

At this point in setting up the Base Example co-simulation, we have:

1. Placed the necessary HELICS components in each federate program
2. Written the configuration JSON files for each federate

It's now time to launch the co-simulation with the HELICS runner. This is accomplished by creating a **runner** JSON file. The HELICS runner allows the user to launch multiple simulations in one command line, which otherwise would have required multiple terminals.

The runner JSON for the Base Example is called `fundamental_default_runner.json`:

```
{
  "name": "fundamental_default",
  "broker": true,
  "federates": [
    {
      "directory": ".",
      "exec": "python -u Charger.py 1",
      "host": "localhost",
      "name": "Charger"
    },
    {
      "directory": ".",
      "exec": "python -u Battery.py 1",
      "host": "localhost",
      "name": "Battery"
    }
  ]
}
```

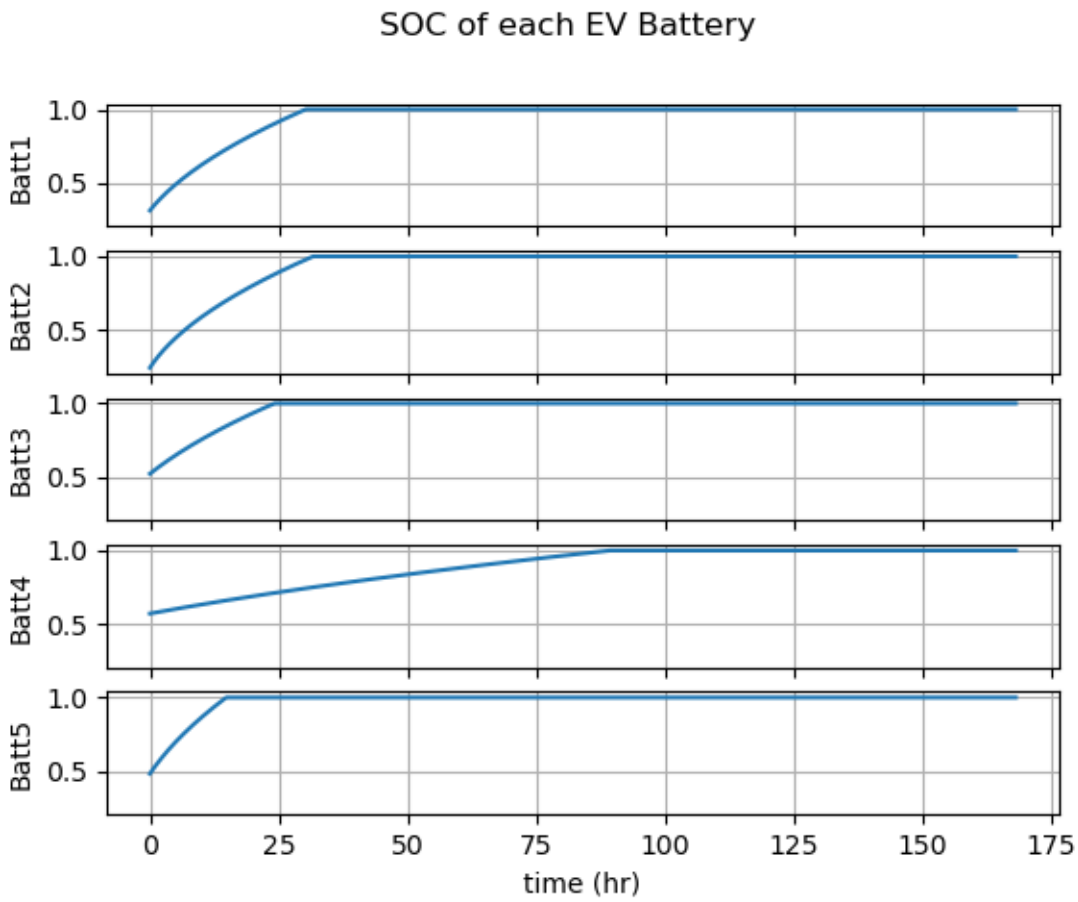
This runner tells HELICS to launch a broker and two federates:

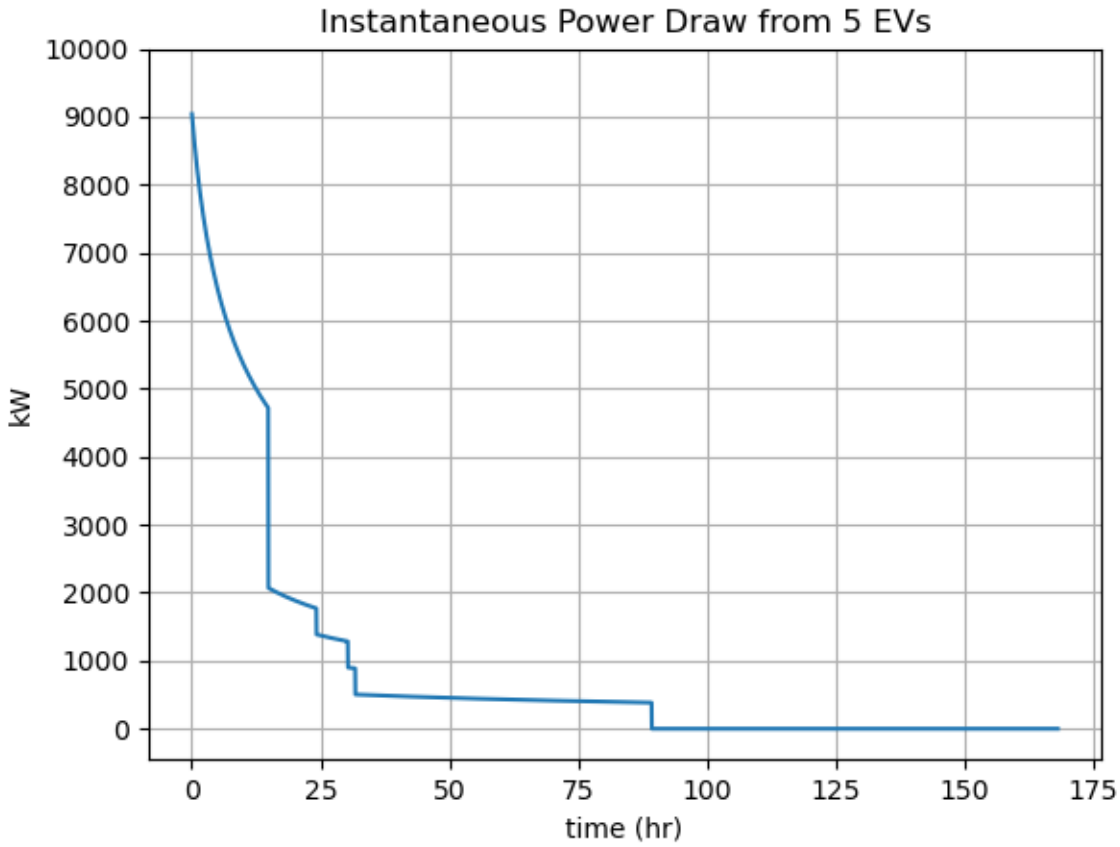
1. Launch a broker for the co-simulation.
2. Launch the `Charger.py` federate in the current directory: `python -u Charger.py 1`
3. Launch the `Battery.py` federate in the current directory: `python -u Battery.py 1`

The final step is to launch our Base Example with the HELICS runner from the command line (making sure you've *installed the HELICS cli extension*):

```
helics run --path=fundamental_default_runner.json
```

If all goes well, this will reward us with two figures:


SOC of each EV Battery
Batt1
Batt2
Batt3
Batt4
Batt5
1.0
0.5
0
25
50
75
100
125
150
175
time (hr)



We can see the state of charge of each battery over the duration of the co-simulation in the first figure, and the aggregated instantaneous power draw in the second. As the engineer tasked with assessing the power needs for this charging garage, do you think you have enough information at this stage? If not, how would you change the co-simulation to better model the research needs?

Questions and Help

Do you have questions about HELICS or need help?

1. Come to office hours!
2. Post on the gitter!
3. Place your question on the github forum!

Federate Message + Communication Configuration

Endpoint Federates

The Federate Message + Communication Configuration Example extends the Base Example to demonstrate how to register federates which send messages from/to endpoints instead of values from/to pub/subs.

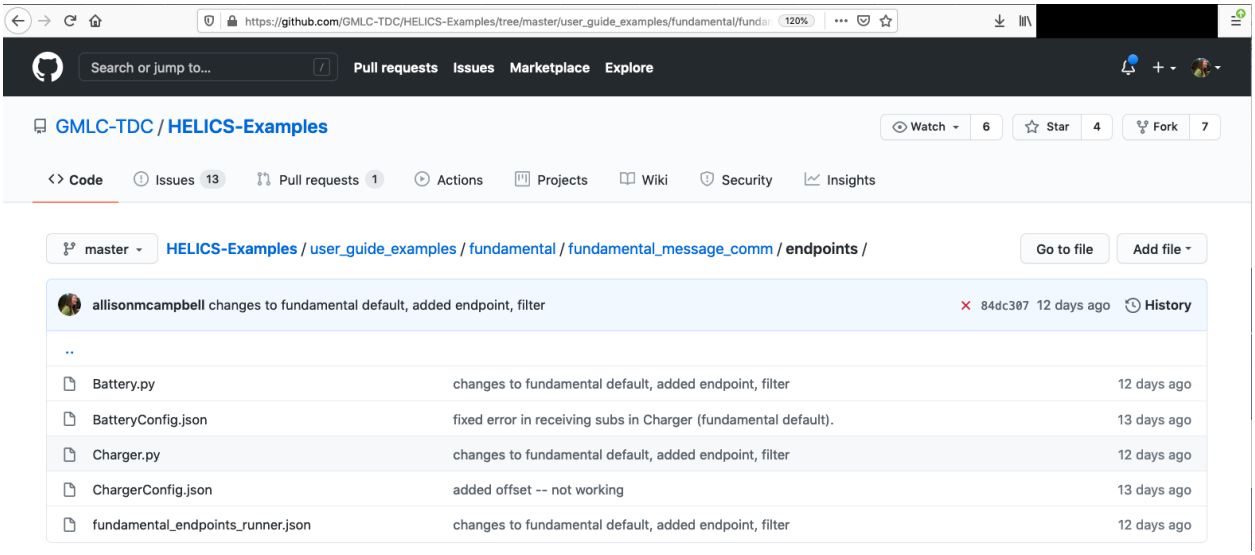
This tutorial is organized as follows:

- *Endpoint Federates*
  - *Example files*

- *Federate Communication with Endpoints*
  - *When to use pub/subs vs endpoints*
  - *Translation from pub/sub to endpoints*
    - *Config Files*
    - *Simulators*
    - *Battery*
    - *Charger*
  - *Co-simulation execution*
- *Questions and Help*

Example files

All files necessary to run the Federate Integration Example can be found in the Fundamental examples repository:



The files include:

- Python program and configuration JSON for Battery federate
- Python program and configuration JSON for Charger federate
- HELICS runner JSON to enable execution of the co-simulation

Federate Communication with Endpoints

There are two fundamental cases where you may find yourself using endpoints to send messages.

1. The federate is modeling communication of information (typically as a string)
2. The federate is incorrectly modeling communication of physical dynamics

What's the difference? In the *Base Example*, the federation consists of two "value" federates – one passes its current, the other passes a voltage. The two depend on this information. The Battery says to the Charging port, "I have a starting current!", to which the Charger responds, "Great, here's your voltage!" These two **value** federates must be coupled with pubs/subs, because they are linked by a physical system.

However, as the author of a HELICS co-simulation, there is nothing preventing you from connecting these two federates with endpoints in place of pub/subs. The co-simulation will give the same results in simple cases, but be wary of taking this type of short cut – resurrecting code which passes information in the incorrect way may introduce nefarious results. The recommended approach is to register federates modeling **physics** as **value federates**, and those modeling *information* as *message federates*.

Casting this guidance to the wind, this example walks through how to set up the Base Example (which passes current and voltage) as **message federates** – landing this example squarely in situation #2 above. This is just for demonstration purposes, and this is the only example in the documentation which violates best practices.

When to use pub/subs vs endpoints

The easiest way to determine whether you should use pub/subs vs endpoints to connect your federates is to ask yourself: "Does this message have a physical unit of measurement?" As noted above, the Battery-Charger federation **does** model physical components, and the config files make this clear with the inclusion of units:

`BatteryConfig.json`:

```
"publications":[
  {
    "key":"Battery/EV1_current",
    "type":"double",
    "unit":"A",
    "global": true
  },
]
```

```
"subscriptions":[
  {
    "key":"Charger/EV1_voltage",
    "type":"double",
    "unit":"V",
    "global": true
  },
]
```

`ChargerConfig.json`:

```
"publications":[
  {
    "key":"Charger/EV1_voltage",
    "type":"double",
    "unit":"V",
    "global": true
  },
]
```

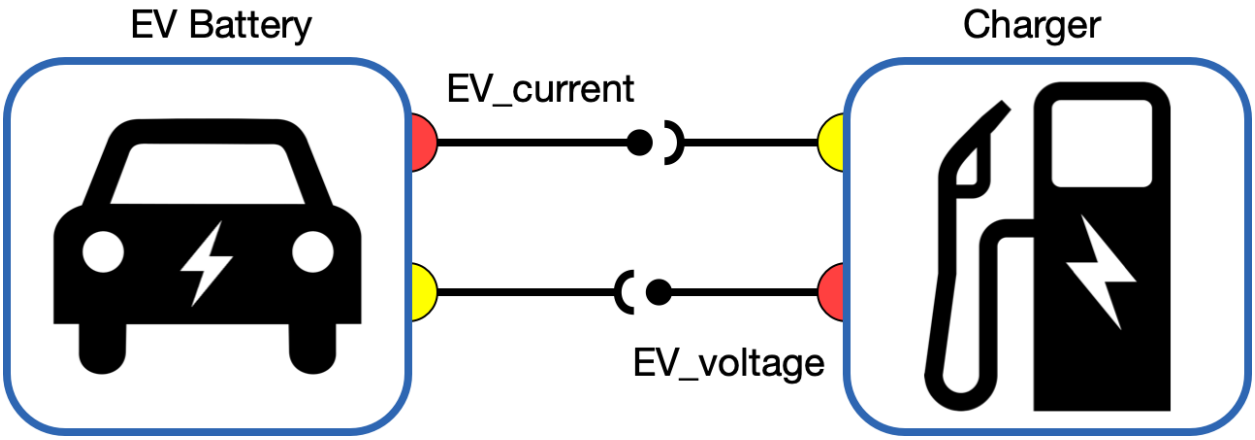
```
"subscriptions":[
  {
    "key":"Battery/EV1_current",
    "type":"double",
    "unit":"A",
    "global": true
```

(continues on next page)

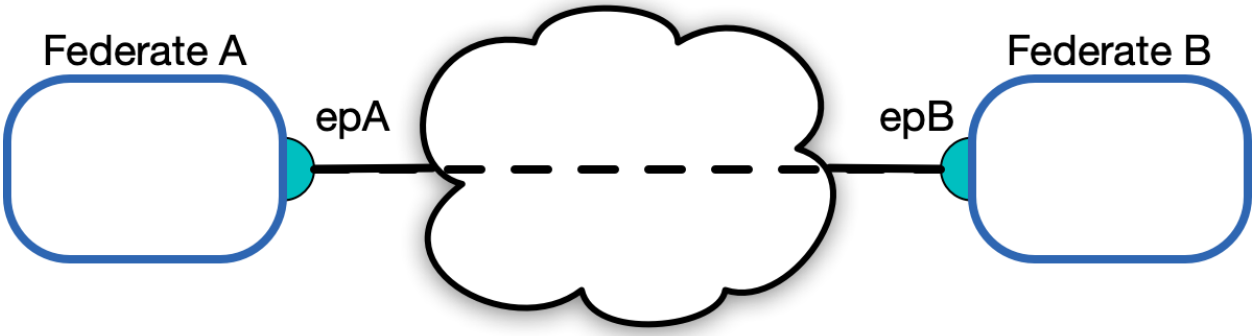
(continued from previous page)

```
  },
]
```

With this pub/sub configuration, we have established a **direct** communication link between the Battery and Charger:



If we accept that the information being passed between the two is not physics-based, then the communication link only depends on each federate having an endpoint:



In departure from the directly-coupled communication links of pub/subs, messages sent from **endpoints** can be intercepted, delayed, or picked up by any federate. In that sense, communication via pub/subs can be thought of as sealed letters sent via pneumatic tubes, and messages sent via endpoints as a return-address labeled letter sent into the postal service system.

You might ask yourself: "How does HELICS know where to send the message?" There are ways to set this up as default. Before we dive into the code, it's important to understand the following about messages and endpoints:

1. Endpoints send messages as encoded strings
2. Endpoints can have default destinations, but this is not required
3. Endpoints should not be used to model physics
4. Messages sent from endpoints are allowed to be corrupted (see *Filters*!)
5. Messages sent from endpoints do not show up on a HELICS `dependency_graph` (A `dependency_graph` is a graph of dependencies in a federation. Because pub/subs have explicit connections, HELICS can establish when the information arrives through a `dependency_graph`. See *Queries* for more information.)

Translation from pub/sub to endpoints

We are throwing caution to the wind using endpoints to model the physics in the *Base Example*. Changes need to be made in the config files and the simulator code.

Config Files

As with the Base Example, configuration can be done with JSON files. The first change we need to make is to replace the `publications` and `subscriptions` with `endpoints`:

`BatteryConfig.json`:

```
"endpoints":[
  {
    "name":"Battery/EV1_current",
    "destination":"Charger/EV1_voltage",
    "global": true
  },
]
```

`ChargerConfig.json`:

```
"endpoints":[
  {
    "name":"Charger/EV1_voltage",
    "destination":"Battery/EV1_current",
    "global": true
  },
]
```

If you have run the Base Example, you will have seen that the information passed between the federates occurs at the same HELICS time; both federates have `"period": 60` in their config files. Recall from the *Configuration Options Reference* that the `period` controls the minimum time resolution for a federate.

We have a federation sending messages at the same time (`"period": 60`), and HELICS doesn't know which message arrives first. We need to introduce an `offset` to the config file of one of the federates to force it to wait until the message has been received. We also need to keep `"uninterruptible": false`, so that the federate will be granted the time at which it has received a message (which will be `"period": 60`).

The order of operation is:

Step	HELICS Time	Charger	Battery
1	0	requests time = 60, granted time = 60	requests time = 70
2	60	sends message to default destination `"Battery/EV1_current"`	granted time = 60 because message has arrived
3	60		sends message to default destination `"Charger/EV1_voltage"`

Introducing the `offset` into the config file (along with `"uninterruptible": false`) instructs HELICS about the dependencies. These adjustments are:

`BatteryConfig.json`:

```
{
  "name": "Battery",
  "loglevel": 7,
  "coreType": "zmq",
  "period": 60.0,
  "offset": 10.0,
  "uninterruptible": false,
  "terminate_on_error": true,
  "endpoints": []
}
```

`ChargerConfig.json`:

```
{
  "name": "Charger",
  "loglevel": 7,
  "coreType": "zmq",
  "period": 60,
  "uninterruptible": false,
  "terminate_on_error": true,
  "endpoints": []
}
```

Notice that we have only introduced an `offset` into the Battery config file, as we have set up the federates such that the Battery is waiting for information from the Charger.

Simulators

The simulators in this co-simulation (`*.py`) must be edited from the Base Example to register endpoints (instead of pub/subs) and ensure that messages are sent and received.

Battery

First let's make changes to `Battery.py`. In the Registration Step, we need to register the endpoints and get the endpoint count:

```
fed = h.helicsCreateMessageFederateFromConfig("BatteryConfig.json")
federate_name = h.helicsFederateGetName(fed)
logger.info(f"Created federate {federate_name}")
print(f"Created federate {federate_name}")

end_count = h.helicsFederateGetEndpointCount(fed)
logging.debug(f"\tNumber of endpoints: {end_count}")

# Diagnostics to confirm JSON config correctly added the required
#   endpoints
endid = {}
for i in range(0, end_count):
    endid[i] = h.helicsFederateGetEndpointByIndex(fed, i)
    end_name = h.helicsEndpointGetName(endid[i])
    logger.debug(f"\tRegistered Endpoint ---> {end_name}")
```

After entering Execution Mode but before the Time Loop begins, we need to extract the offset value:

```
update_offset = int(h.helicsFederateGetTimeProperty(fed, h.helics_property_time_offset))
```

And add that offset to `requested_time`:

```
requested_time = grantedtime + update_interval + update_offset
```

The next largest difference with implementing communication between simulators with endpoints vs pub/subs comes from the lack of innate message dependency, as described above with the `dependency_graph`. (Which can be accessed for pub/subs with a *query*.) Best practice for handling message receipt is to check if a message is waiting to be retrieved for an endpoint. The following code replaces `charging_voltage = h.helicsInputGetDouble((subid[j]))` from the Base Example (we are looping over `end_count`, the number of endpoints for this federate):

```
endpoint_name = h.helicsEndpointGetName(endid[j])
if h.helicsEndpointHasMessage(endid[j]):
    msg = h.helicsEndpointGetMessage(endid[j])
    charging_voltage = float(h.helicsMessageGetString(msg))
```

If we want to know who sent the message (which can be helpful for both debugging and simplifying code), we invoke:

```
source = h.helicsMessageGetOriginalSource(msg)
```

An alternative to using `h.helicsMessageGetOriginalSource(msg)` is to set a default `destination` in the JSON config file. Use of both can help with debugging.

The `Battery.py` simulator takes the `charging_voltage` from the `Charger.py` simulator and calculates the `charging_current` to send back. Sending messages to a default `destination` is then done with:

```
h.helicsEndpointSendBytesTo(endid[j], "", str(charging_current))
```

Where the "" can also be replaced with a string for the desired destination – we can check "" against `source` to confirm the messages are going to their intended destinations.

Charger

As with the `Battery.py` simulator, we need to Register the Charger federate as a Message Federate and get the endpoint ids:

```
fed = h.helicsCreateMessageFederateFromConfig("ChargerConfig.json")
federate_name = h.helicsFederateGetName(fed)
logger.info(f"Created federate {federate_name}")
print(f"Created federate {federate_name}")
end_count = h.helicsFederateGetEndpointCount(fed)
logging.debug(f"\tNumber of endpoints: {end_count}")

# Diagnostics to confirm JSON config correctly added the required
#   endpoints
endid = {}
for i in range(0, end_count):
    endid[i] = h.helicsFederateGetEndpointByIndex(fed, i)
    end_name = h.helicsEndpointGetName(endid[i])
    logger.debug(f"\tRegistered Endpoint ---> {end_name}")
```

The next difference with the Base Example `Charger.py` simulator is in sending the initial voltage to Battery Federate:

```
for j in range(0, end_count):
    message = str(charging_voltage[j])
    h.helicsEndpointSendBytesTo(endid[j], "", message.encode())  #
```

Notice that we are sending the message to the default `destination` with "". We cannot use `h.helicsMessageGetOriginalSource(msg)`, as no messages have been received by the Charger Federate yet.

Within the Time Loop, we change the message receipt component in the same way as the `Battery.py` simulator, where `h.helicsInputGetDouble((subid[j]))` is replaced with:

```
endpoint_name = h.helicsEndpointGetName(endid[j])
if h.helicsEndpointHasMessage(endid[j]):
    msg = h.helicsEndpointGetMessage(endid[j])
    charging_current[j] = float(h.helicsMessageGetString(msg))
```

There's one final difference. Which API do we call to send the message to the Battery Federate?

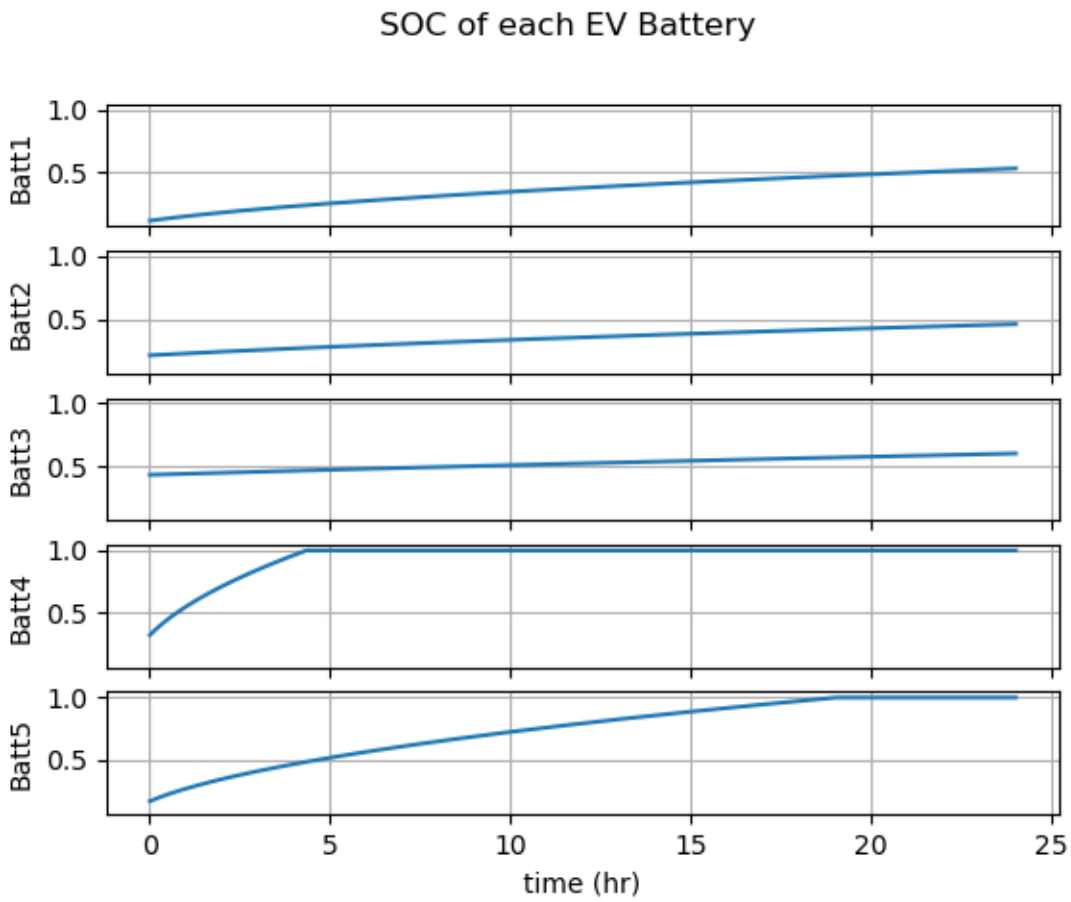
```
# Send message of voltage to Battery federate
h.helicsEndpointSendBytesTo(endid[j], "",f'{charging_voltage[j]:4f}'.encode())  #
```

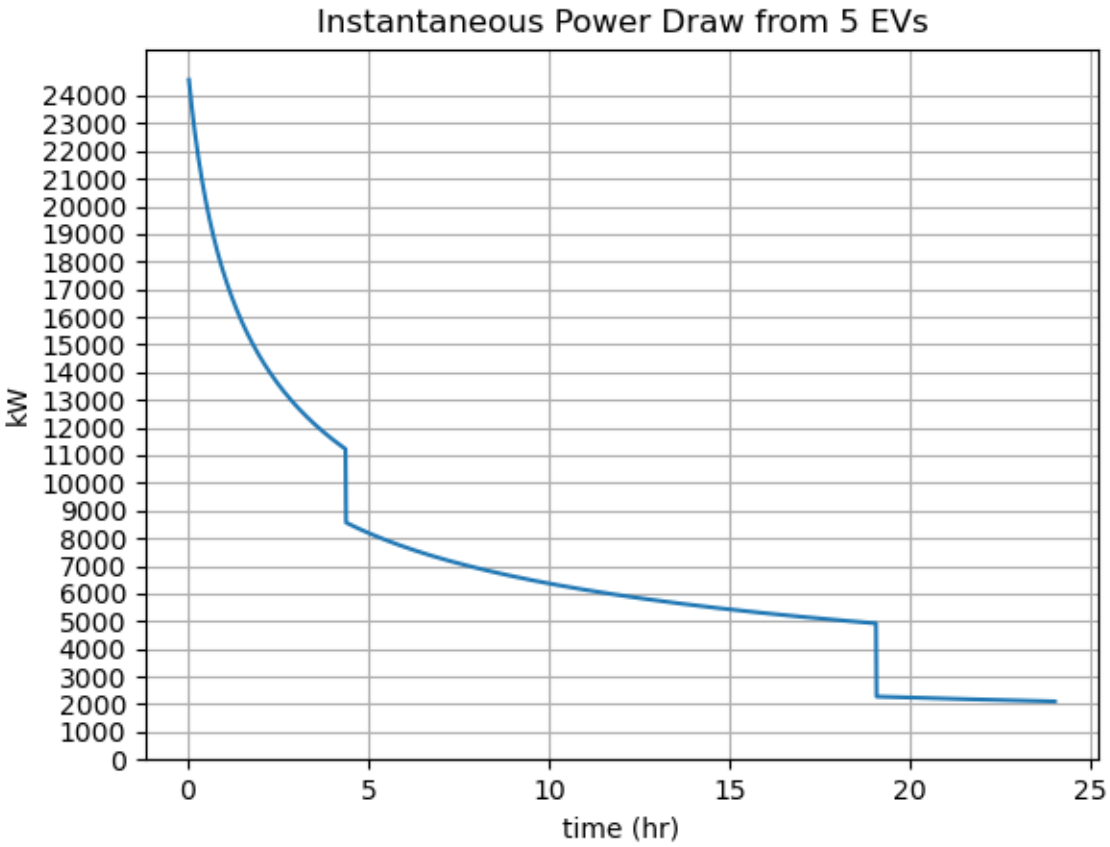
Co-simulation execution

We run the co-simulation just as before in the Base Example – the `runner.json` is exactly the same:

```
helics run --path=fundamental_endpoints_runner.json
```

And we get these figures:


SOC of each EV Battery
Batt1
Batt2
Batt3
Batt4
Batt5
1.0
0.5
0
5
10
15
20
25
time (hr)



Armed now with the knowledge of endpoints and messages, how could you change the research question?

Questions and Help

Do you have questions about HELICS or need help?

1. Come to office hours!
2. Post on the gitter!
3. Place your question on the github forum!

Combination Federation

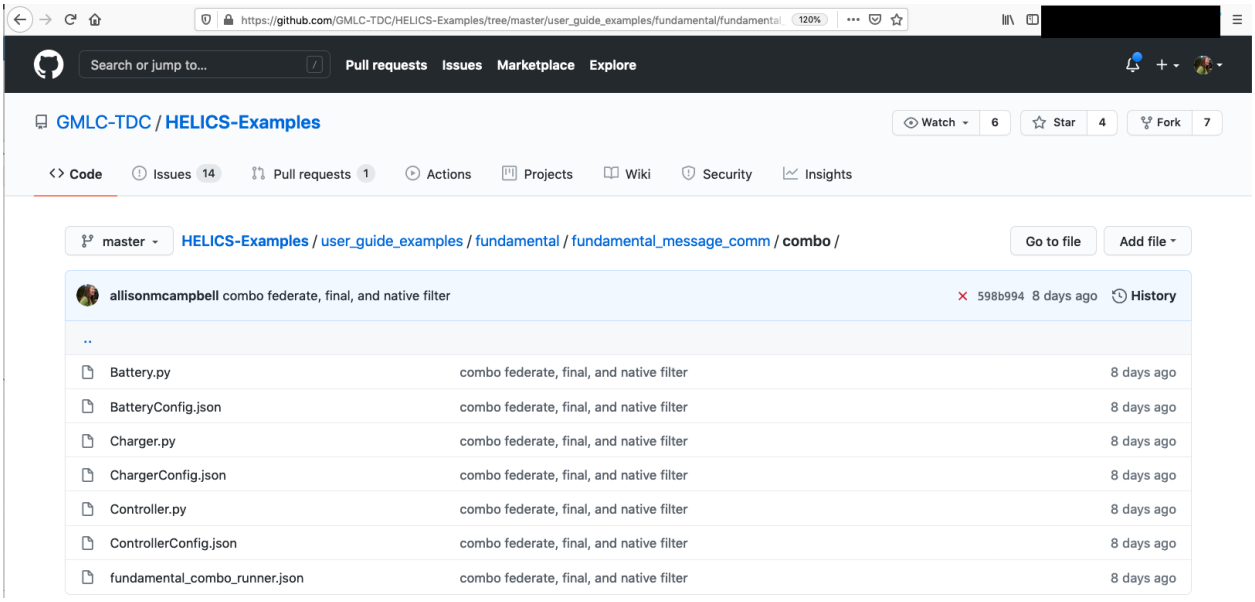
The Federate Message + Communication Configuration Example extends the Base Example to demonstrate how to register federates which can send/receive messages from endpoints and values from pub/subs. This example assumes the user has already worked through the *Endpoints Example*.

This tutorial is organized as follows:

- *Example files*
- *Combination Federates*
  - *Co-simulation Execution*
  - *Alternative implementation: Multi-Agent*
- *Questions and Help*

Example files

All files necessary to run the Federate Integration Example can be found in the Fundamental examples repository:



- Python program and configuration JSON for Battery federate
- Python program and configuration JSON for Charger federate
- Python program and configuration JSON for Controller federate
- HELICS runner JSON to enable execution of the co-simulation

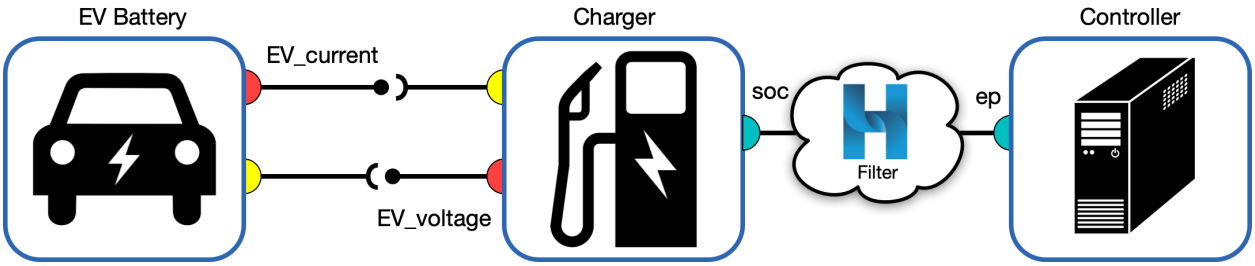
Combination Federates

A quick glance at the Fundamental examples repository on github will show that almost all these introductory examples are mocked up with two federates. These two federates pass information back and forth, and the examples show different ways this can be done.

This is the only example in the Fundamental series which models three federates – it is also exactly the same model as the *Base Example* in the Advanced series. Why are we introducing a third federate?

In the *Endpoints Example*, we learned how to pass messages between two federates. The problem with this setup – which we will resolve in this example – is that **physical values** should not be modeled with messages/endpoints (see *the example* for a reminder). We introduce a third federate – a **combination federate** – to preserve the handling of physical values among *value federates* and allow for nuanced message passing (and interruption) among *message federates*. The key with combo federates is that they are the go-between for these two types. Combination federates can update (send) values *and* intercept messages. (For a refresher on values and messages, see the section on *Types of Federates*. In brief: values have a physics-based unit, and messages are typically strings).

Here is our new federation of three federates:



We have:

- Battery (**value federate**: passes values with Charger through pub/subs)
- Charger (**combo federate**: passes values with Battery, passes messages with Controller)
- Controller (**message federate**: passes messages with Charger through endpoints)

Redistribution of Federate Roles

The full co-simulation is still asking the same question: "What is the expected instantaneous power draw from a dedicated EV charging garage?" With the introduction of a *Controller* federate, we now have additional flexibility in addressing the nuances to this question. For example, the charging controller does not have direct knowledge of the instantaneous current in the battery – the onboard charger needs to estimate this in order to calculate the EV's state of charge. Let's walk through the roles of each federate.

Battery

The Battery federate operates in the same way as in the Base Example. The only difference is that it is now allowed to request a new battery when an existing one is deemed to have a full SOC. This information is in the `charging_voltage` value from the Battery's subscription to the Charger; if the Charger applies zero voltage, this means the Battery can no longer charge. The Battery federate selects a new battery randomly from three sizes – small, medium, and large – and assigns a random SOC between 0% and 80%.

There are no differences in the config file. As in the Base Example, the Battery federate logs and plots the internally calculated SOC over time at each charging port.

Charger

The Charger federate is now a combination federate – it will communicate via pub/subs with the Battery, and via endpoints with the Controller. This difference from the Base Example Charger is seen in the config file; in addition to the pub/subs with the Battery, there are now also endpoints. Notice that the default destination for each of these named endpoints is the same – there is one controller for all the charging ports.

```
  "endpoints": [
    {
      "name": "Charger/EV1.soc",
      "destination": "Controller/ep",
      "global": true
    },
```

Since this federate also communicated via endpoints, we need to register them along with the existing pub/subs:

```
############  Registering  federate from json  ##########################
fed = h.helicsCreateCombinationFederateFromConfig("ChargerConfig.json")
federate_name = h.helicsFederateGetName(fed)
logger.info(f"Created federate {federate_name}")
```

(continues on next page)

(continued from previous page)

```
end_count = h.helicsFederateGetEndpointCount(fed)
logger.info(f"\tNumber of endpoints: {end_count}")
sub_count = h.helicsFederateGetInputCount(fed)
logger.info(f"\tNumber of subscriptions: {sub_count}")
pub_count = h.helicsFederateGetPublicationCount(fed)
logger.info(f"\tNumber of publications: {pub_count}")
```

The Charger federate is gaining the new role of *estimating the Battery's current* and shifting the role of *deciding when to stop charging* to the Controller federate.

The Charger federate estimates the Battery federate's current with a new helper function call `estimate_SOC`. The Charger does not know the exact SOC of the Battery; it must estimate the SOC from the effective resistance, which is a function of applied voltage (from the Charger) and the measured current (from the Battery). This is the same function as used in the Battery federate, but with noise added to the measurement of the current.

```
def estimate_SOC(charging_V, charging_A):
    socs = np.array([0, 1])
    effective_R = np.array([8, 150])
    mu = 0
    sigma = 0.2
    noise = np.random.normal(mu, sigma)
    measured_A = charging_A + noise
    measured_R = charging_V / measured_A
    SOC_estimate = np.interp(measured_R, effective_R, socs)

    return SOC_estimate
```

This function is called after the Charger has received the charging current from the Battery federate and needs to update the SOC; if the current is not zero, the Charger estimates the SOC with the inclusion of measurement error on the current. This allows the co-simulation to model the separation of knowledge of the physics between the two federates: the Battery knows its internal current, but the on board Charger must estimate it.

If the current received from the Battery federate is zero, this means that we have plugged a new EV into the charging port and we need to determine the voltage to apply with the Charger. This is accomplished by calling `get_new_EV(1)` and `calc_charging_voltage()`. `get_new_EV(1)` is a helper function which selects the charging level (1, 2, or 3) based on a set probability distribution and `calc_charging_voltage()` gives the applied voltage for that level. Once a "new EV" (the charging level) has been retrieved, the federate is assigned a SOC of 0 as an initial estimate prior to measuring the current.

The estimated SOC is sent to the Controller every 15 minutes – this mimics an on board charging agent regularly pinging the charging port to confirm if it should continue charging:

```
# Send message to Controller with SOC every 15 minutes
if grantedtime % 900 == 0:
    h.helicsEndpointSendBytesTo(endid[j], "", f"{currentsoc[j]:4f}".encode())
```

The Charger federate is allowed to be interrupted if there is a message from the Controller.

```
# Check for messages from EV Controller
endpoint_name = h.helicsEndpointGetName(endid[j])
if h.helicsEndpointHasMessage(endid[j]):
    msg = h.helicsEndpointGetMessage(endid[j])
    instructions = h.helicsMessageGetString(msg)
```

The Charger will receive a message every 15 minutes as well, however it will only change actions if it is told to stop charging. When this happens, the Charger "disengages" from the charging port by applying zero voltage to the Battery.

```
if int(instructions) == 0:
    # Stop charging this EV
    charging_voltage[j] = 0
    logger.info(f"\tEV full; removing charging voltage")
```

Controller

The Controller is a new federate whose role is to decide whether to keep charging an EV based. This decision is based entirely on the estimated SOC calculated by the Charger. Since this decision logic is simple and can be applied to all the EVs modeled by the federation, we can set up the config file with one endpoint:

```
  "endpoints": [
    {
      "name": "Controller/ep",
      "global": true
    }
  ]
```

Note that there is no default destination – the Controller will *respond* to a request for instructions from the Charger. This is accomplished by calling the `h.helicsMessageGetOriginalSource()` API:

```
while h.helicsEndpointHasMessage(endid):

    # Get the SOC from the EV/charging terminal in question
    msg = h.helicsEndpointGetMessage(endid)
    currentsoc = h.helicsMessageGetString(msg)
    source = h.helicsMessageGetOriginalSource(msg)
```

And then sending the message to this source:

```
message = str(instructions)
h.helicsEndpointSendBytesTo(endid, source, message.encode())
```

The Controller federate only operates when it receives a message – it is a *passive* federate. This can be set up by:

1. Initializing the start time of the federate to `h.HELICS_TIME_MAXTIME`:

   ```
   fake_max_time = int(h.HELICS_TIME_MAXTIME)
   starttime = fake_max_time
   logger.debug(f"Requesting initial time {starttime}")
   grantedtime = h.helicsFederateRequestTime(fed, starttime)
   ```

2. Allow the federate to be interrupted and set a minimum `timedelta` (`ControllerConfig.json`):

   ```
   {
     "name": "Controller",
     "timedelta": 1,
     "uninterruptible": false
   }
   ```

3. Only execute an action when there is a message:

```
while h.helicsEndpointHasMessage(endid):
    pass  # placeholder for loop body
```

4. Re-request the `h.HELICS_TIME_MAXTIME` after a message has been received:

```
grantedtime = h.helicsFederateRequestTime(fed, fake_max_time)
```

The message the Controller receives is the SOC estimated by the Charger. If the estimated SOC is greater than 95%, the Controller sends the message back to stop charging.

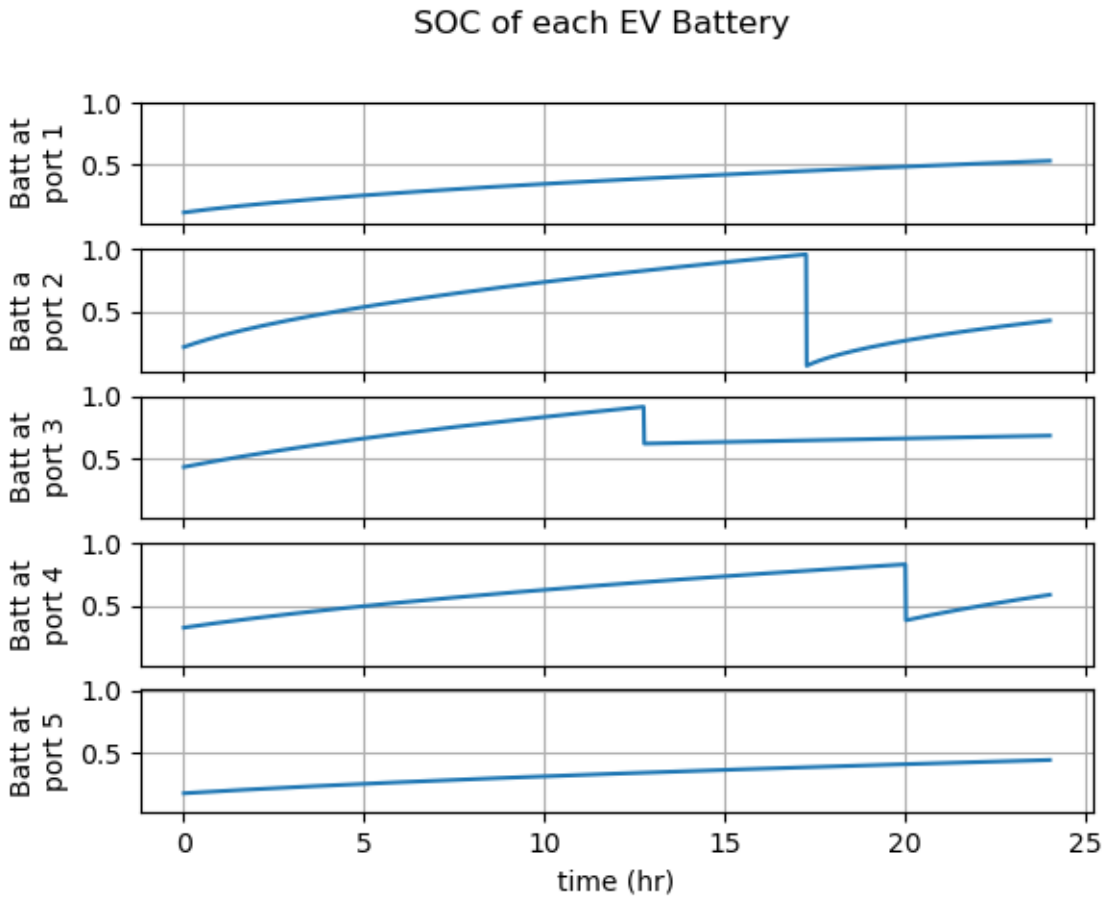
```
soc_full = 0.95
if float(currentsoc) <= soc_full:
    instructions = 1
else:
    instructions = 0
```

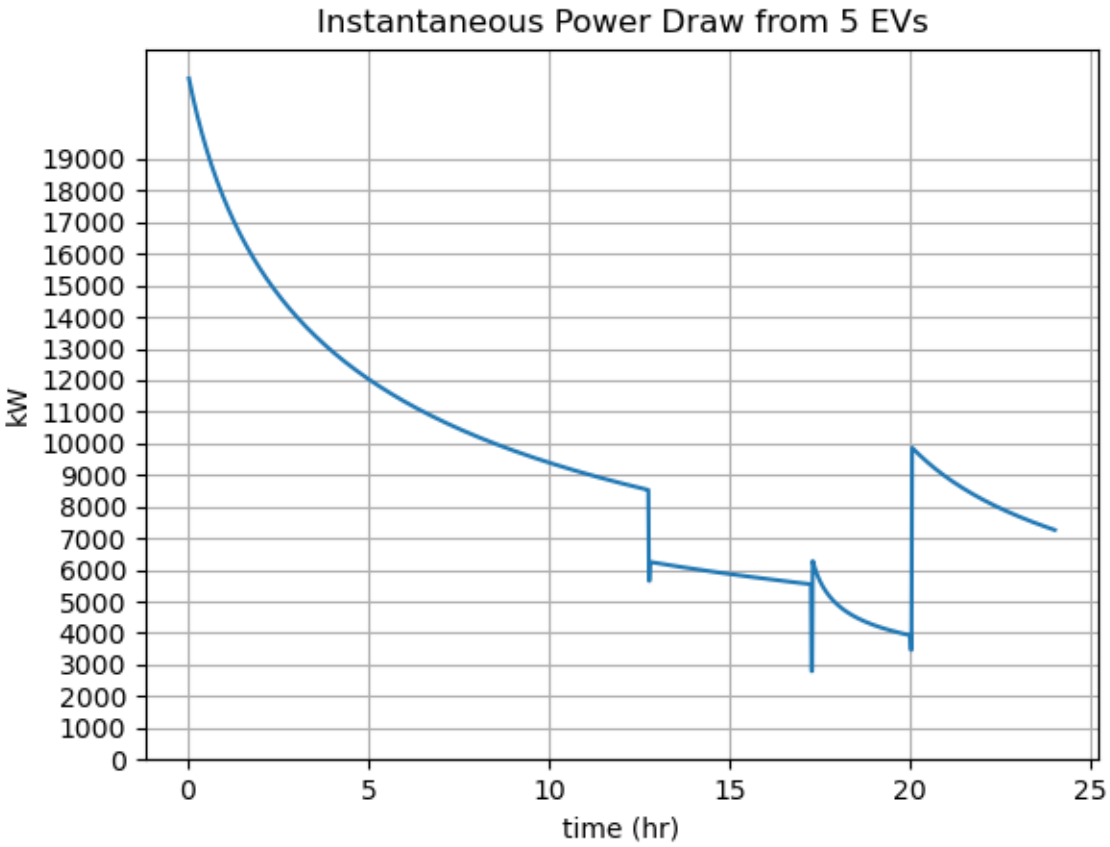
Co-simulation execution

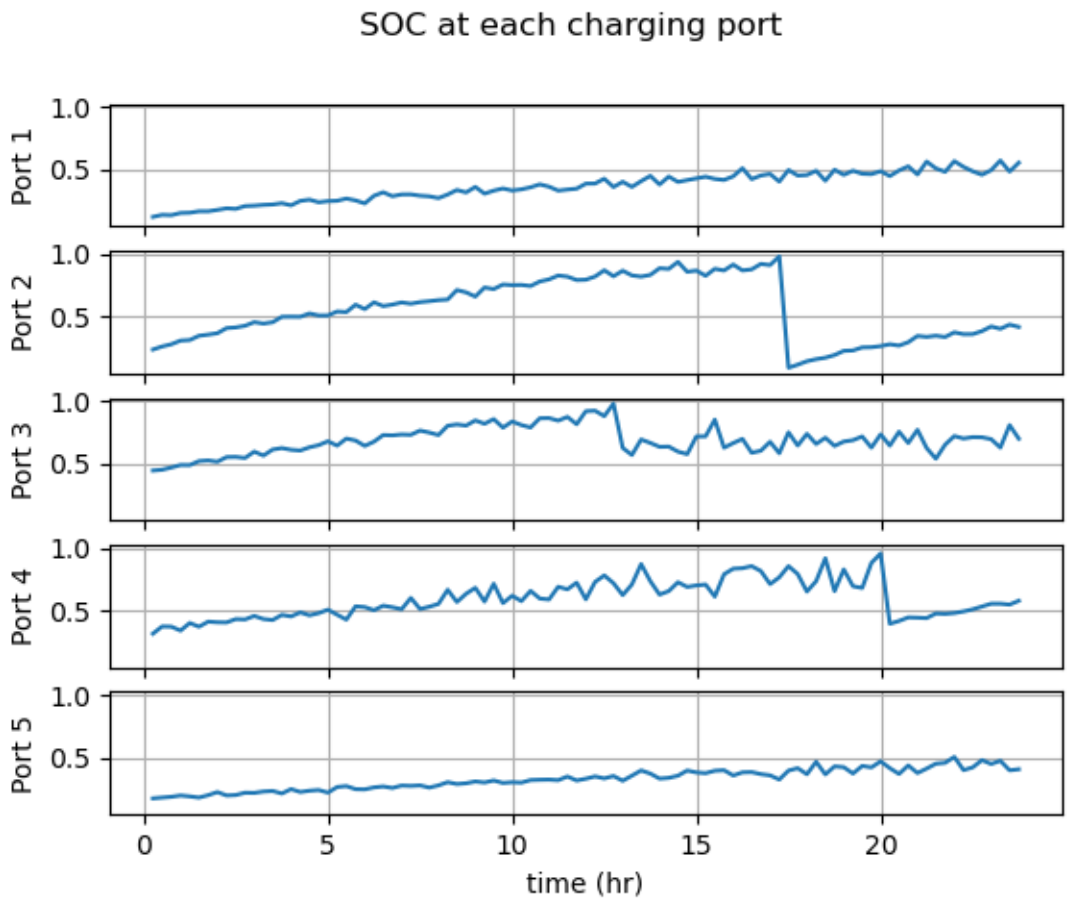
With these three federates – Battery, Charger, and Controller – we have partitioned the roles into the most logical places. Execution of this co-simulation is done as before with the HELICS runner:

```
helics run --path=fundamental_combo_runner.json
```

The resulting figures show the actual on board SOC at each EV charging port, the instantaneous power draw, and the SOC estimated by the on board charger.


SOC of each EV Battery
Batt at port 1
Batt a port 2
Batt at port 3
Batt at port 4
Batt at port 5
1.0
0.5
0
5
10
15
20
25
time (hr)


Instantaneous Power Draw from 5 EVs
kW
19000
18000
17000
16000
15000
14000
13000
12000
11000
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0
5
10
15
20
25
time (hr)



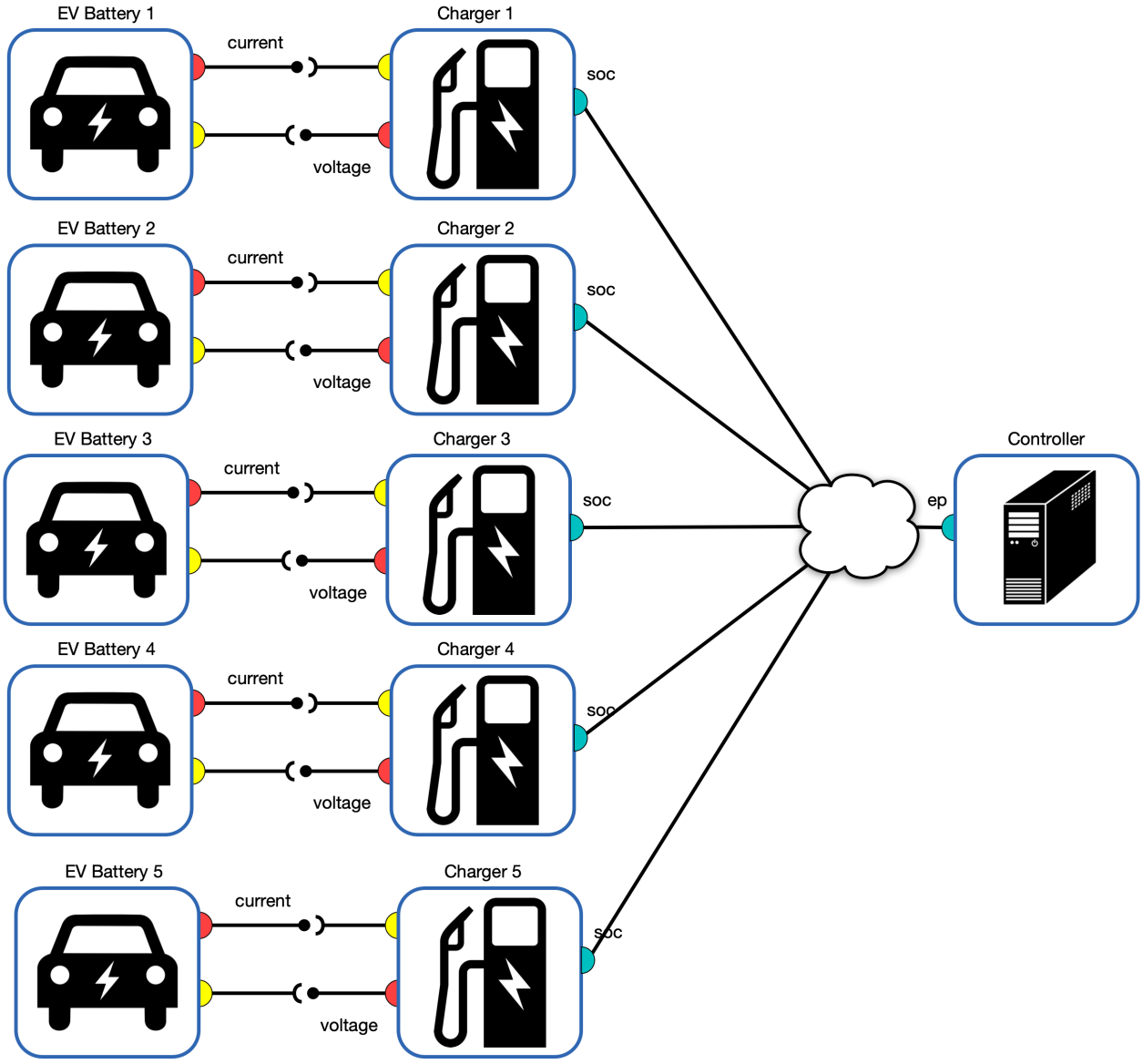
Note that we have made a number of simplifying assumptions in this analysis:

- There will always be an EV waiting to be charged (the charging ports are never idle).
- There is a constant number of charging ports – we know what the power draw will look like given a static number of ports, but we do not know the underlying demand for power from EVs.
- The equipment which ferries the messages between the Charger and the Controller never fails – we haven't incorporated *Filters*.

How would you model an unknown demand for vehicle charging? How would you model idle charging ports? What other simplifications do you see that can be addressed?

Alternative implementation: Multi-Agent

As a point of comparison, there is an alternative implementation of the same functionality where the five chargers and five batteries in `Battery.py` and `Charger.py` are disaggregated into unique federates of their own. This creates a co-simulation with five battery federates, five charger federates, and one controller federate. The resulting message-passing topology looks like this:



You can run this example just like the integrated example from the `multi_agent` folder:

```
helics run --path=fundamental_multi_agent_runner.json
```

The example takes longer to run but produces identical results to the integrated example. The longer runtime demonstrates the common performance cost of co-simulation: passing information via HELICS, where most HELICS cores use the network stack, is much slower than passing values in memory. We use HELICS, and co-simulation in general, to integrate models that otherwise would not be possible to integrate.

Questions and Help

Do you have questions about HELICS or need help?

1. Come to office hours!
2. Post on the gitter!
3. Place your question on the github forum!

Combination Federation with Native HELICS Filters

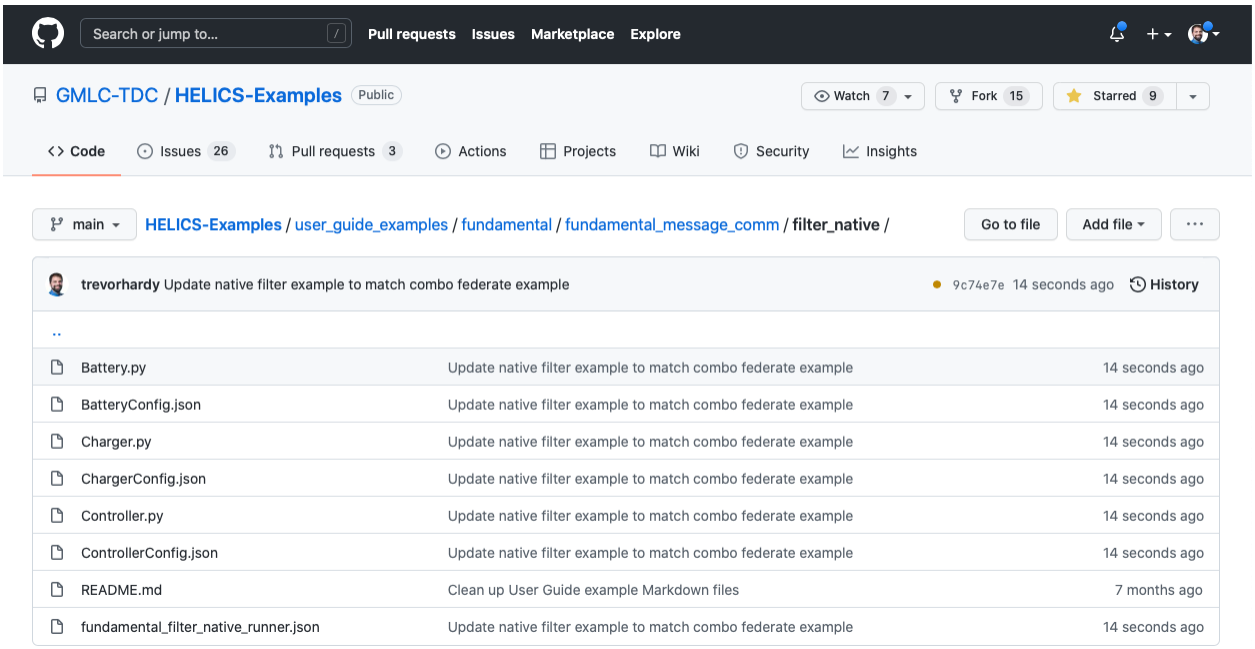
This custom filter federate example expands the *combination federation example* to demonstrate how HELICS filters can be added to endpoints.

This tutorial is organized as follows:

- *Example files*
- *HELICS Filters*
- *Co-simulation Execution*
- *Questions and Help*

Example files

All files necessary to run the Native Filter Example can be found in the Fundamental examples repository:



- Python program and configuration JSON for Battery federate
- Python program and configuration JSON for Charger federate
- Python program and configuration JSON for Controller federate
- HELICS runner JSON to enable execution of the co-simulation

HELICS filters

As discussed in the *User Guide*, filters have the ability to act on messages being sent between endpoints and perform various functions on them (delay, drop, reroute, etc). In this example, we're going to take the *Combination Federate Example* and add a filter between the Charger and the Controller (since this is where the messages are flowing). Specifically, we're adding a destination filter to the controller endpoint such that all messages received by that endpoint will be delayed. Note that this filter will only act on the messages received at this endpoint and not those sent out from it. Here's the entire Controller JSON config:

```
{
  "name": "Controller",
  "log_level": "warning",
  "core_type": "zmq",
  "time_delta": 1,
  "uninterruptible": false,
  "terminate_on_error": true,
  "endpoints": [
    {
      "name": "Controller/ep",
      "global": true
    }
  ],
  "filters": [
    {
      "name": "ep_filter",
      "destination_target": "Controller/ep",
      "operation": "delay",
      "properties": {
        "name": "delay",
        "value": "900s"
      }
    }
  ]
}
```

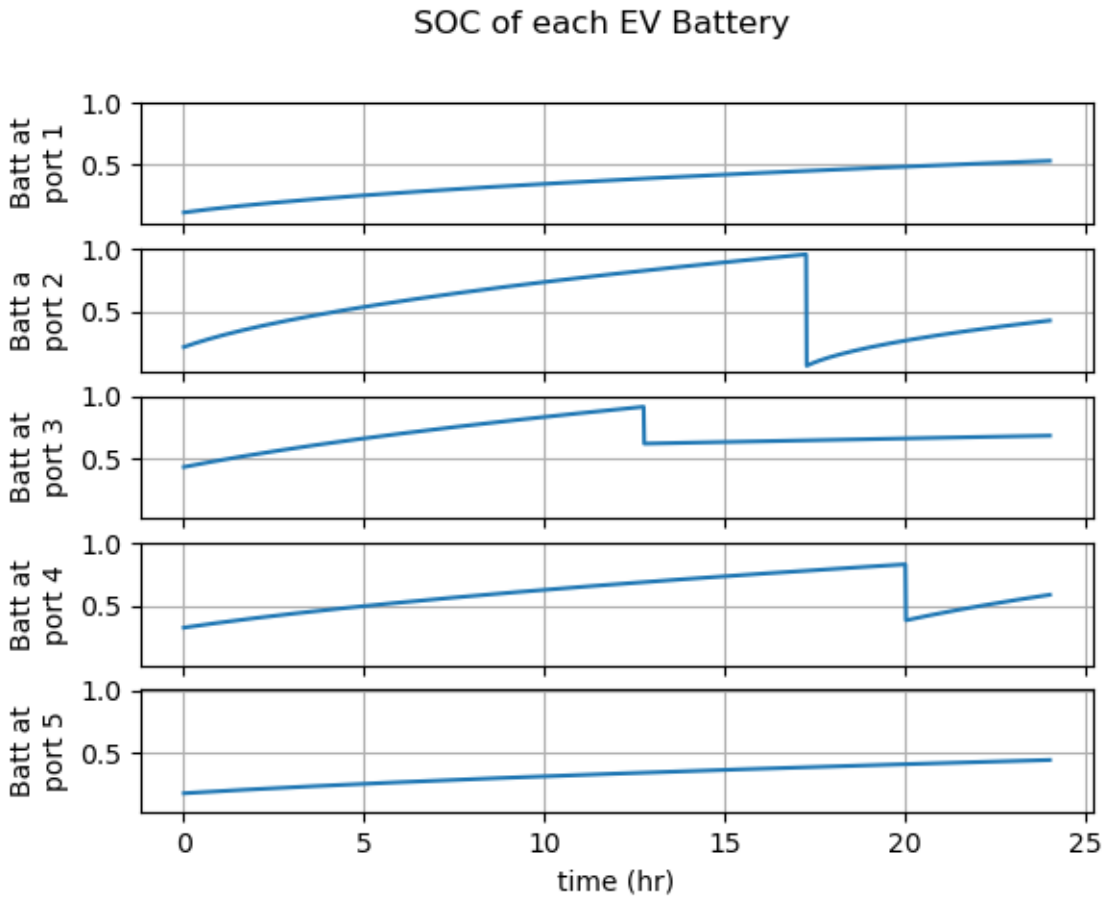
The delay value is set to 900 seconds so that the impact of the fictitious communication system delay is obvious in the results of this example and the delay can be set to much lower or higher values.

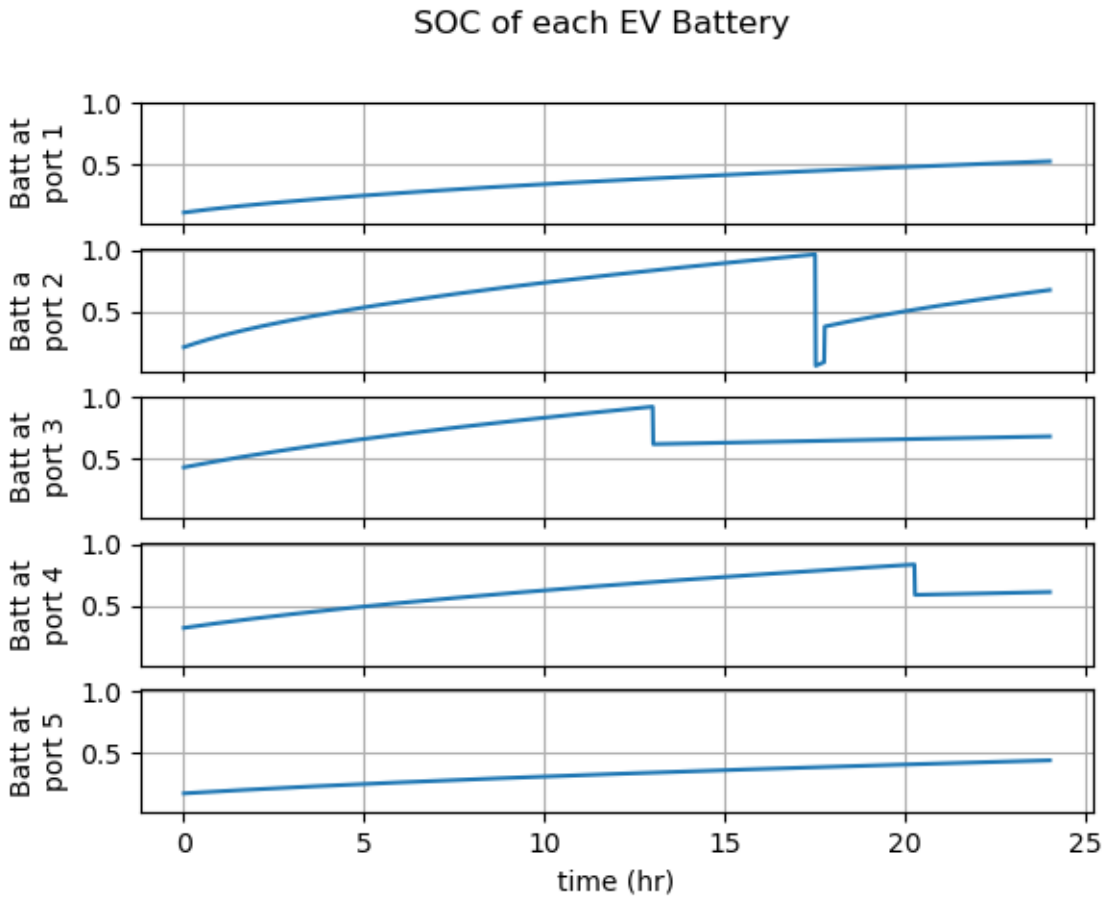
Co-simulation execution

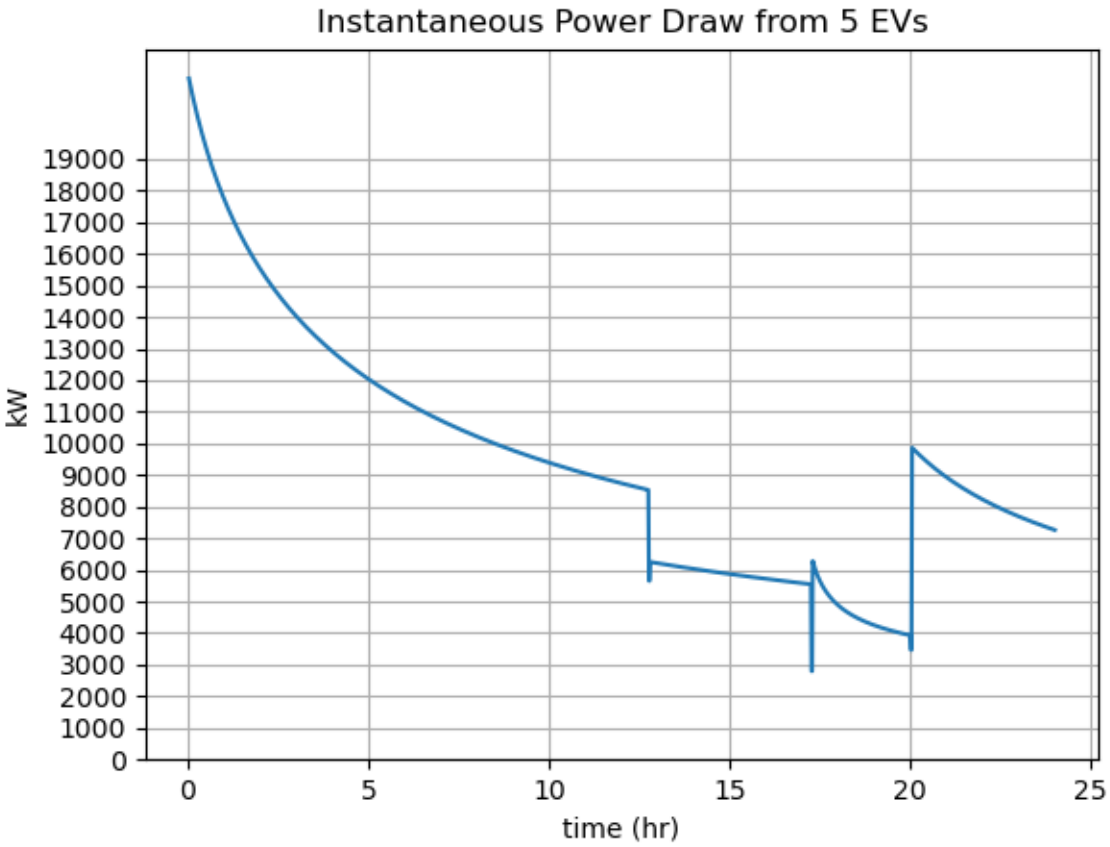
Execution of this co-simulation is done as before with the `helics run` command:

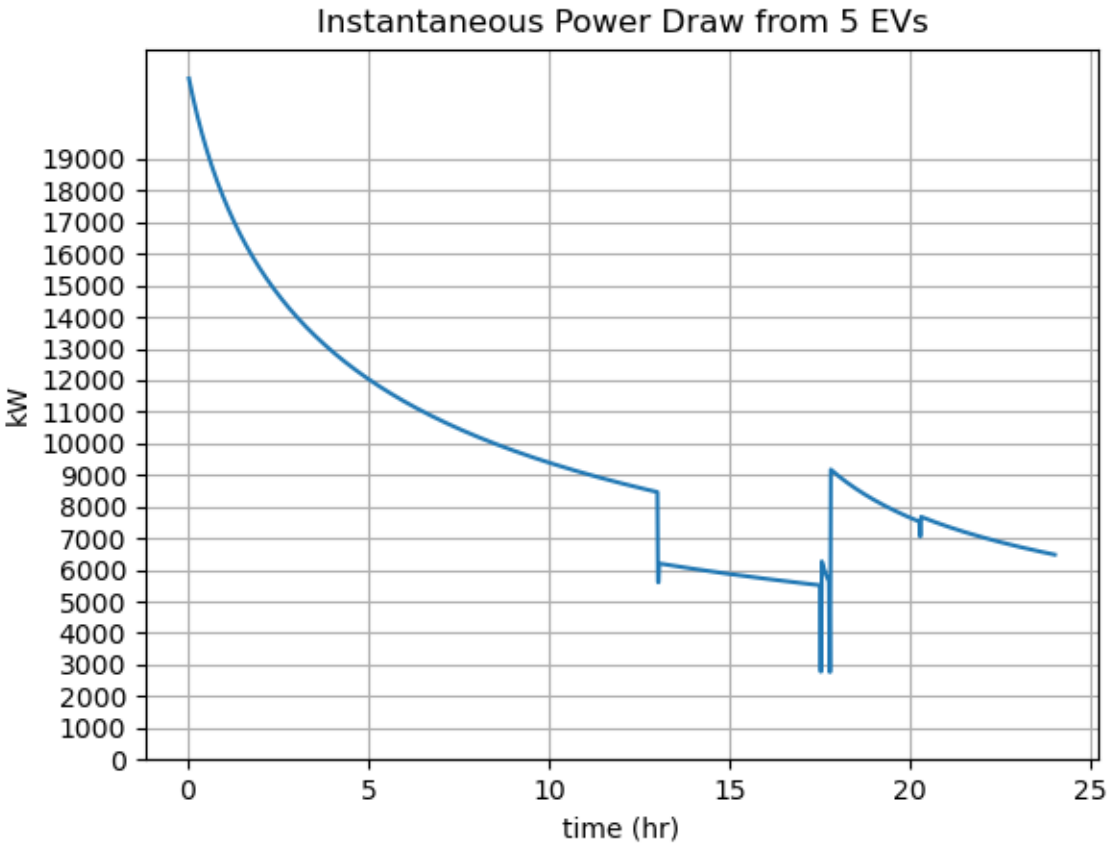
```
helics run --path=fundamental_combo_runner.json
```

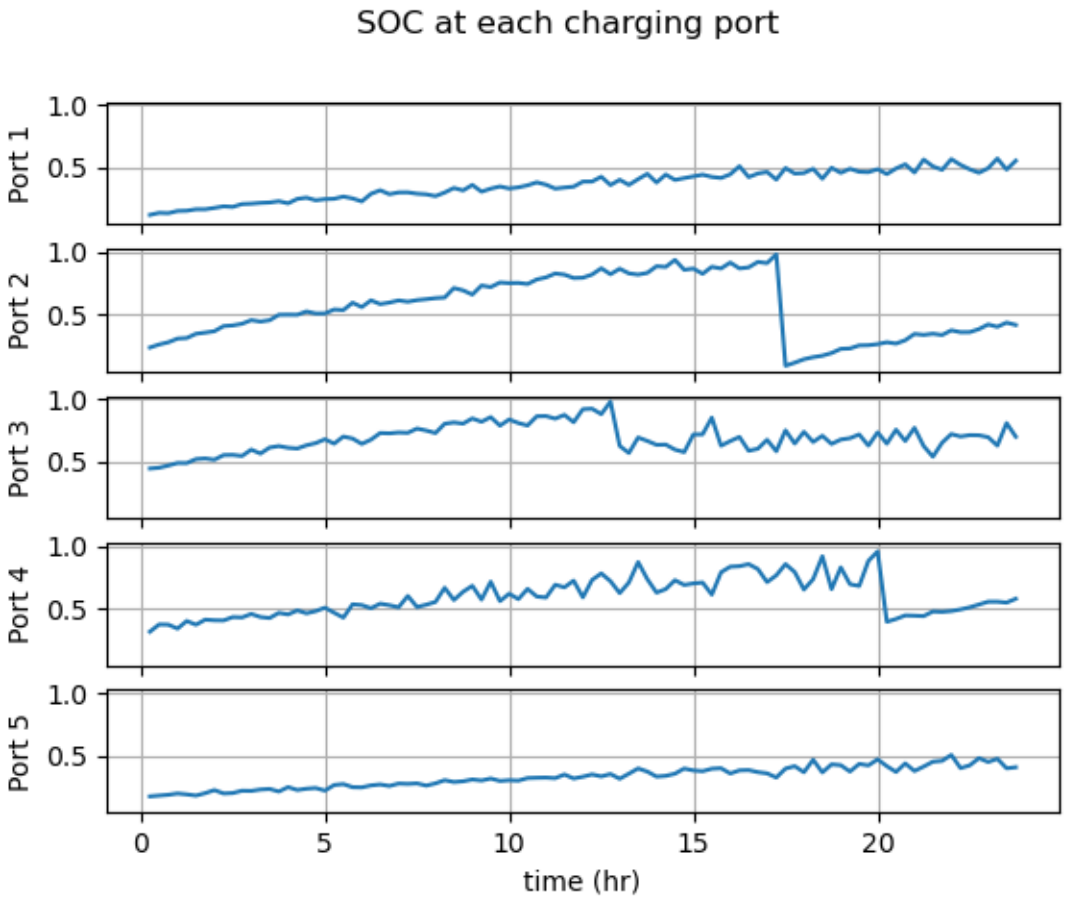
Below are pairs of output graphs with the first from each pair being from the original Combination Federate Example and the second from this Native Filter Example with the communication delay.

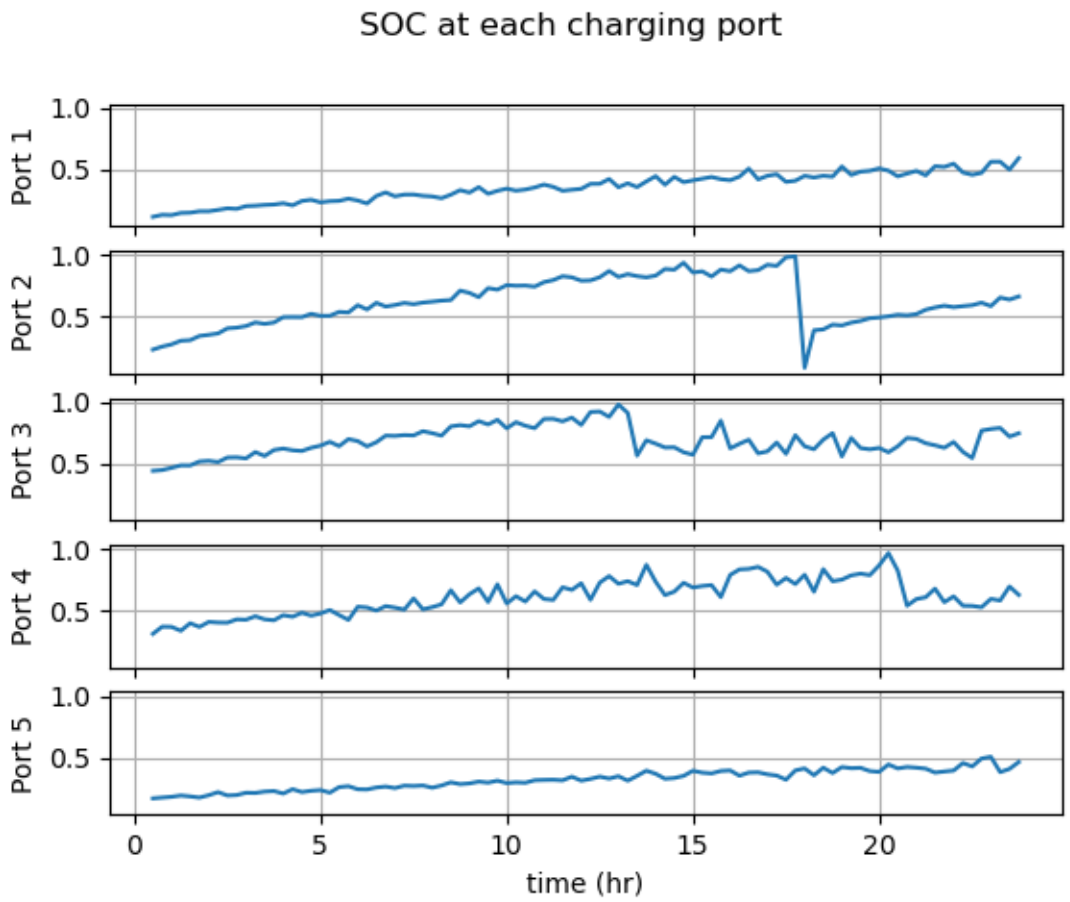

SOC of each EV Battery
Batt at port 1
Batt a port 2
Batt at port 3
Batt at port 4
Batt at port 5
1.0
0.5
0
5
10
15
20
25
time (hr)


SOC of each EV Battery
Batt at port 1
Batt a port 2
Batt at port 3
Batt at port 4
Batt at port 5
1.0
0.5
0
5
10
15
20
25
time (hr)


Instantaneous Power Draw from 5 EVs
kW
19000
18000
17000
16000
15000
14000
13000
12000
11000
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0
5
10
15
20
25
time (hr)


Instantaneous Power Draw from 5 EVs
kW
19000
18000
17000
16000
15000
14000
13000
12000
11000
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0
5
10
15
20
25
time (hr)


SOC at each charging port
Port 1
Port 2
Port 3
Port 4
Port 5
1.0
0.5
0
5
10
15
20
time (hr)



Questions and Help

Do you have questions about HELICS or need help?

1. Come to office hours!
2. Post on the gitter!
3. Place your question on the github forum!

Combination Federation with Custom Filter Federates

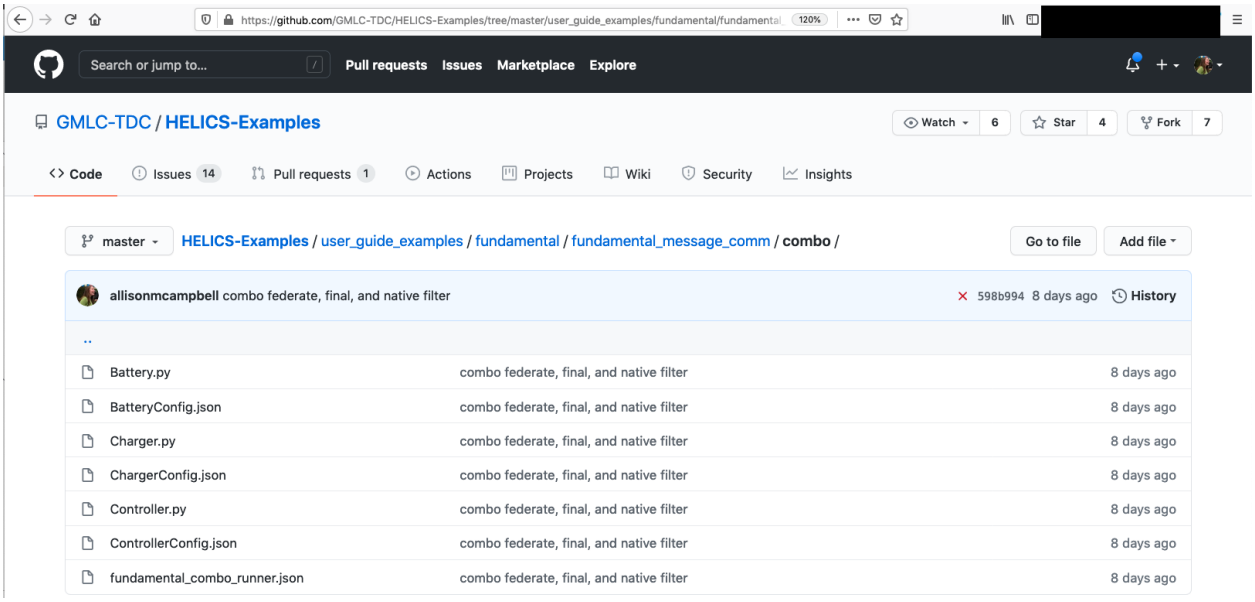
This custom filter federate example expands the Native Filters example to demonstrate . This example assumes the user has already worked through the *Native Filter Example* and understands the role of filters in a HELICS-based co-simulation.

This tutorial is organized as follows:

- *Example files*
- *Filter Federates*
  - *Co-simulation Execution*
- *Questions and Help* talk

Example files

All files necessary to run the Federate Integration Example can be found in the Fundamental examples repository:

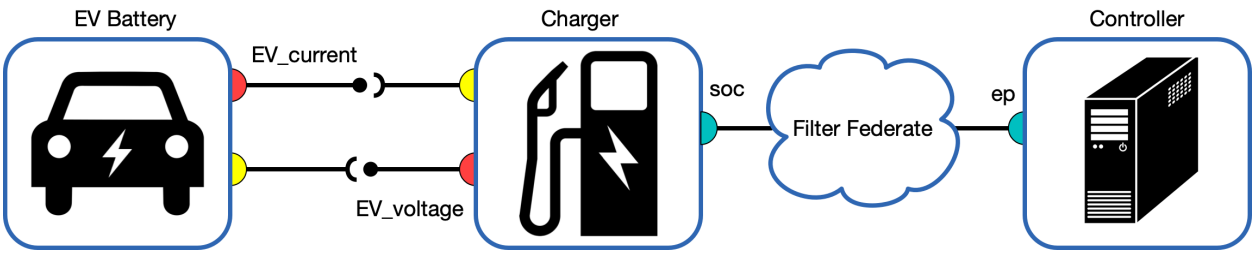


- Python program and configuration JSON for Battery federate
- Python program and configuration JSON for Charger federate
- Python program and configuration JSON for Controller federate
- Python program and configuration JSON for Filter federate
- (Bonus Python program where the Filter Federate does no filtering and forwards on all received messages)
- HELICS runner JSON to enable execution of the co-simulation

Filter Federates

For situations that require filtering beyond what native HELICS filters can provide, it is possible to create a custom filter federate that acts in specific, user-define ways. Not only is it possible to re-create the native HELICS filter functions (*e.g.* delaying or randomly dropping messages) but it is also possible act on the payloads of the messages themselves. Some of these functionalities (such as dropped or delayed messages) can be achieved through existing simulation tools such as an ns-3. Even in an ns-3, though custom functionality that would modify message payloads would require writing custom applications and compiling them into ns-3. Though this example custom filter federate doesn't implement many of the detailed networking models in an ns-3 such as TCP and IP addressing, it's simplicity is a virtue in that it demonstrates how to implement a custom filter federate to act on HELICS messages an arbitrary manner.

Here is our new federation using a custom filter federate:



We have:

- Battery (**value federate**): passes values with Charger through pub/subs)
- Charger (**combo federate**): passes values with Battery, passes messages with Controller)
- Controller (**message federate**): passes messages with Charger through endpoints)
- Filter (**message federate**): acts on all messages sent between the Charger and the Controller)

Filter Federate Configuration

Though not shown in the Federation diagram, the use of native HELICS filters is an essential part of using a custom filter federate. A "reroute" native helix filter is installed on all the federeation endpoints as a part of the configuration of the filter federate. This reroute filter sends all messages from the M points to the filter federate for processing.

```
{
  "name": "Filter",
  "event_triggered": true,
  "endpoints": [
    {
      "name": "filter/main",
      "global": true
    }
  ],
  "filters": [
    {
      "name": "filterFed",
      "sourcetargets": [
        "Charger/EV1.soc",
        "Charger/EV2.soc",
        "Charger/EV3.soc",
        "Charger/EV4.soc",
        "Charger/EV5.soc",
        "Controller/ep"
      ],
      "operation": "reroute",
      "properties": {
        "name": "newdestination",
        "value": "filter/main"
      }
    }
  ]
}
```

The filter federate will now be granted time whenever a message is sent from any of the existing federation endpoints shown in the `sourcetargets` list. The filter federate has only a single endpoint which it uses to receive the rerouted messages and send on any modified messages.

Additionally, all filter federates should set the `event_triggered` flag as shown above. This increases the timing efficiency managed by HELICS and avoids potential timing lock-ups.

Filter Federate Operations

The filter federate included in this example implements for functions:

- **Random message drops** - A random number generator and a user-adjustable parameter defines what portion of messages traveling through the filter are randomly dropped/deleted.

- **Message delay** - A random number generator and a user-adjustable parameter defines a random delay time for all messages traveling through the filter.
- **Message hacking** - for all messages originating from the controller, a random number generator and a user defined parameter define whether the contents of that message are adjusted. In this case, the contents of the payload ate either a one or zero used to indicate where the EV should continue charging; the filter federate simply inverts the message value if it is selected to be hacked.
- **Interference** - Messages that are destined to reach the Controller at the same time can interfere with each other. User-defined parameter defines how closely in time two messages must be arriving to interfere with each other.

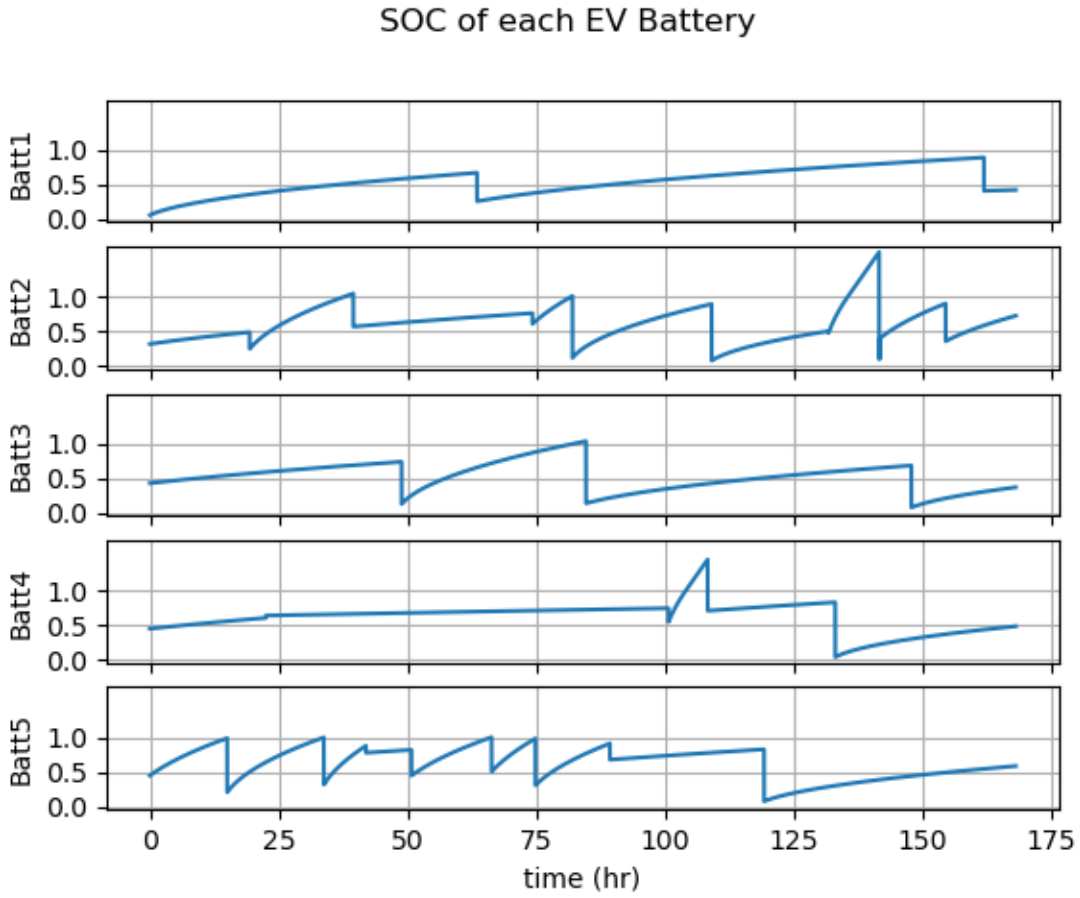
Though these operations are just a sample of what any filter federate could do, they are representative of the types of communication system effects that are often sought to be represented through more complex simulators such as ns-3. This simulator contains no network topology; all messages are processed as if traveling through a single communication node. The source code shows the implementation of these functions, and the log files generated by the Filter federate are comprehensive.

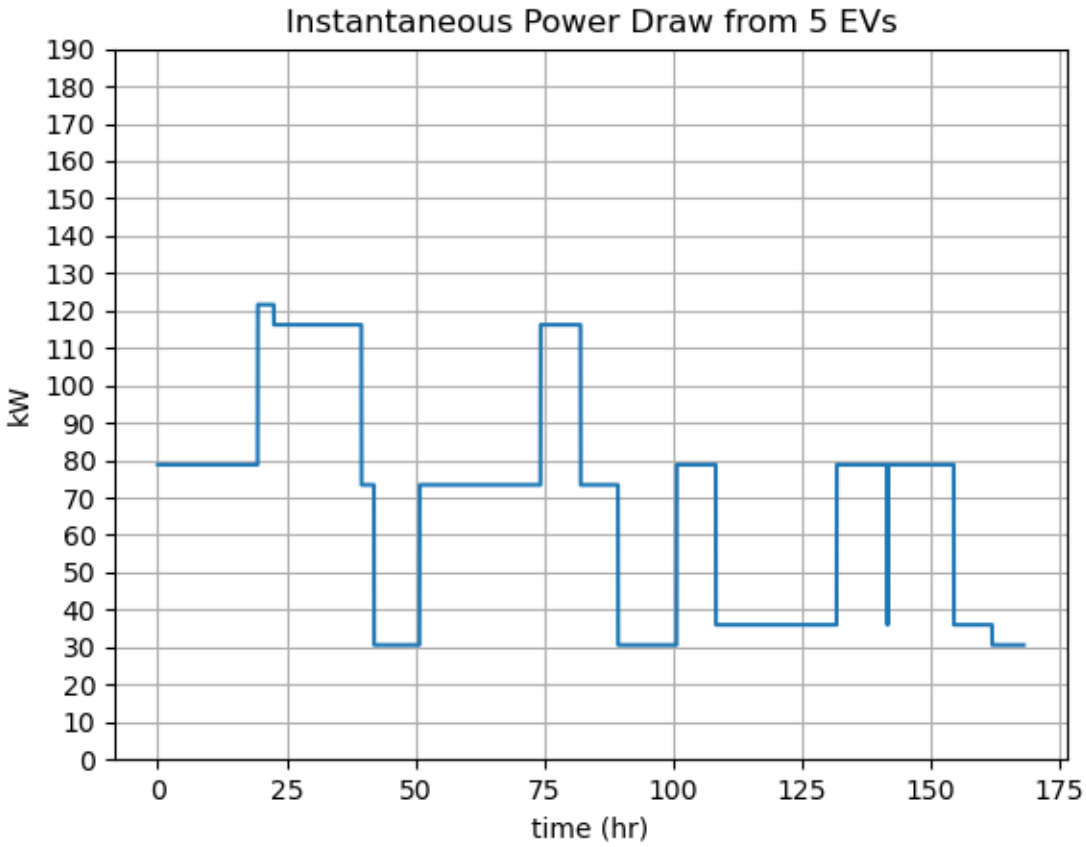
Co-simulation execution

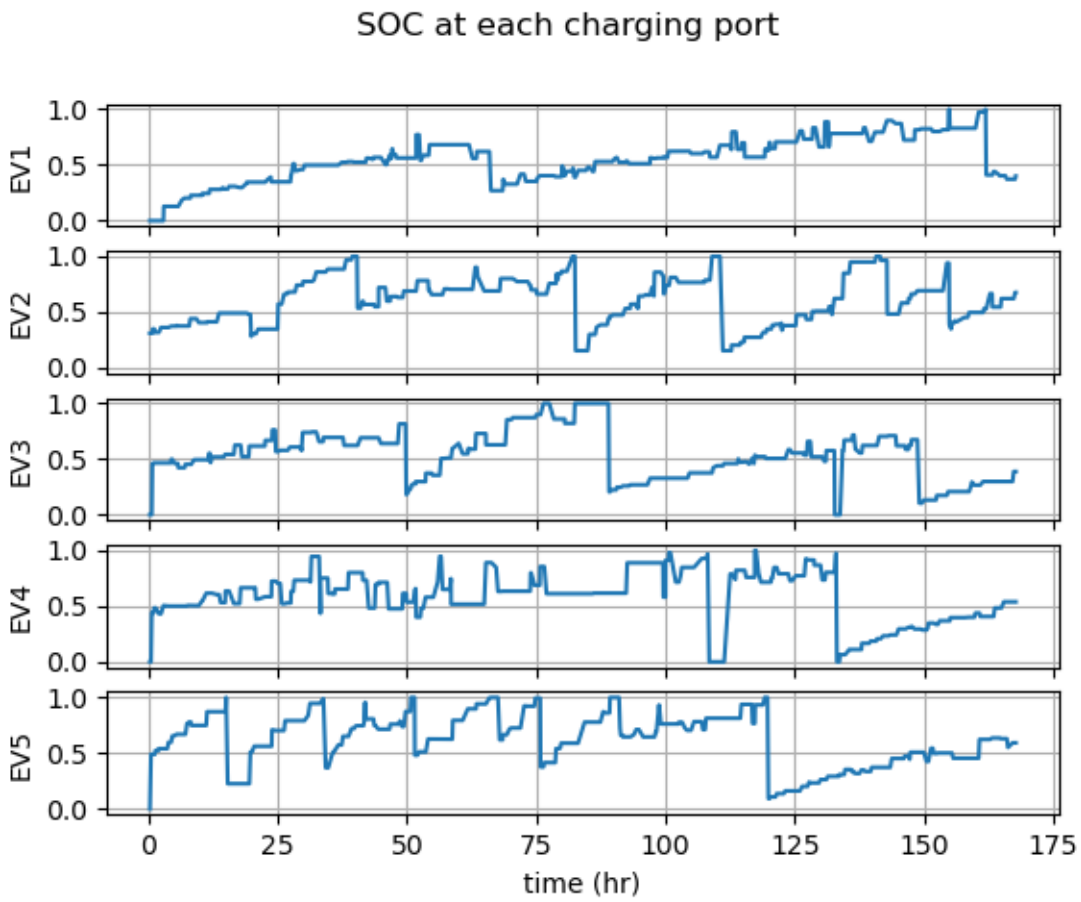
Execution of this co-simulation is done as before with the `helics_run` command:

```
helics run --path=./fundamental_filter_runner.json
```

The resulting figures show the actual on board SOC at each EV charging port, the instantaneous power draw, and the SOC estimated by the on board charger.




Instantaneous Power Draw from 5 EVs
kW
time (hr)
0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150
160
170
180
190
0
25
50
75
100
125
150
175



When comparing to the results from the *previous example without any filters*, the effects of the filter federate are clear. By modifying the control signals between the controller and charger it is relatively easy to cause significantly different behavior in the system.

Questions and Help

Do you have questions about HELICS or need help?

1. Come to office hours!
2. Post on the gitter!
3. Place your question on the github forum!

In the *Base Example*, we saw information passed between two federates using publications and subscriptions (pubs/subs). In addition to pubs/subs, where information is passed as *values* (physical parameters of the system), federates can also pass information between **endpoints**, where this information is now a *message*.

This section on message and communication configuration will walk through how to set up two federates to pass messages using *endpoints*, and how to set up three federates which pass between them values and messages with a *combination* of the two configurations. It will also demonstrate how to use native HELICS filters and how to set up a *custom filter federate* to act on messages in-flight between federates.

Federate Integration with PyHELICS API

The Federate Integration Example extends the Base Example to demonstrate how to integrate federates using the HELICS API instead of JSON config files. This tutorial is organized as follows:

- *Computing Environment*
- *Example files*
- *Federate Integration using the PyHELICS API*
  - *Translation from JSON to PyHELICS API methods*
  - *Federate Integration with API calls*
  - *Dynamic Pub/Subs with API calls*
  - *Co-simulation Execution*
- *Questions and Help*

Computing Environment

This example was successfully run on `Tue Nov 10 11:16:44 PST 2020` with the following computing environment.

- Operating System

```
$ sw_vers
ProductName:    Mac OS X
ProductVersion:    10.14.6
BuildVersion:    18G6032
```

- python version

```
$ python
Python 3.7.6 (default, Jan  8 2020, 13:42:34)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

- python modules for this example

```
$ pip list | grep matplotlib
matplotlib                    3.1.3
$ pip list | grep numpy
numpy                         1.18.5
```

If these modules are not installed, you can install them with

```
$ pip install matplotlib
$ pip install numpy
```

- helics_broker version

```
$ helics_broker --version
2.4.0 (2020-02-04)
```

- pyhelics version

```
$ helics --version
0.4.1-HEAD-ef36755
```

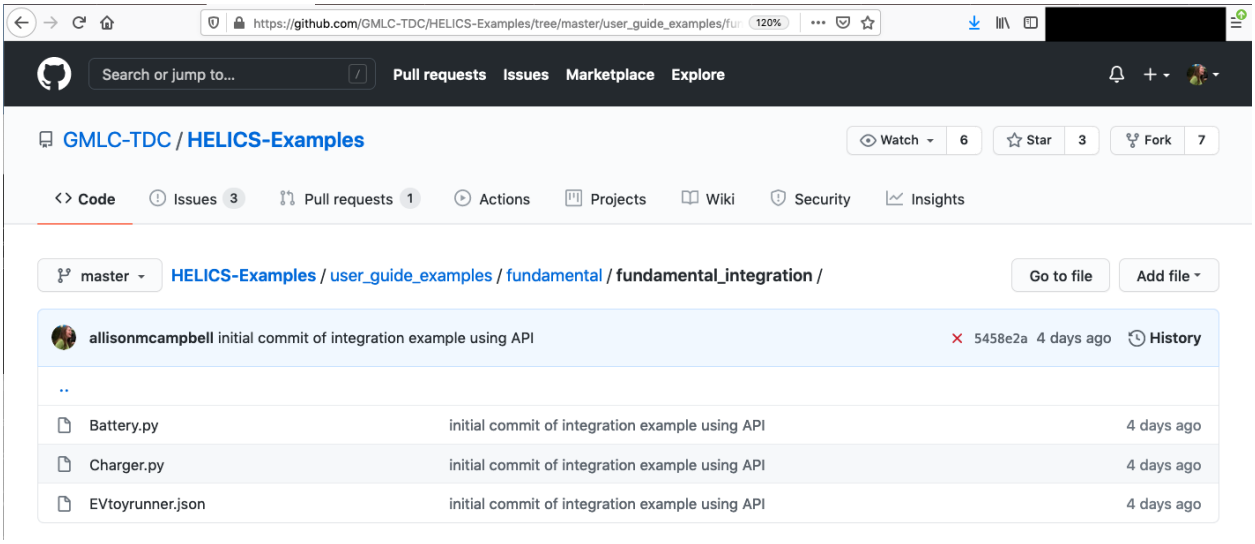
- pyhelics init file

```
$ python

>>> import helics as h
>>> h.__file__
'/Users/[username]/Software/pyhelics/helics/__init__.py'
```

Example files

All files necessary to run the Federate Integration Example can be found in the Fundamental examples repository:



The files include:

- Python program for Battery federate
- Python program for Charger federate
- HELICS runner JSON to enable execution of the co-simulation

Federate Integration using the PyHELICS API

This example differs from the Base Example in that we integrate the federates (simulators) into the co-simulation using the API instead of an external JSON config file. Integration and configuration of federates can be done either way – the biggest hurdle for most beginning users of HELICS is learning how to look for the appropriate API key to mirror the JSON config style.

For example, let's look at our JSON config file of the Battery federate from the Base Example:

```
{
  "name": "Battery",
  "loglevel": 1,
  "coreType": "zmq",
  "period": 60,
  "uninterruptible": false,
  "terminate_on_error": true,
  "wait_for_current_time_update": true,
  "publications": [],
```

(continues on next page)

(continued from previous page)

```
  "subscriptions": []
}
```

We can see from this config file that we need to find API method to assign the `name`, `loglevel`, `coreType`, `period`, `uninterruptible`, `terminate_on_error`, `wait_for_current_time_update`, and `pub/subs`. In this example, we will be using the PyHELICS API methods. This section will discuss how to translate JSON config files to API methods, how to configure the federate with these API calls in the co-simulation, and how to dynamically register publications and subscriptions with other federates.

Translation from JSON to PyHELICS API methods

Configuration with the API is done within the federate, where an API call sets the properties of the federate. With our Battery federate, the following API calls will set all the properties from our JSON file (except pub/sub, which we'll cover in a moment). These calls set:

1. `name`
2. `loglevel`
3. `coreType`
4. Additional core configurations
5. `period`
6. `uninterruptible`
7. `terminate_on_error`
8. `wait_for_current_time_update`

```
h.helicsCreateValueFederate("Battery", fedinfo)
h.helicsFederateInfoSetIntegerProperty(fedinfo, h.HELICS_PROPERTY_INT_LOG_LEVEL, 1)
h.helicsFederateInfoSetCoreTypeFromString(fedinfo, "zmq")
h.helicsFederateInfoSetCoreInitString(fedinfo, fedinitstring)
h.helicsFederateInfoSetTimeProperty(fedinfo, h.HELICS_PROPERTY_TIME_PERIOD, 60)
h.helicsFederateInfoSetFlagOption(fedinfo, h.HELICS_FLAG_UNINTERRUPTIBLE, False)
h.helicsFederateInfoSetFlagOption(fedinfo, h.HELICS_FLAG_TERMINATE_ON_ERROR, True)
h.helicsFederateInfoSetFlagOption(
    fedinfo, h.HELICS_FLAG_WAIT_FOR_CURRENT_TIME_UPDATE, True
)
```

If you find yourself wanting to set additional properties, there are a handful of places you can look:

- C++ source code: Do a string search for the JSON property. This can provide clarity into which `enum` to use from the API.
- PyHELICS API methods: API methods specific to PyHELICS, with suggestions for making the calls pythonic.
- *Configuration Options Reference*: API calls for C++, C, Python, and Julia

Federate Integration with API calls

We now know which API calls are analogous to the JSON configurations – how should these methods be called in the co-simulation to properly integrate the federate?

It's common practice to rely on a helper function to integrate the federate using API calls. With our Battery/Controller co-simulation, this is done by defining a `create_value_federate` function (named for the fact that the messages passed between the two federates are physical values). In `Battery.py` this function is:

```
def create_value_federate(fedinitstring, name, period):
    fedinfo = h.helicsCreateFederateInfo()
    h.helicsFederateInfoSetCoreTypeFromString(fedinfo, "zmq")
    h.helicsFederateInfoSetCoreInitString(fedinfo, fedinitstring)
    h.helicsFederateInfoSetIntegerProperty(fedinfo, h.HELICS_PROPERTY_INT_LOG_LEVEL, 1)
    h.helicsFederateInfoSetTimeProperty(fedinfo, h.HELICS_PROPERTY_TIME_PERIOD, period)
    h.helicsFederateInfoSetFlagOption(fedinfo, h.HELICS_FLAG_UNINTERRUPTIBLE, False)
    h.helicsFederateInfoSetFlagOption(fedinfo, h.HELICS_FLAG_TERMINATE_ON_ERROR, True)
    h.helicsFederateInfoSetFlagOption(
        fedinfo, h.HELICS_FLAG_WAIT_FOR_CURRENT_TIME_UPDATE, True
    )
    fed = h.helicsCreateValueFederate(name, fedinfo)
    return fed
```

Notice that we have passed three items to this function: `fedinitstring`, `name`, and `period`. This allows us to flexibly reuse this function if we decide later to change the name or the period (the most common values to change).

We create the federate and integrate it into the co-simulation by calling this function at the beginning of the program main loop:

```
fedinitstring = " --federates=1"
name = "Battery"
period = 60
fed = create_value_federate(fedinitstring, name, period)
```

What step created the value federate?

`fed = h.helicsCreateValueFederate(name, fedinfo)`

Notice that we pass to this API the `fedinfo` set by all preceding API calls.

Dynamic Pub/Subs with API calls

In the Base Example, we configured the pubs and subs with an external JSON file, where *each* publication and subscription between federate interfaces needed to be explicitly defined for a predetermined number of connections:

```
"publications":[
  {
    "key":"Battery/EV1_current",
    "type":"double",
    "unit":"A",
    "global": true
  },
  {}
],
"subscriptions":[
  {
    "key":"Charger/EV1_voltage",
    "type":"double",
    "unit":"V",
    "global": true
  },
  {}
]
```

With the PyHELICS API methods, you have the flexibility to define the connection configurations *dynamically* within execution of the main program loop. For example, in the Base Example we defined **five** communication connections between the Battery and the Charger, meant to model the interactions of five EVs each with their own charging port. If we want to increase or decrease that number using JSON configuration, we need to update the JSON file (either manually or with a script).

Using the PyHELICS API methods, we can register any number of publications and subscriptions. This example sets up pub/sub registration using for loops:

```
num_EVs = 5
pub_count = num_EVs
pubid = {}
for i in range(0, pub_count):
    pub_name = f"Battery/EV{i+1}_current"
    pubid[i] = h.helicsFederateRegisterGlobalTypePublication(
        fed, pub_name, "double", "A"
    )

sub_count = num_EVs
subid = {}
for i in range(0, sub_count):
    sub_name = f"Charger/EV{i+1}_voltage"
    subid[i] = h.helicsFederateRegisterSubscription(fed, sub_name, "V")
```

Here we only need to designate the number of connections to register in one place: `num_EVs = 5`. Then we register the publications using the `h.helicsFederateRegisterGlobalTypePublication()` method, and the subscriptions with the `h.helicsFederateRegisterSubscription()` method. Note that subscriptions are analogous to inputs, and as such retain similar properties.

Co-simulation Execution

In this tutorial, we have covered how to integrate federates into a co-simulation using the PyHELICS API. Integration covers configuration of federates and registration of communication connections. Execution of the co-simulation is done the same as with the Base Example, using a runner JSON. The runner JSON has not changed from the Base Example:

```
{
  "name": "fundamental_integration",
  "federates": [
    {
      "directory": ".",
      "exec": "helics_broker -f 3 --loglevel=warning",
      "host": "localhost",
      "name": "broker"
    },
    {
      "directory": ".",
      "exec": "python -u Charger.py",
      "host": "localhost",
      "name": "Charger"
    },
    {
      "directory": ".",
      "exec": "python -u Controller.py",
```

(continues on next page)

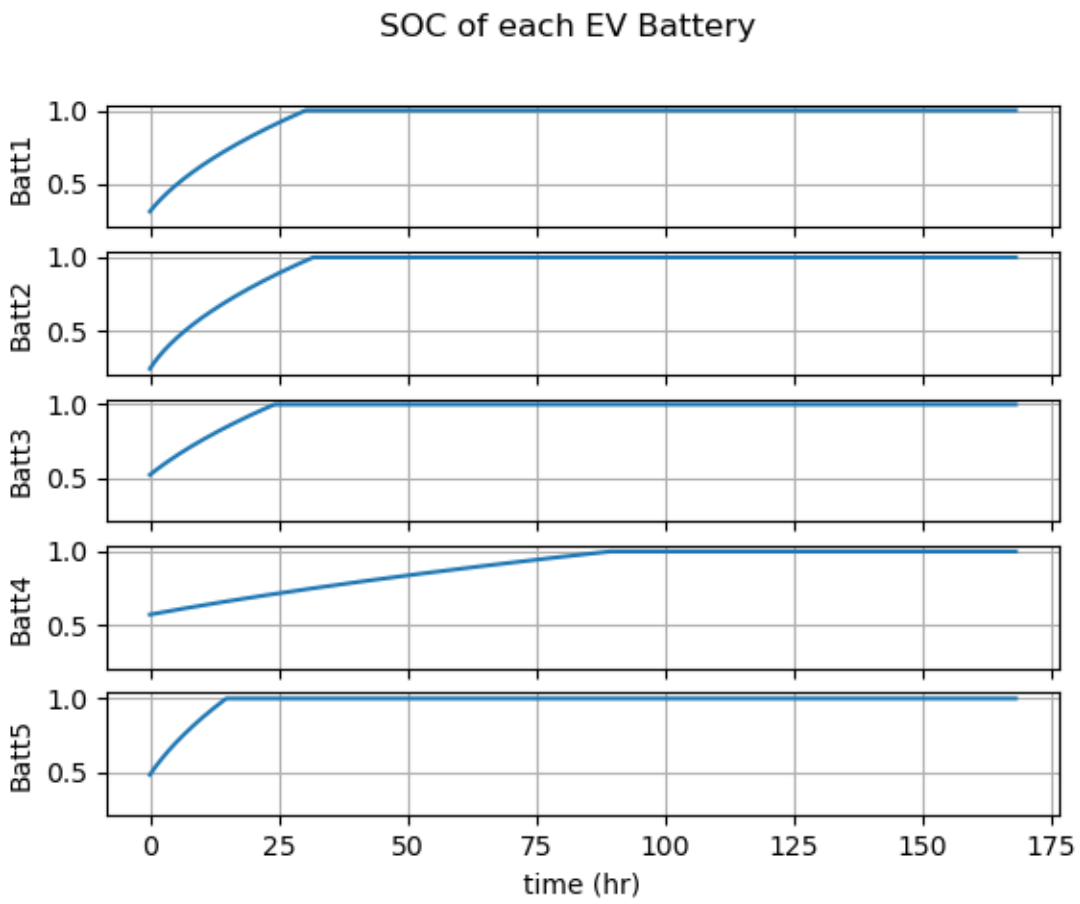
(continued from previous page)

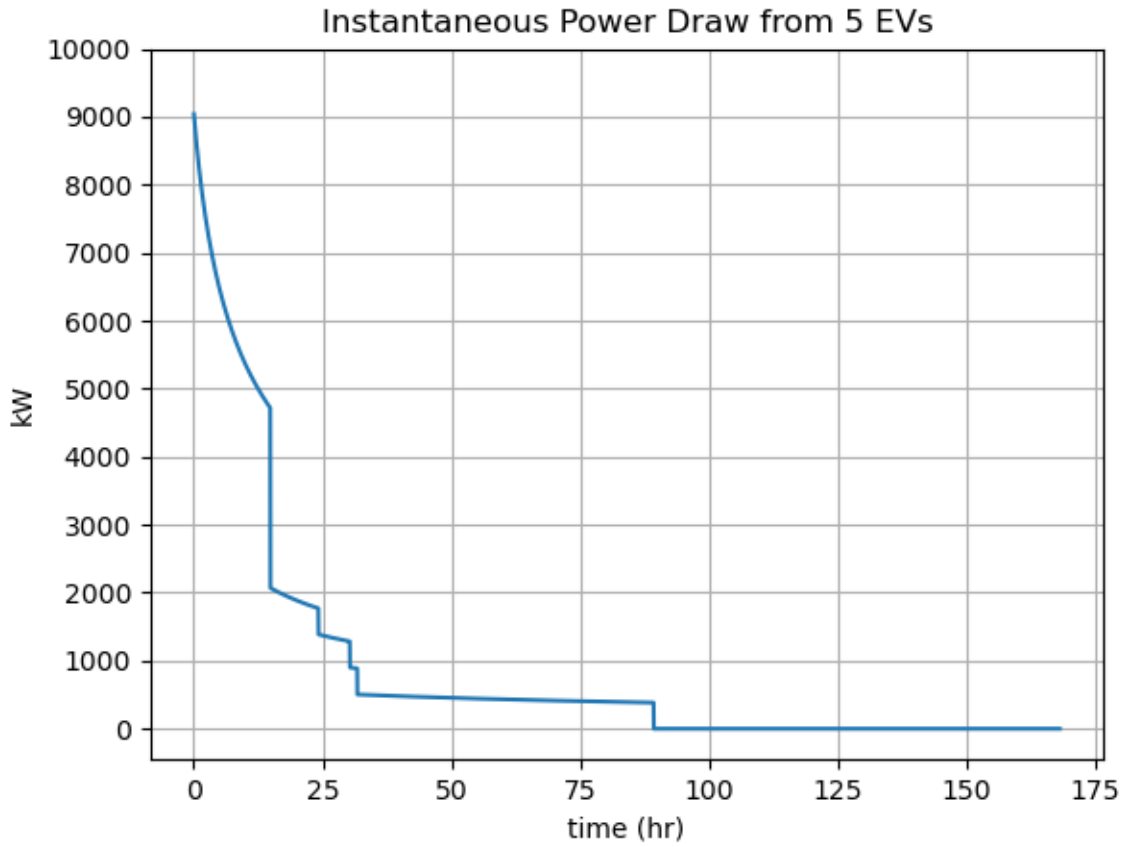
```
      "host": "localhost",
      "name": "Controller"
    },
    {
      "directory": ".",
      "exec": "python -u Battery.py",
      "host": "localhost",
      "name": "Battery"
    }
  ]
}
```

Execute the co-simulation with the same command as the Base Example

```
helics run --path=fundamental_integration_runner.json
```

This results in the same output; the only thing we have changed is the method of configuring the federates and integrating them.





If your output is not the same as with the Base Example, it can be helpful to pinpoint source of the difference – have you used the correct API method?

Questions and Help

Do you have questions about HELICS or need help?

1. Come to office hours!
2. Post on the gitter!
3. Place your question on the github forum!

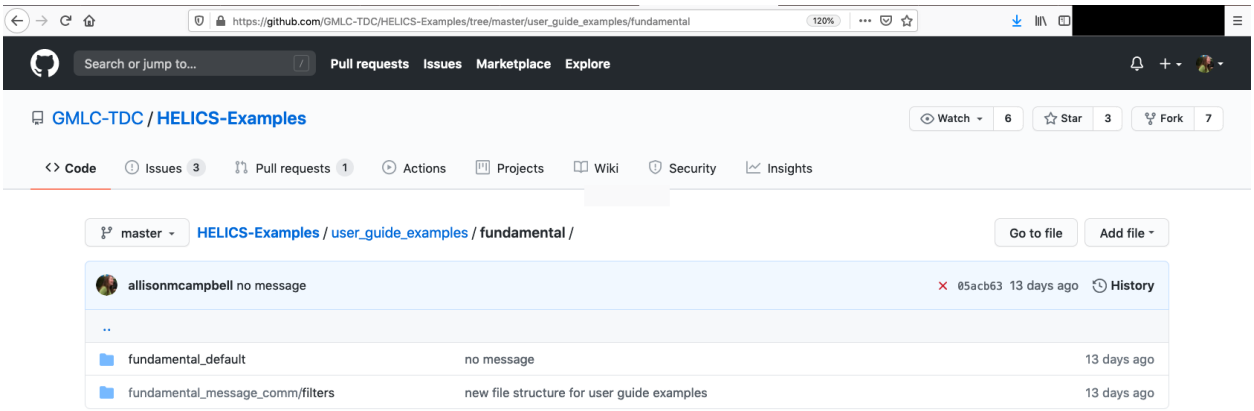
The Fundamental examples are meant to build in complexity – if you are new to HELICS, we recommend you start with the Base Example, which is also the recommended default setup. The examples in this section start with the simplest configuration method, which makes assumptions about the system which may not be completely valid but are reasonable for learning purposes.

This page describes the model – what is the research question addressed, and what are the components to a simple HELICS co-simulation:

- Where is the Code?
- What is this Co-simulation Doing?
- HELICS Components
  - Register and Configure Federates

- Enter Execution Mode
- Define Time Variables
- Initiate Time Steps for the Time Loop
- Send/Receive Communication between Federates
- Finalize Co-simulation

The code for the Fundamental examples can be found in the HELICS-Examples repository on github. If you have issues navigating the examples, visit the HELICS Gitter page or the user forum on GitHub.



The Fundamental Examples model the interaction between five electric vehicles (EVs) each connected to a charging station – five EVs, five charging stations. You can imagine that five EVs enter a parking garage filled with charging stations (charging garage). We will be modeling the EVs as if they are solely their on-board batteries.

Imagine you are the engineer assigned to assess how much power will be needed to serve EVs in this charging garage. The goal of the co-simulation is to calculate the **instantaneous power draw** from the EVs in the garage.

Some questions you might ask yourself include:

1. How many EVs are likely to be charging at any given time?
2. What is the charge rate (power draw limitation) for these EVs?
3. What is the battery size (total capacity) for these EVs?

For now, we've decided that there will always be five EVs and five charging stations. We can also define a few functions to assign the charge rates and the battery sizes. This requires some thinking about *which* federate will manage *what* information.

The co-simulation has two federates: one for the EVs, and one for the Chargers. The batteries on board the EVs will remain the domain of the EVs and inform the EV state of charge (SOC). The Chargers will manage the voltage applied to the EV batteries, and will retain knowledge of the rate of charge.

Within the federate `Battery.py`, EV battery sizes can be generated using the function `get_new_battery`:

```
def get_new_battery(numBattery):
    # Probabilities of a new EV battery having small capacity (lvl1),
    # medium capacity (lvl2), and large capacity (lvl3).
    lvl1 = 0.2
    lvl2 = 0.2
    lvl3 = 0.6

    # Batteries have different sizes:
```

(continues on next page)

(continued from previous page)

```
    # [25,62,100]
    listOfBatts = np.random.choice(
        [25, 62, 100], numBattery, p=[lvl1, lvl2, lvl3]
    ).tolist()
    return listOfBatts
```

Within the federate `Charger.py`, the charge rate for each EV is generated using the function `get_new_EV`:

```
def get_new_EV(numEVs):
    # Probabilities of a new EV charging at the specified level.
    lvl1 = 0.05
    lvl2 = 0.6
    lvl3 = 0.35
    listOfEVs = np.random.choice([1, 2, 3], numEVs, p=[lvl1, lvl2, lvl3]).tolist()
    numLvl1 = listOfEVs.count(1)
    numLvl2 = listOfEVs.count(2)
    numLvl3 = listOfEVs.count(3)

    return numLvl1, numLvl2, numLvl3, listOfEVs
```

The probabilities assigned to each of these functions are placeholders – a more advanced application can be found in the *Orchestration Tutorial*.

Now that we know these three quantities – the number of EVs, the capacity of their batteries, and their charge rates, we can build a co-simulation from the two federates. The `Battery.py` federate will update the SOC of each EV after it receives the voltage from the `Charger.py` federate. The `Charger.py` federate will send a voltage signal to the EV until it tells the Charger it has reached its full capacity.

The `Battery.py` federate can tell us the SOC of each EV throughout the co-simulation, and the `Charger.py` federate can tell us the aggregate power draw from all the EVs throughout the co-simulation. The co-simulation will be run for one week.

We know conceptually what we want to (co-)simulate. What are the necessary HELICS components to knit these two federates into one co-simulation?

The first task is to register and configure the federates with HELICS within each python program:

```
##########  Registering  federate and configuring from JSON#################
fed = h.helicsCreateValueFederateFromConfig("BatteryConfig.json")
federate_name = h.helicsFederateGetName(fed)
logger.info(f"Created federate {federate_name}")
```

Since we are configuring with external JSON files, this is done in one line!

The HELICS co-simulation starts by instructing each federate to enter execution mode.

```
##############  Entering Execution Mode  ##################################
h.helicsFederateEnterExecutingMode(fed)
logger.info("Entered HELICS execution mode")
```

Time management is a vital component to HELICS co-simulations. Every HELICS co-simulation needs to be provided information about the start time (`grantedtime`), the end time (`total_interval`) and the time step (`update_interval`). Federates can *step through time at different rates*, and it is allowable to have federates start and stop at different times, but this must be curated to meet the needs of the research question.

```
hours = 24 * 7
total_interval = int(60 * 60 * hours)
update_interval = int(
    h.helicsFederateGetTimeProperty(fed, h.helics_property_time_period)
)
grantedtime = 0
```

Starting the co-simulation time sequence is also a function of the needs of the research question. In the Base Example, the EVs will already be "connected" to the Chargers and will be waiting for the voltage signal from the Charger. This means we need to set up a signal to send from the Charger to the EV *before* the EV requests the signal.

In the `Battery.py` federate, Time is initiated by starting a `while` loop and requesting the first time stamp:

```
while grantedtime < total_interval:

    # Time request for the next physical interval to be simulated
    requested_time = grantedtime + update_interval
    logger.debug(f"Requesting time {requested_time}")
    grantedtime = h.helicsFederateRequestTime(fed, requested_time)
    logger.debug(f"Granted time {grantedtime}")
```

In the `Charger.py` federate, we need to send the first signal **before** entering the time `while` loop. This is accomplished by requesting an initial time (outside the `while` loop), sending the signal, and then starting the time `while` loop:

```
# Blocking call for a time request at simulation time 0
initial_time = 60
logger.debug(f"Requesting initial time {initial_time}")
grantedtime = h.helicsFederateRequestTime(fed, initial_time)
logger.debug(f"Granted time {grantedtime}")


# Apply initial charging voltage
for j in range(0, pub_count):
    h.helicsPublicationPublishDouble(pubid[j], charging_voltage[j])
    logger.debug(
        f"\tPublishing charging voltage of {charging_voltage[j]} "
        f" at time {grantedtime}"
    )


########## Main co-simulation loop ########################################
# As long as granted time is in the time range to be simulated...
while grantedtime < total_interval:

    # Time request for the next physical interval to be simulated
    requested_time = grantedtime + update_interval
    logger.debug(f"Requesting time {requested_time}")
    grantedtime = h.helicsFederateRequestTime(fed, requested_time)
    logger.debug(f"Granted time {grantedtime}")
```

Once inside the time loop, information is requested and sent between federates at each time step. In the Base Example, the federates first request information from the interfaces to which they have subscribed, and then send information from the interfaces from which they publish.

The `Battery.py` federate first asks for voltage information from the interfaces to which it subscribes:

```
# Get the applied charging voltage from the EV
charging_voltage = h.helicsInputGetDouble((subid[0]))
logger.debug(
    f"\tReceived voltage {charging_voltage:.2f} from input"
    f" {h.helicsSubscriptionGetKey(subid[0])}"
)
```

And then (after doing some internal calculations) publishes the charging current of the battery at its publication interface:

```
# Publish out charging current
h.helicsPublicationPublishDouble(pubid[j], charging_current)
logger.debug(f"\tPublished {pub_name[j]} with value " f"{charging_current:.2f}")
```

Meanwhile, the `Charger.py` federate asks for charging current from the interfaces to which it subscribes:

```
charging_current[j] = h.helicsInputGetDouble((subid[j]))
logger.debug(
    f"\tCharging current: {charging_current[j]:.2f} from "
    f"input {h.helicsSubscriptionGetKey(subid[j])}"
)
```

And publishes the charging voltage at its publication interface:

```
# Publish updated charging voltage
h.helicsPublicationPublishDouble(pubid[j], charging_voltage[j])
logger.debug(
    f"\tPublishing charging voltage of {charging_voltage[j]} " f" at time {grantedtime}"
)
```

After all the time steps have completed, it's good practice to finalize the co-simulation by freeing the federates and closing the HELICS libraries:

```
status = h.helicsFederateFinalize(fed)
h.helicsFederateFree(fed)
h.helicsCloseLibrary()
```

2.4.2 Advanced Examples

Default Advanced Example

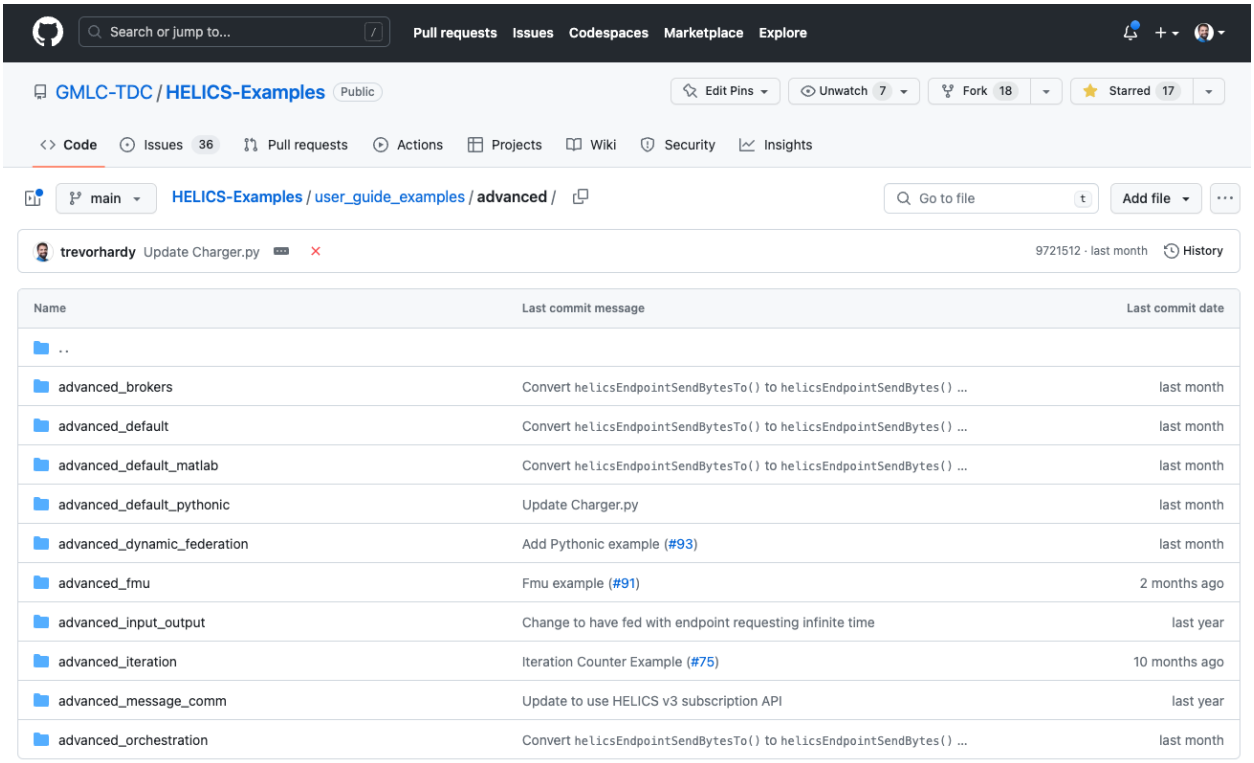
The Advanced Base example walks through a HELICS co-simulation between three python federates, one of each type: value federate (`Battery.py`), message federate (`Controller.py`), and combination federate (`Charger.py`). This serves as the starting point for many of the other advanced examples and is an extension of the *Base Example.*

The Advanced Base Example tutorial is organized as follows:

- *Example files*
- *Co-simulation Setup*
  - *Messages + Values*
  - *Co-simulation Execution and Results*
- *Questions and Help*

Example files

All files necessary to run the Advanced Base Example can be found in the Advanced Examples repository:



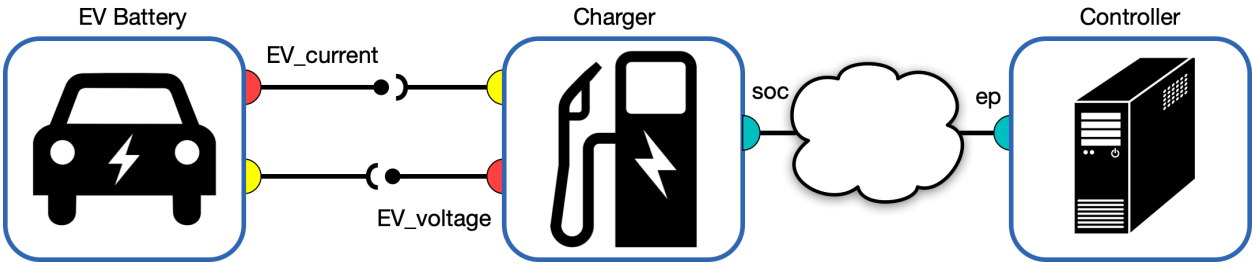
The files include:

- Python program and configuration JSON for Battery federate
- Python program and configuration JSON for Charger federate
- Python program and configuration JSON for Controller federate
- HELICS runner JSON to enable execution of the co-simulation

Co-simulation Setup

Messages and Values

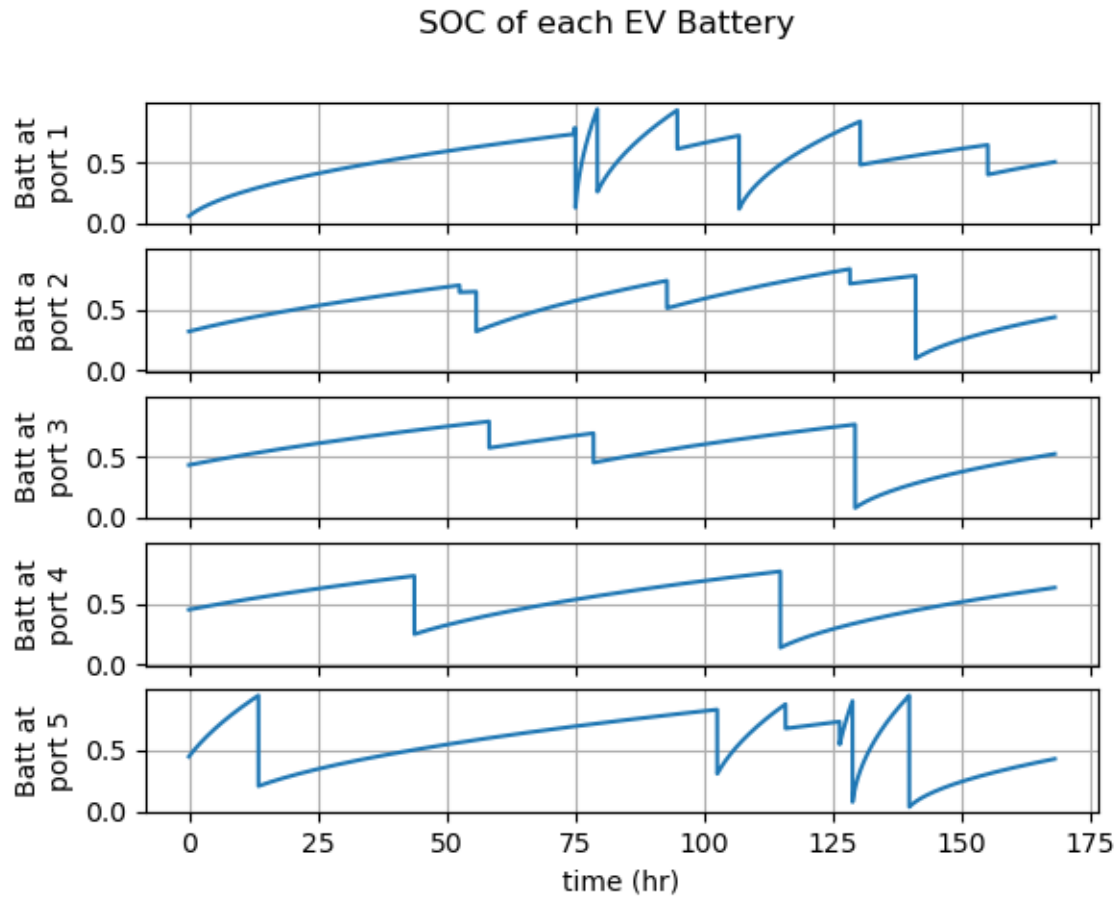
As you may or may not have read in the *User Guide*, one of the key differences between value exchange and the message exchange is that value exchange paths are defined once the federation has been initialized but message exchanges are dynamic and can travel from any endpoint to any endpoint throughout the co-simulation. The diagram below shows the three federates used in this example with the representative interfaces for both the value and message exchanges.



Co-simulation Execution and Results

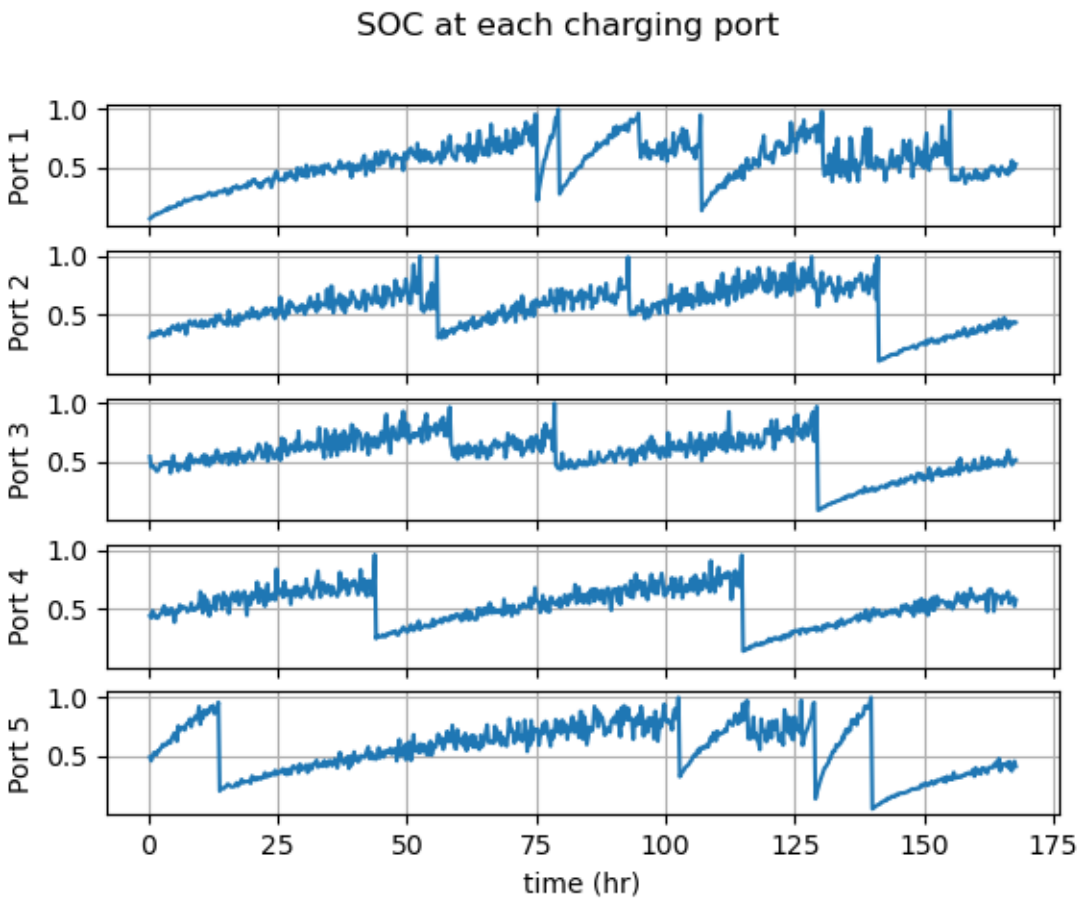
As in the *Fundamental Base Example*, the HELICS runner is used to launch the co-simulation:

```
> helics run --path=advanced_default_runner.json
```



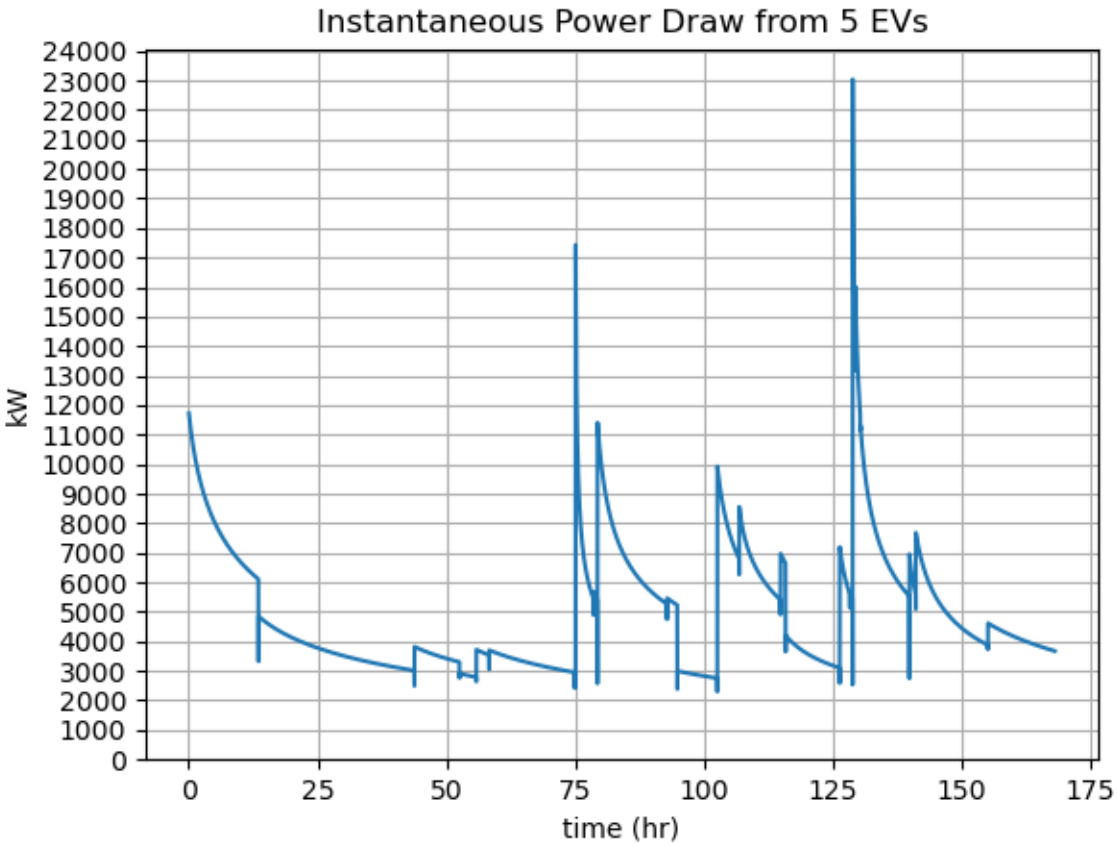
This is the view of each battery as it is charged and two things are immediately obvious:

1. The impact of the charging level is pronounced. The first Batt1 takes almost half the simulation to charge but when its replacement is placed on the charger, it starts at a similar SOC but charges in a fraction of the time. The impact of the charging power supported by each EV is significant.
2. Most of the batteries fail to reach 100% SOC, some dramatically so. This is due to the current measurement error leading to a mis-estimate of SOC and thus premature termination of the charging. This can be seen the following graph



As previously mentioned, the current measurement noise is a function of the total magnitude of the current and thus as the battery charges up and the current draw drops, the noise in the measurement becomes a bigger fraction of the overall value. This results in the noisiest SOC estimates at higher SOC values. This is clearly seen in the EV1 value that starts the co-simulation relatively smooth and steadily increases in noisiness.

This graph also clearly shows that each EV was estimated to have a 100% SOC when the charging was terminated even though we know from the previous graph that full charge had not been reached.



The data shown in the power graph is arguably the point of the analysis. It shows our maximum charging power for this simulated time as 80 kW. If this is the only simulation result we have, we would be inclined to use this as a design value for our electricity delivery infrastructure. More nuanced views could be had, though, by:

1. Running this co-simulation multiple times using a different random seed to see if 80 kW is truly the maximum power draw. (We do a version of this in an *example demonstrating how to run multiple HELICS co-simulations simultaneously* on a single compute node.)
2. Plotting the charging power as a histogram to get a better understanding of the distribution of the instantaneous charging power. (We also do this as part of our *example on using an orchestration tool to use HELICS co-simulations as part of a more complex analysis*.)

Questions and Help

Do you have questions about HELICS or need help?

1. Come to office hours!
2. Post on the gitter!
3. Place your question on the github forum!

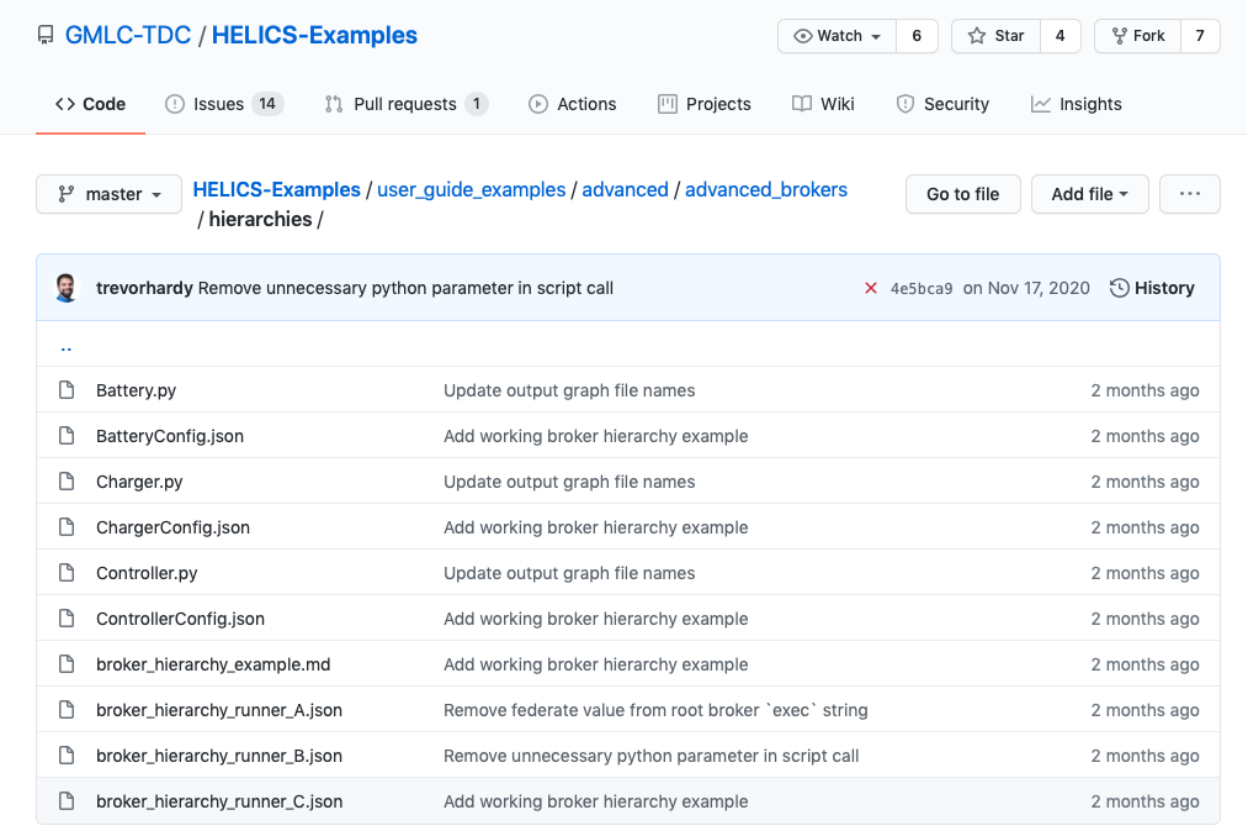
Brokers - Simultaneous Co-simulations

This example shows how to configure a HELICS co-simulation so that multiple co-simulations can run simultaneously on one computer. Understanding this configuration is a pre-requisite to running the other advanced broker examples (which also involve multiple brokers running on one computer).

- *Where is the code?*
- *What is this Co-simulation doing?*
  - *Differences Compared to the Fundamental Examples*
    - *HELICS Differences*
    - *Research Question Complexity Differences*
- *Execution and Results*

Where is the code?

The code for the Advanced examples can be found in the HELICS-Examples repository on GitHub. The example on simultaneous co-simulations can be found here. If you have issues navigating the examples, visit the HELICS Gitter page or the user forum on GitHub.



What is this co-simulation doing?

Using the exact same federates as in the *Advanced Default example*, the same co-simulation is run multiple times (simultaneously) with different random number generator seeds. The example both demonstrates how to run multiple HELICS co-simulations simultaneously on one computer without the messages between federates getting mixed up, but also shows a simple way to do sensitivity analysis. A better way is shown later in the *orchestration example.*

Differences compared to the advanced default example

Two primary changes:

1. This example contains a set of co-simulations with each instance using a different random number generator seed in Battery.py

```
if __name__ == "__main__":
    np.random.seed(2608)
```

and Charger.py

```
if __name__ == "__main__":
    np.random.seed(1490)
```

The values shown above are from federation_1. Identical lines with alternative values can be found in federation_2 and federation_3.

2. The brokers are configured to ensure that messages from one federation do not get routed to federates in another federation.

HELICS differences

With no extra configuration, it is only possible to run one HELICS co-simulation on a given computer. In the most popular HELICS cores (ZMQ being the most common, by far), messages are sent between federates using the networking stack. (There are other ways, though. For example, the IPC core uses the Boost library inter-process communication.). If you want to run multiple co-simulations on one compute need, an extra step needs to be taken to keep the messages from each federation separate from each other and non-interfering. Since we're using the network stack, this can be easily accomplished by assigning each broker a unique port to use. Looking at the federation launch config files, you can see this clearly expressed:

federation_1_runner.json

```
{
  "federates": [
    {
      "directory": ".",
      "exec": "helics_broker -f 3 --loglevel=1 --port=20100",
      "host": "localhost",
      "name": "broker"
    },
```

federation_2_runner.json

```
{
  "federates": [
    {
      "directory": ".",
      "exec": "helics_broker -f 3 --loglevel=1 --port=20200",
      "host": "localhost",
      "name": "broker"
    },
```

federation_3_runner.json

```
{
  "federates": [
    {
      "directory": ".",
      "exec": "helics_broker -f 3 --loglevel=1 --port=20300",
      "host": "localhost",
      "name": "broker"
    },
```

Research question complexity differences

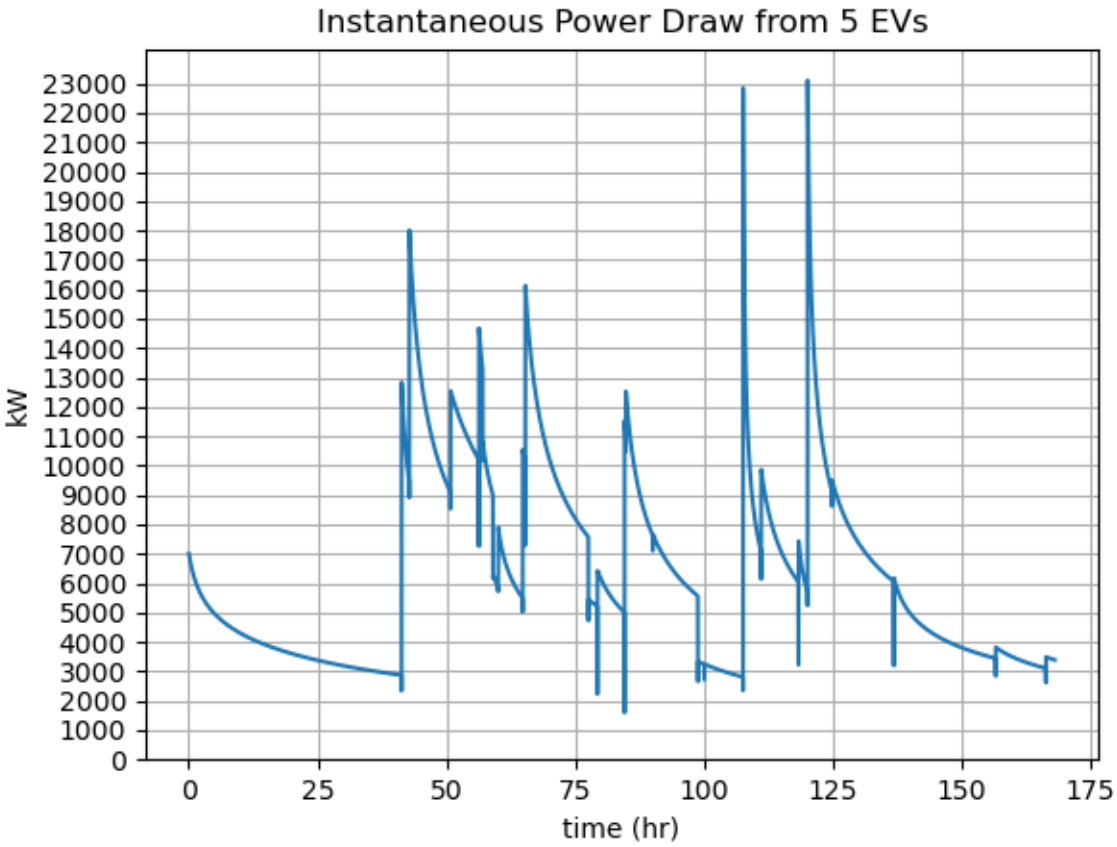
The Advanced Default example uses a random number generator to determine things like initial state-of-charge of the battery and charging power of the individual EV batteries. These factors have an impact on the charging duration for each EV battery and the peak charging power seen over the duration of the simulation. Since the later is *the* key metric of the simulation experiment, there is strong motivation to vary the seed value for the random number generator to expand the range of results, effectively increasing the sample size. These co-simulations could each be run serially but assuming the computer in question has the horsepower, there's no reason not to run them in parallel.

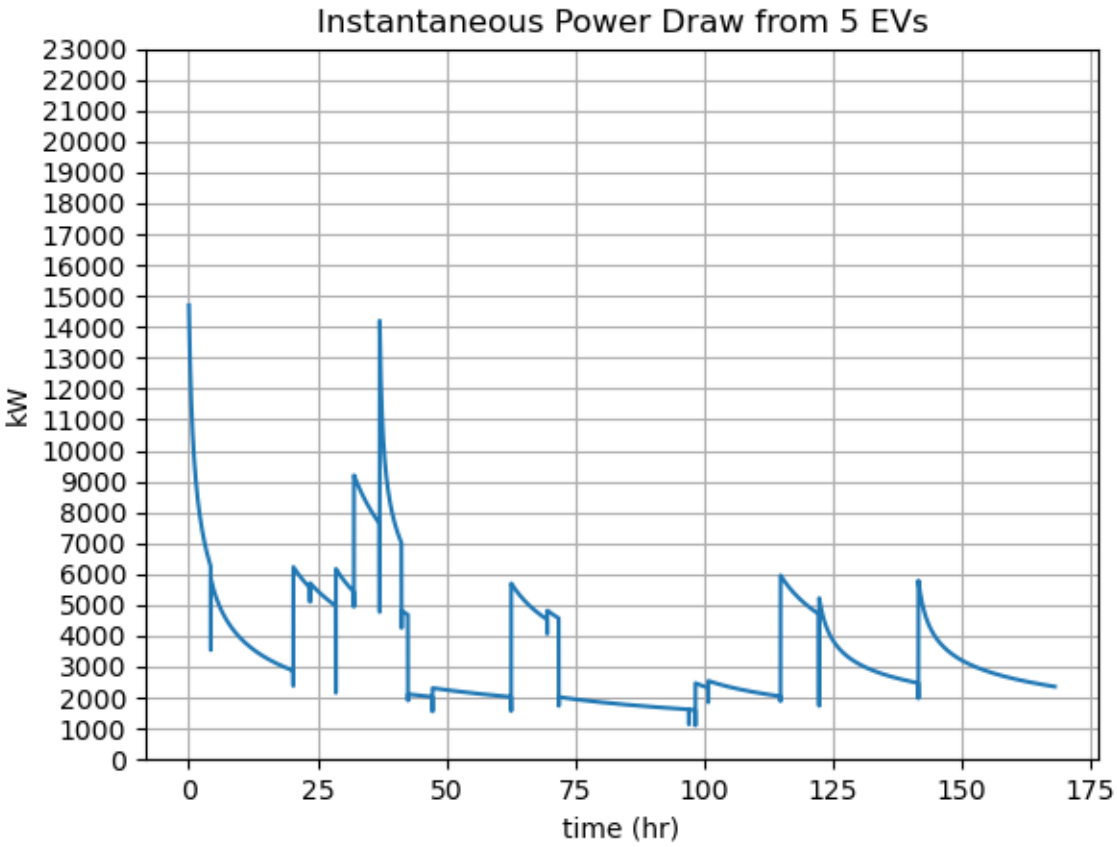
Execution and Results

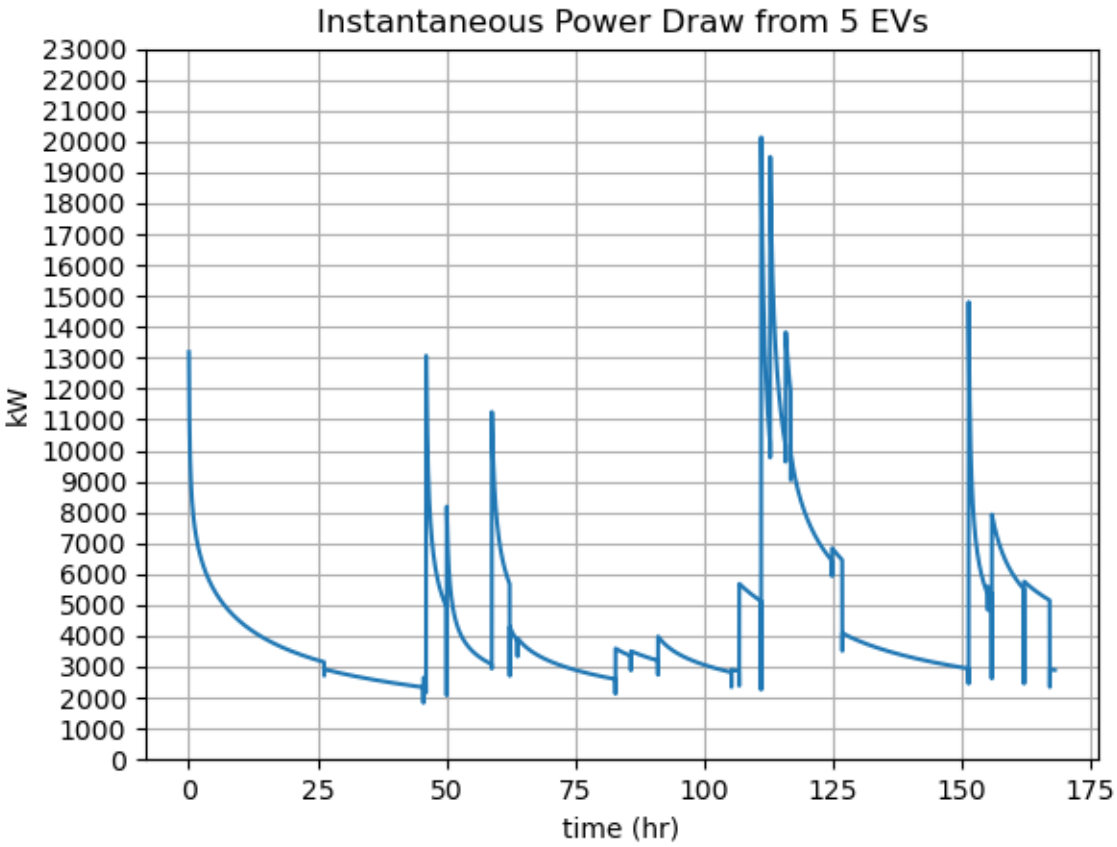
To run the co-simulations simultaneously, all that is required is having the HELICS runner launch each individually. The trailing & in the shell commands below background the command and return another shell prompt to the user.

```
$ helics run --path=./federation_1/federation_1_runner.json &       $ helics run --path=.
/federation_2/federation_2_runner.json &              $ helics run --path=./federation_2/
federation_3_runner.json &
```

The peak charging results are shown below. As can be seen, the peak power amplitude and the total time at peak power are impacted by the random number generator seed.


Instantaneous Power Draw from 5 EVs
kW
time (hr)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
12000
13000
14000
15000
16000
17000
18000
19000
20000
21000
22000
23000
0
25
50
75
100
125
150
175


Instantaneous Power Draw from 5 EVs
kW
time (hr)
23000
22000
21000
20000
19000
18000
17000
16000
15000
14000
13000
12000
11000
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0
25
50
75
100
125
150
175



To do a more legitimate sensitivity analysis to the population of EVs that are being charged, a sample size larger than three is almost certainly necessary. *We've put together another example* to show how to orchestrate running larger sets of co-simulations to address exactly these kinds of needs.

Questions and Help

Do you have questions about HELICS or need help?

1. Come to office hours!
2. Post on the gitter!
3. Place your question on the github forum!

Brokers - Hierarchies

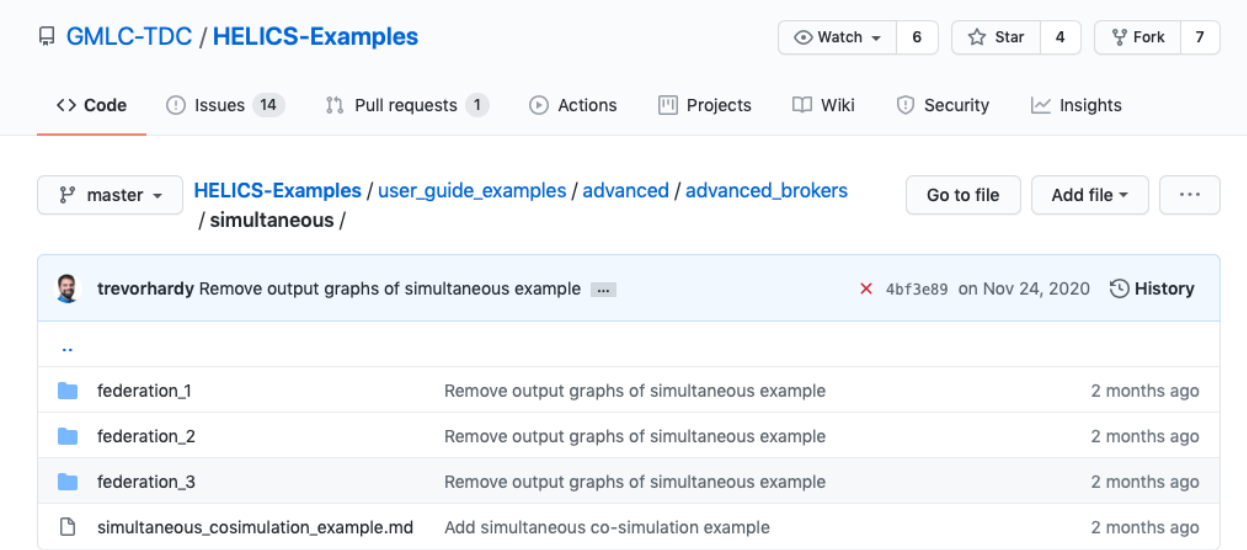
This example shows how to configure a HELICS co-simulation to allow the use of multiple brokers in a single co-simulation.

- *Where is the code?*
- *What is this Co-simulation doing?*
  - *Differences Compared to the Advanced Default Example*
    - *HELICS Differences*
  - *HELICS Components*

- *Execution and Results*

Where is the code?

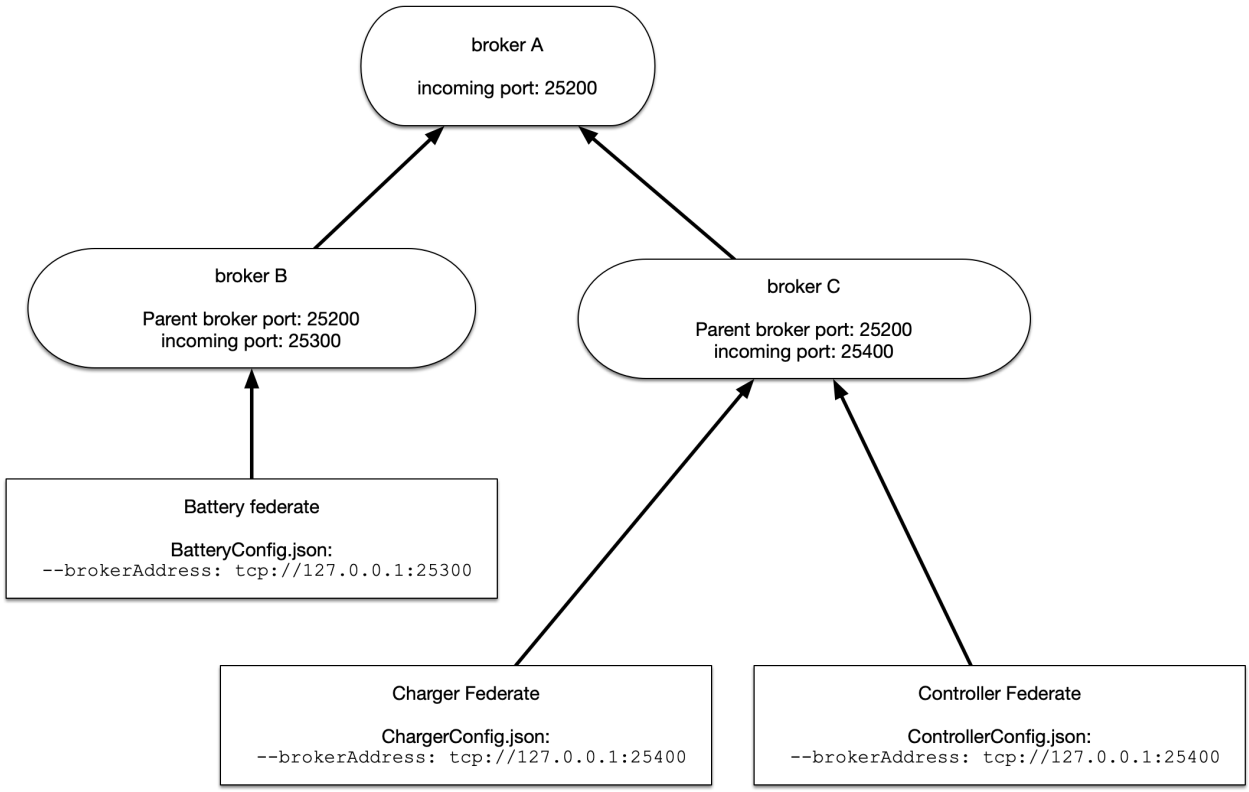
The code for the Advanced examples can be found in the HELICS-Examples repository on GitHub. The example on broker hierarchies can be found here. If you have issues navigating the examples, visit the HELICS Gitter page or the user forum on GitHub.



What is this co-simulation doing?

This example shows you how to configure a co-simulation to take advantage of multiple brokers. Though we'll be running this example on a single computer, the application of broker hierarchies is more common when running a co-simulation across multiple computers.

Differences compared to Advanced Default example



As will be shown, the use of multiple brokers will not affect the results of the co-simulation.

HELICS differences

Broker hierarchies are primarily used to help improve the performance of the co-simulation by allowing federates that interact strongly with each other to run on a single compute node, thereby allowing them to exchange information with each other quickly rather than over a relatively slow network connection to the rest of the federates on other compute nodes. This can be particularly helpful when the other compute nodes reside at off-site locations and the co-simulation communication is taking place between them over the public internet. (See the *User Guide section on broker hierarchies* for further details.)

Not all federations lend themselves to segregation like this; the example here doesn't really support such segregation as both the Battery and the Controller talk frequently with the Charger.

HELICS components

When implementing across compute nodes, the configuration is simpler will be simpler than in this example because the need to segregate the federates and brokers is only a function of IP address where HELICS can use the default port number on each compute node. To get this example to run on a single computer, the hierarchy must be implemented through the use of specific port numbers for specific brokers.

When running across multiple compute nodes, the relevant portion of the HELICS runner files would look like this:

`broker_hierarchy_runner_A.json`

```
"exec": "helics_broker --loglevel=7 --timeout='10s' ",
```

`broker_hierarchy_runner_B.json` and `broker_hierarchy_runner_C.json`

```
"exec": "helics_broker -f <number of federates> --loglevel=7 --timeout='10s' --broker_
↪address=tcp://<IP address of broker A>",
```

Additionally, the federates would not need `brokerAddress` in their configuration since they would look on their local node for a broker by default.

Since this example *was* made to run on a single computer, we use the port number to segregate the federates and broker. `--port=` is use to define the port number on which the broker looks for connections to federates and `--broker_address=` is used to define the IP address with port of the parent broker. (The loopback address of 127.0.0.1 is used to look for the broker on the same node.)

Additionally, each federate has to define the broker to which it is attempting to connect by including the `brokerAddress` parameter in its own configuration; this allows for the definition of the port number the federate should use to connect to it's broker:

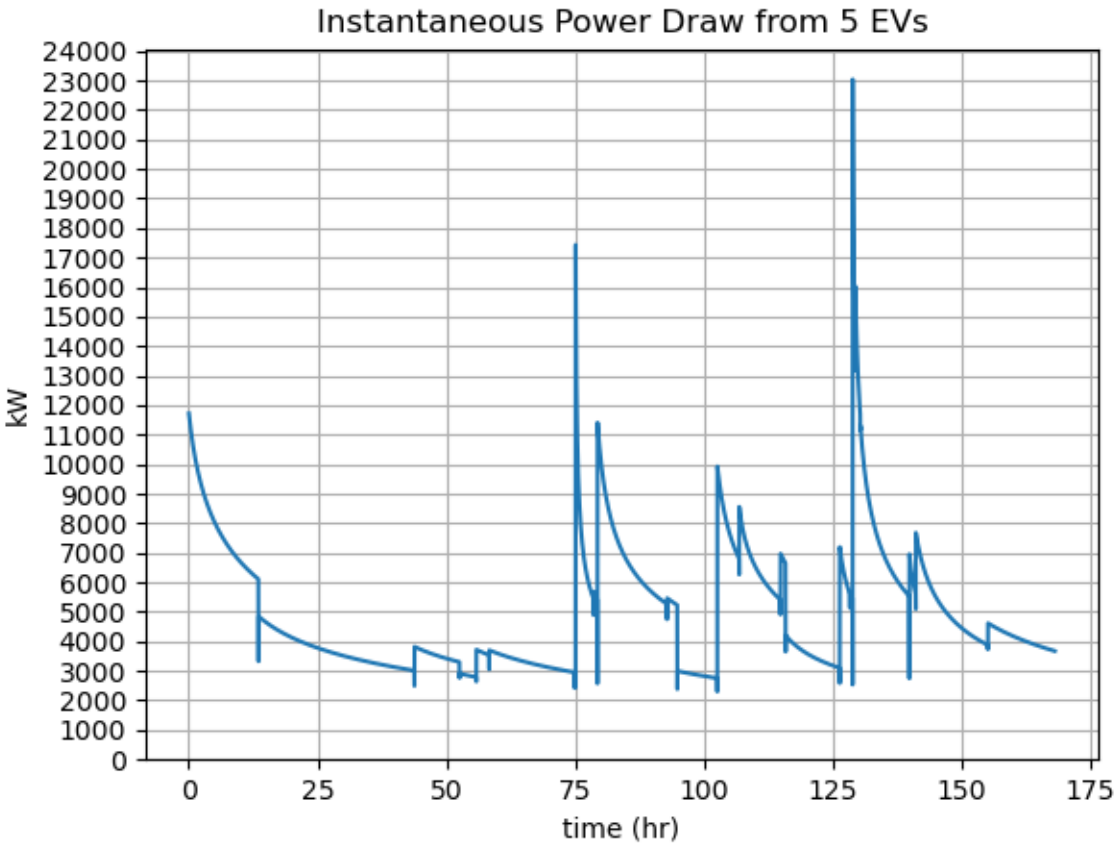
```
"brokerAddress": "tcp://127.0.0.1:25300",
```

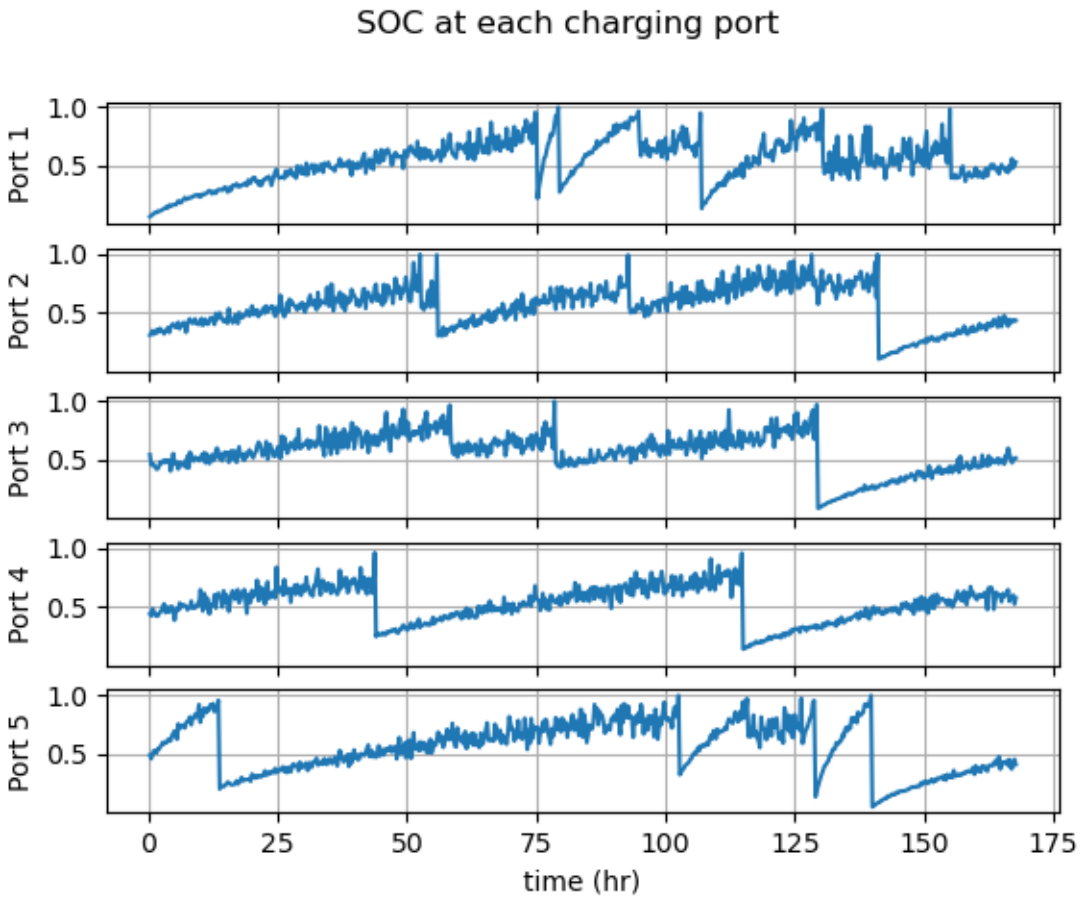
Execution and Results

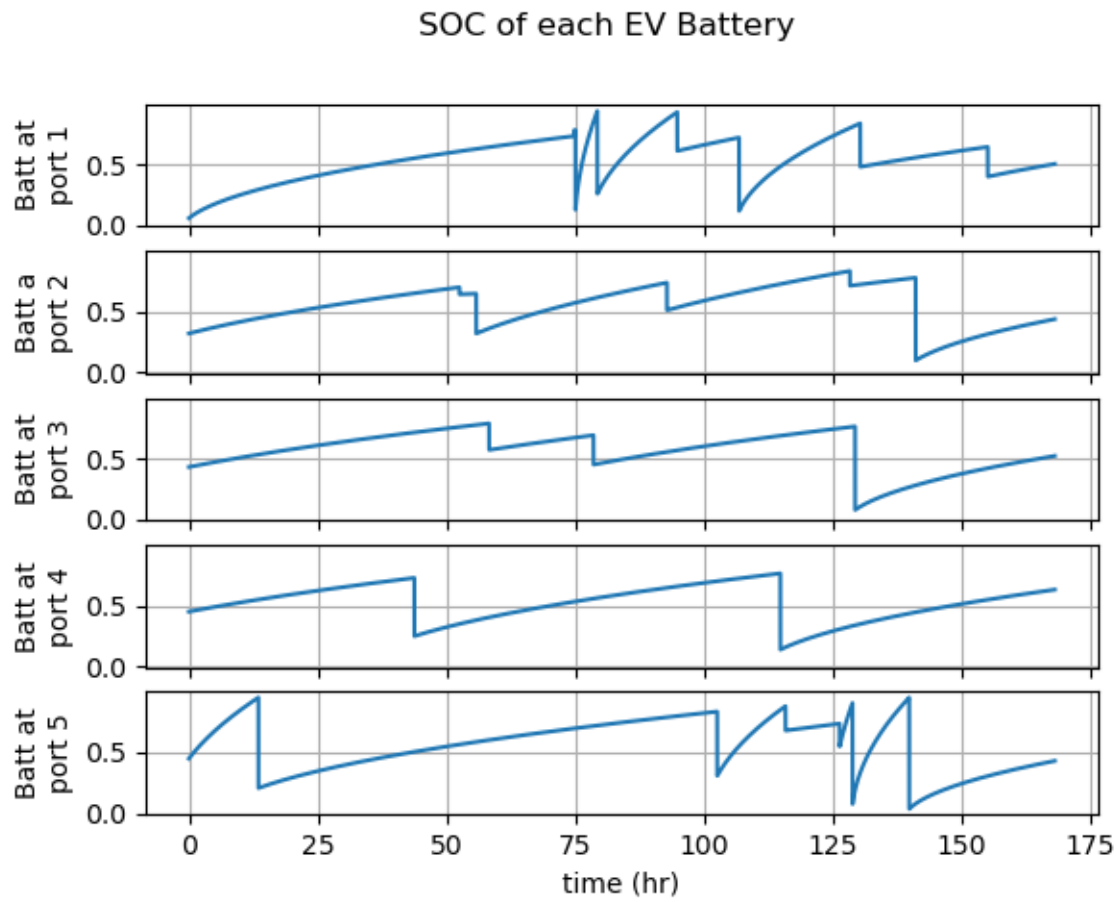
Since this example requires three brokers and their respective federates to run simultaneously, `helic_cli` will be used to launch the three sets of brokers and federates, just like the in *simultaneous co-simulation example*

- `$ helics run --path=./broker_hierarchy_runner_A.json &`
- `$ helics run --path=./broker_hierarchy_runner_B.json &`
- `$ helics run --path=./broker_hierarchy_runner_C.json &`

The peak charging results are shown below. As can be seen, the peak power amplitude and the total time at peak power are impacted by the random number generator seed.


Instantaneous Power Draw from 5 EVs
kW
time (hr)
24000
23000
22000
21000
20000
19000
18000
17000
16000
15000
14000
13000
12000
11000
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0
25
50
75
100
125
150
175


SOC at each charging port
Port 1
Port 2
Port 3
Port 4
Port 5
1.0
0.5
0
25
50
75
100
125
150
175
time (hr)



These results are identical to those in the base *Advanced Default example*; this is expected as only the structure of the co-simulation has been changed and not any of the federate code itself.

Questions and Help

Do you have questions about HELICS or need help?

1. Come to office hours!
2. Post on the gitter!
3. Place your question on the github forum!

Brokers - Multi-Protocol Brokers

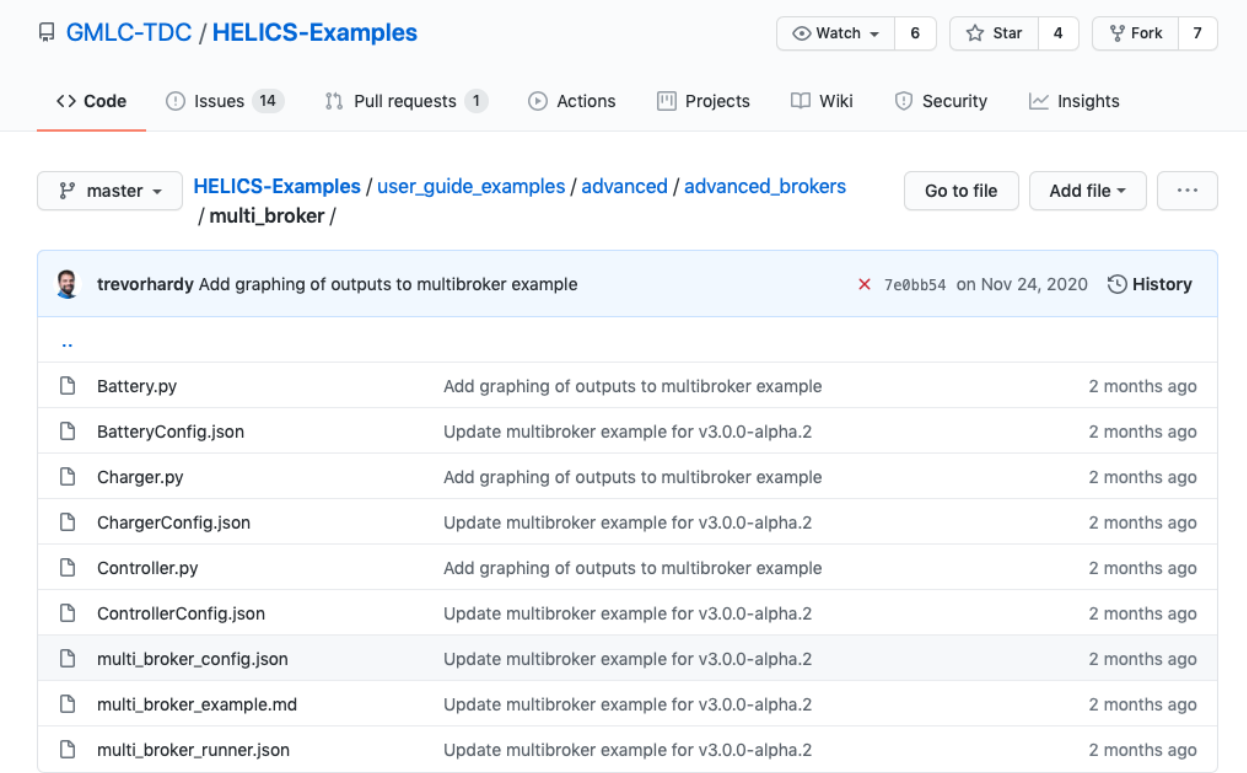
This example shows how to configure a HELICS co-simulation to implement a broker structure that utilizes multiple core types in a single co-simulation. Typically, all federates in a single federation use the same core type (ZMQ by default) but HELICS can be set up to utilize different core types in the same federation

- *Where is the code?*
- *What is this Co-simulation doing?*
  - *Differences Compared to the Fundamental Examples*
    - *HELICS Differences*
  - *HELICS Components*

- *Execution and Results*

Where is the code?

The example on multiple brokers can be found here. If you have issues navigating the examples, visit the HELICS Gitter page or the user forum on GitHub.



What is this co-simulation doing?

This example shows you how to configure a co-simulation to use more than one core type in the same federation. The example itself has the same functionality as the Advanced Default example as the only change is a structural to the federation and not the federate code itself.

Differences compared to the Advanced Default example

For this example, the *Advanced Default example* has been split up so that each federate uses a different core type in a single federation.

HELICS differences

Typically, all federates in a federation use the same core type. There can be cases, though, where a multi-site co-simulation with a more complex networking environment or performance requirements dictate the need for some federates to utilize a difference core type than others. For example, the `IPC` core utilizes a Boost library function to allow two executables both using Boost to communicate between themselves when running on the same compute node; since this is in-memory communication rather than over the network stack, performance is expected to be higher. It could be that a particular federation has been optimized to take advantage of this but must also communicate with federates on a separate compute node via ZMQ. In this case, a so-called "multibroker" can be configured to allow for the federation to run. (See the User Guide section on the *multi-protocol broker* and *broker core types* for further details.)

In this example, we won't be doing anything like that but, for demonstration purposes, simply using the same federation from the *Advanced Default example.* and configuring it so each federate uses a different core type.

HELICS Components

To configure a multibroker, the broker configuration line is slightly extended from a traditional federation. From the HELICS runner configuration file `multi_broker_runner.json`

```
"exec": "helics_broker -f 3 --coreType=multi --config=multi_broker_config.json --
↪name=root_broker",
```

The `coreType` of the broker is set to `multi` and a configuration file is specified. That file looks like this:

```
{
  "master": {
    "coreType": "test"
  },
  "comms": [
    {
      "coreType": "zmq",
      "port": 23500
    },
    {
      "coreType": "tcp",
      "port": 23700
    },
    {
      "coreType": "udp",
      "port": 23900
    }
  ]
}
```

The first and most important note: `master` and `comms` are reserved words in this context and **MUST** be used. The `master` core type must be `test` but the core types for the federates can be any of the supported cores. Again, as in *other similar* examples, because we are running this on a single compute node, the port for each core type must be specified and the federates using those core types need to have the `brokerPort` property set to the corresponding core's port number.

BatteryConfig.json

```
  "name": "Battery",
  "loglevel": 1,
  "coreType": "zmq",
  "brokerPort": 23500,
```

ChargerConfig.json

```
  "name": "Charger",
  "loglevel": 1,
  "coreType": "tcp",
  "brokerPort": 23700,
```

ControllerConfig.json

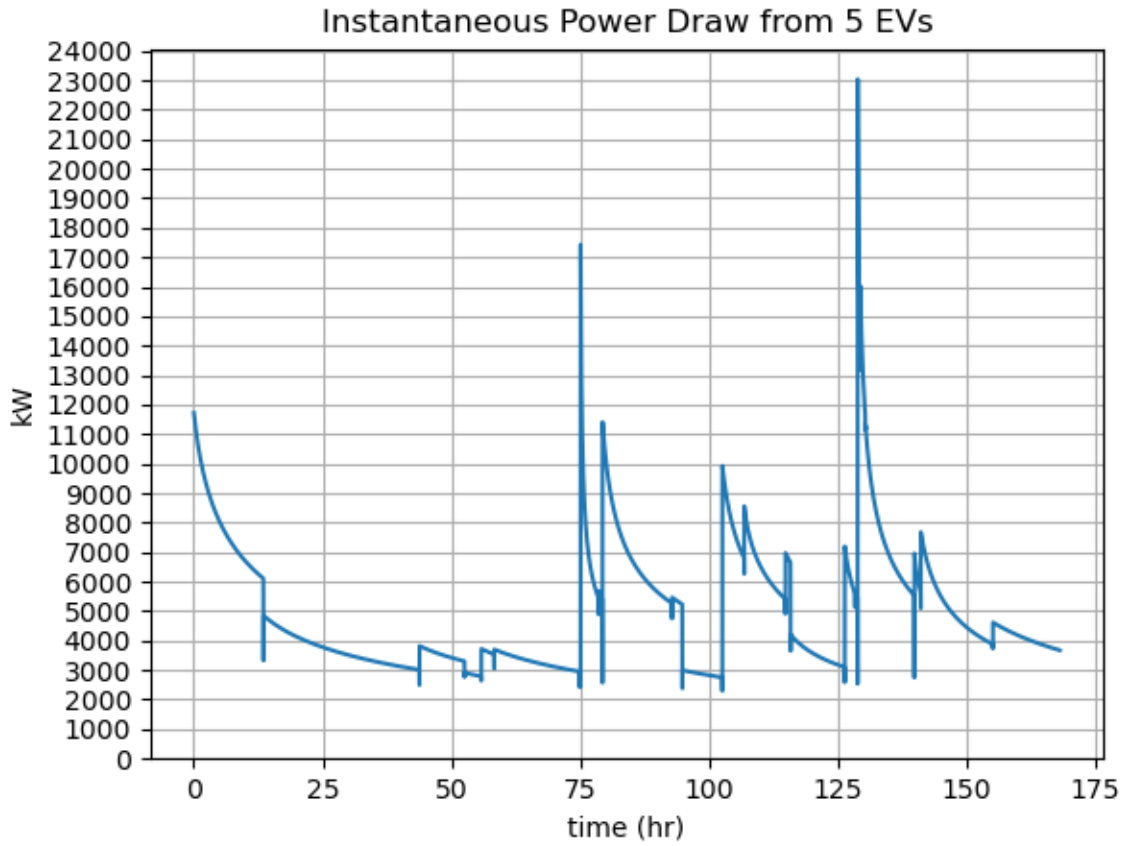
```
  "name": "Controller",
  "loglevel": 1,
  "coreType": "udp",
  "brokerPort": 23900,
```

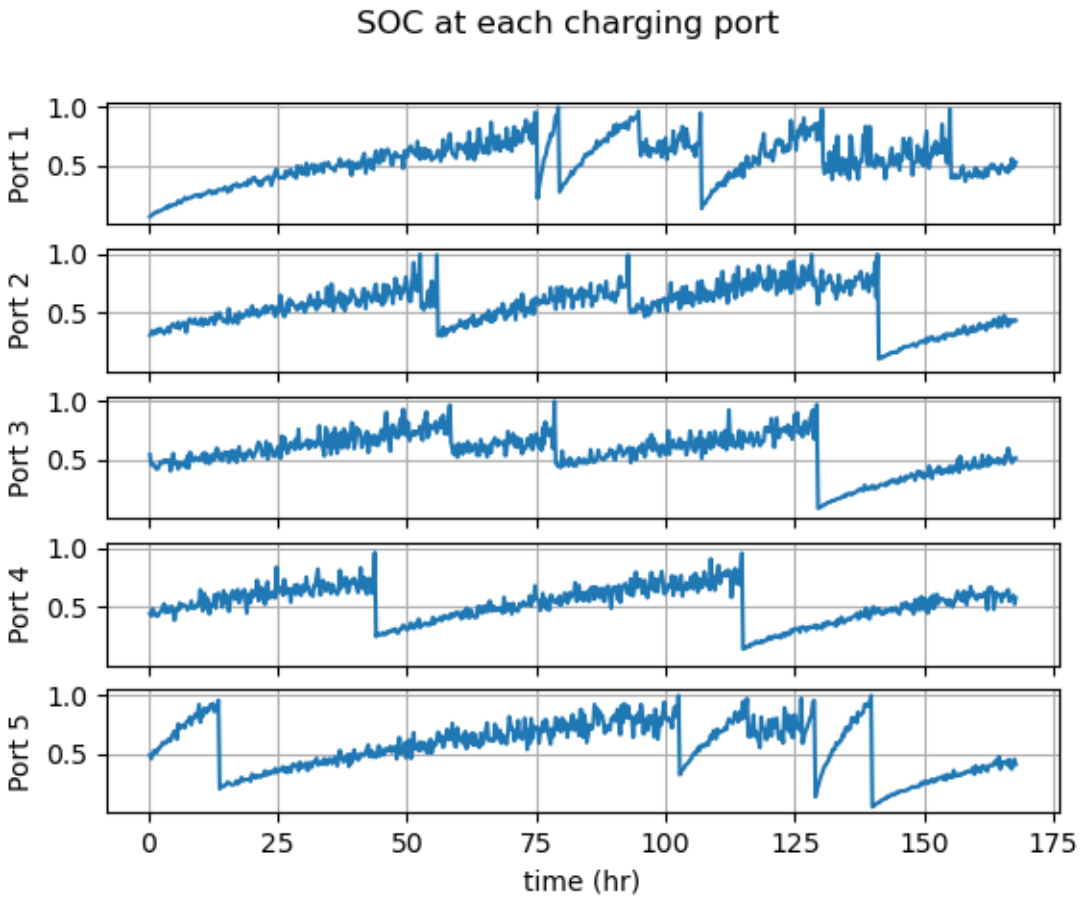
Execution and Results

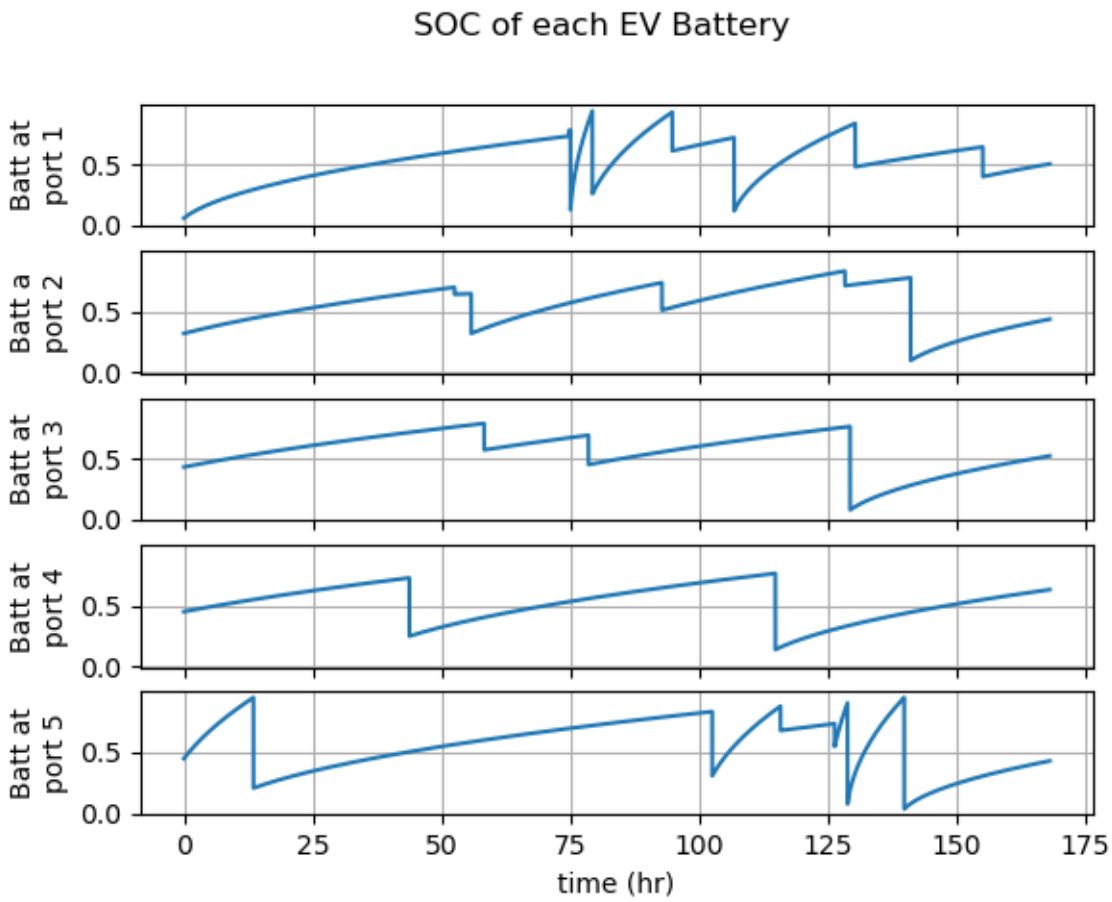
Unlike the other advanced broker examples, this one can be run with a single HELICS runner command:

```
$ helics run --path=./multi_broker_runner.json
```

As has been mentioned, since this is just a change to the co-simulation architecture, the results are identical to those in the *Advanced Default example*.




SOC at each charging port
Port 1
Port 2
Port 3
Port 4
Port 5
1.0
0.5
0
25
50
75
100
125
150
175
time (hr)



Questions and Help

Do you have questions about HELICS or need help?

1. Come to office hours!
2. Post on the gitter!
3. Place your question on the github forum!

Brokers - Multi-computer Co-simulation

This example shows how to configure a HELICS co-simulation to run across multiple computers.

- *Where is the code?*
- *What is this Co-simulation doing?*
  - *Differences Compared to the Advanced Default Example*
    - *HELICS Differences*
  - *HELICS Components*
- *Execution and Results*

Where is the code?

The code for the Advanced examples can be found in the HELICS-Examples repository on GitHub. The example on multi-computer co-simulation can be found here. If you have issues navigating the examples, visit the HELICS Gitter page or the user forum on GitHub.

What is this co-simulation doing?

This example shows you how to configure a co-simulation to take advantage of multiple brokers. Though we'll be running this example on a single computer, the application of broker hierarchies is more common when running a co-simulation across multiple computers.

Differences compared to Advanced Default example

As will be shown, the use of multiple computers to run the co-simulation will not affect the results of the co-simulation.

HELICS differences

Running a HELICS co-simulation across multiple computers is useful in situations when a federate is too large to fit on the same computer node as the rest of the federates, uses proprietary software that can't be installed on the same computer as the rest of the federation, or there are other limitations that require the federation to be split across multiple computers. (See the *User Guide section on broker hierarchies* for further details.)

HELICS components

When running across multiple compute nodes, the relevant portion of the broker instantiation looks like this on the computer where broker is running:

```
"$ helics_broker --loglevel=debug --timeout='10s' --ipv4",
```

The "ipv4" flag opens up an externally accessible ports (by default, 23405) on externally facing network interfaces with an ipv4 address.

On the computer(s) where the broker is not running, each of the federates has to define the "broker_address" as part of the configuration. In this case, that's happening in a JSON configuration file and as such we just need to add a single line to the file:

```
"broker_address": "tcp://<IP address of broker>",
```

If for whatever reason the federation needs to run on a different port, this can easily be done with minor alterations. The broker instantiation would be:

```
$ helics_broker --loglevel=debug --timeout='10s' --ipv4 --port=<port number>",
```

All federates would also need to know to use the new port number. For those that have already specified the IP address of the broker, appending the port in standard networking syntax works:

```
"brokerAddress": "tcp://<IP address>:<port number>",
```

Alternatively, the "broker_port" option can be set:

```
"broker_port": "<port number>",
```

Note that any federates running on the same machine as the broker would also need to set this port option.

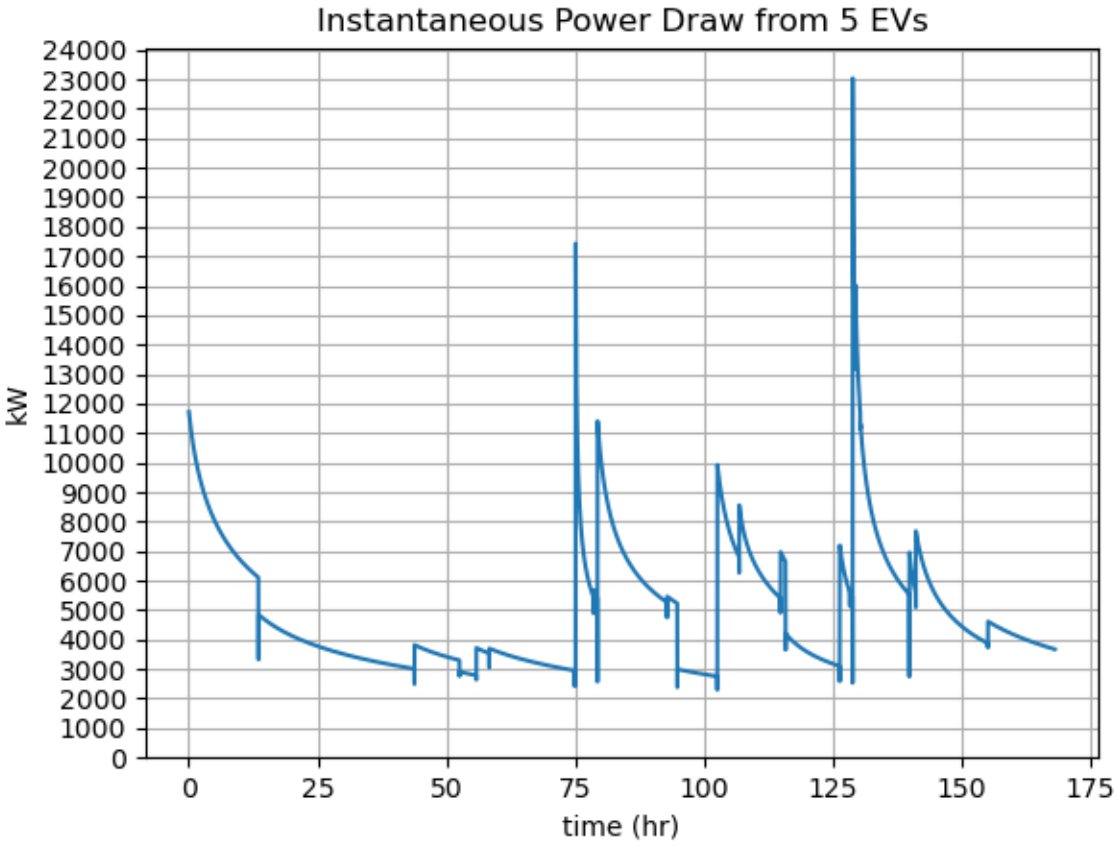
Execution and Results

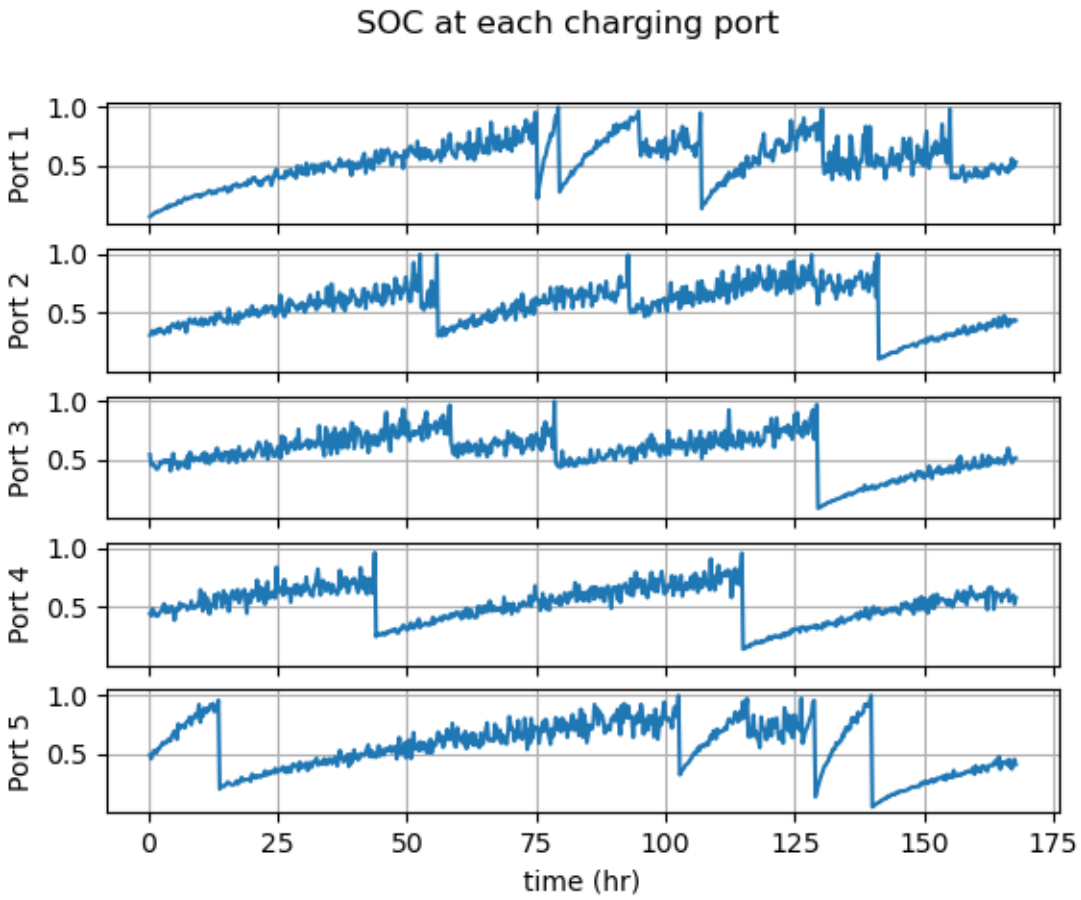
To run this example you'll need to use two computers, running one part of the federation on each. Each computer can launch its part of the federation with the following HELICS runner commands

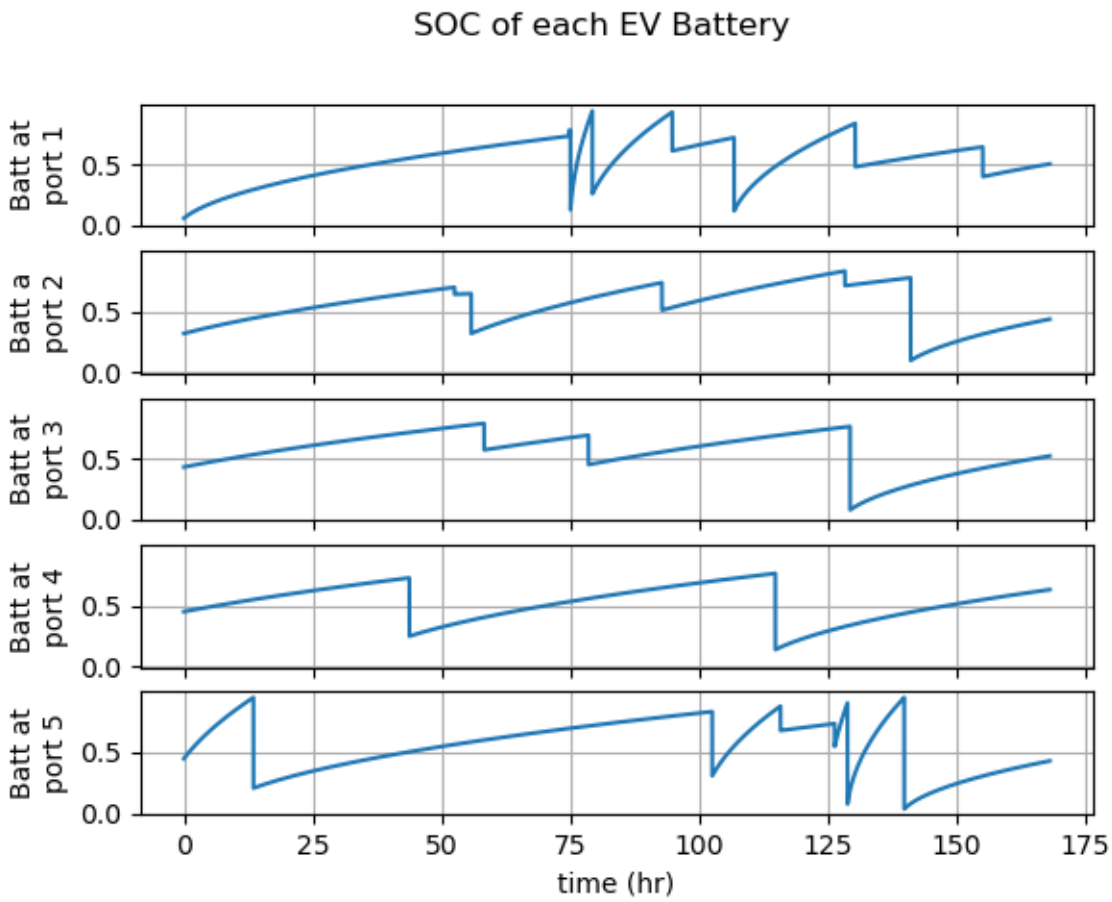
```
$ helics run --path=./multi_computer_1_runner.json
```

```
$ helics run --path=./multi_computer_2_runner.json
```

The peak charging results are shown below and are identical to the results from the similarly configured *broker hierarchy example*.




SOC at each charging port
Port 1
Port 2
Port 3
Port 4
Port 5
1.0
0.5
0
25
50
75
100
125
150
175
time (hr)



Questions and Help

Do you have questions about HELICS or need help?

1. Come to office hours!
2. Post on the gitter!
3. Place your question on the github forum!

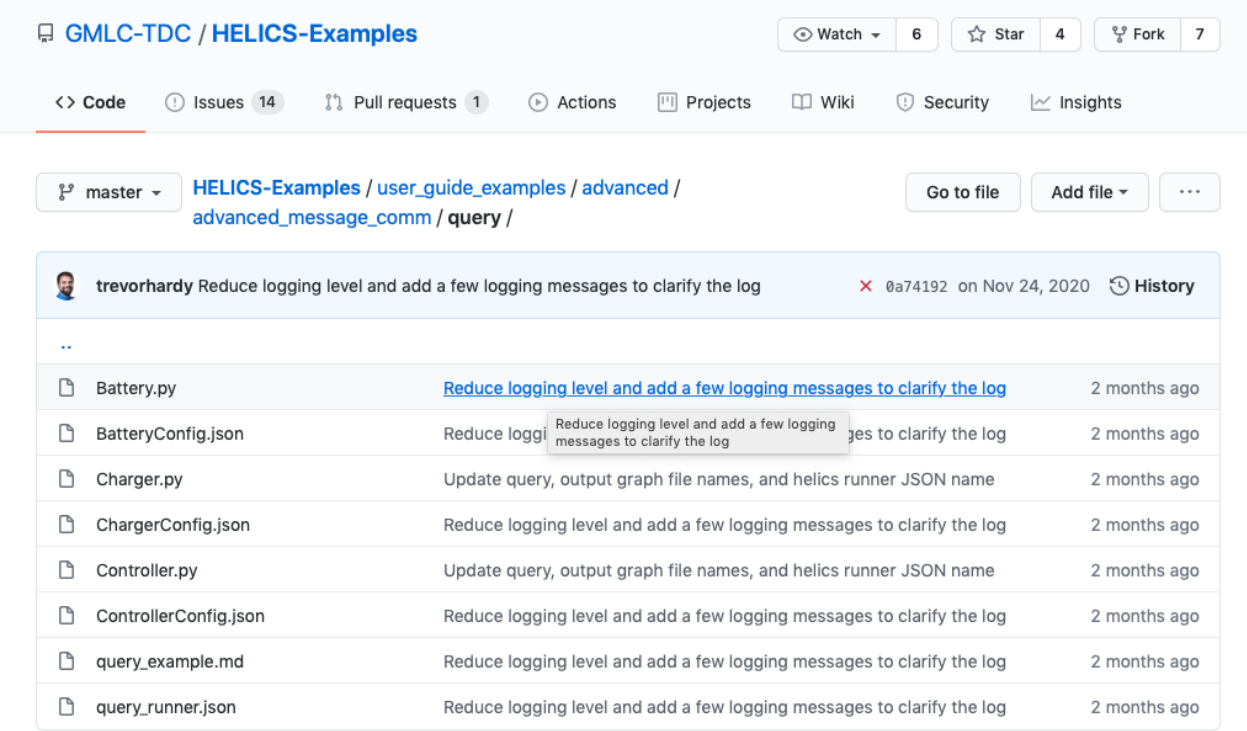
Federation Queries

This demonstrates the use of federation queries and performs dynamic configuration by using the information from the query to configure the Battery federate.

- *Where is the code?*
- *What is this Co-simulation doing?*
  - *Differences Compared to the Fundamental Examples*
    - *HELICS Differences*
  - *HELICS Components*
- *Execution and Results*

Where is the code?

The example on queries can be found here. If you have issues navigating the examples, visit the HELICS Gitter page or the user forum on GitHub.



What is this co-simulation doing?

This example shows how to run queries on a federation and to use the output of the queries to configure a federate. Rather than a static configuration that is defined prior to runtime, this dynamic configuration can be useful for federations that change composition frequently.

Differences compared to the Advanced Default example

This example has the same federates interacting in the same ways as in the *Advanced Default example*. The only difference is the use of queries for dynamic configuration rather than static configuration.

HELICS Differences

In most of the examples presented here, the configuration of the federation is defined prior to executing the co-simulation via the configuration JSON files. It is possible, though, with extra effort and careful design, to write the federate code such that they self-configure based on the other participants in the co-simulation. This example provides a simple demonstration of this by having the Battery federate query the federation and look for the values that the Charger federate is publishing and subscribing to them.

HELICS components

`Battery.py` contains all the changes from the *Advanced Default example* that allow it to perform dynamic configuration. Specifically:

- Sleeping a few seconds to ensure all other federates in the federation have configured so that the Battery federate can query the federation and know it is seeing the comprehensive configuration.

```
sleep_time = 5
logger.debug(f"Sleeping for {sleep_time} seconds")
time.sleep(sleep_time)
```

- `eval_data_flow_graph` is a new function that performs a data graph query on the federation. This query evaluates the connections between federates, showing who is publishing and subscribing to what. The function also takes the output from the query and parses it into a form that can be easily used for the dynamic configuration.

```
def eval_data_flow_graph(fed):
    query = h.helicsCreateQuery("broker", "data_flow_graph")
    graph = h.helicsQueryExecute(query, fed)
```

- The Battery federate subscribes to all the publications from the Charger federate based on the results of the data flow graph.

```
for core in graph["cores"]:
    if core["federates"][0]["name"] == "Charger":
        for pub in core["federates"][0]["publications"]:
            key = pub["key"]
            sub = h.helicsFederateRegisterSubscription(fed, key)
            logger.debug(f"Added subscription {key}")
```

- After making the subscription, the Battery federate re-evaulates the data flow graph and updates its own internal record of the configuration

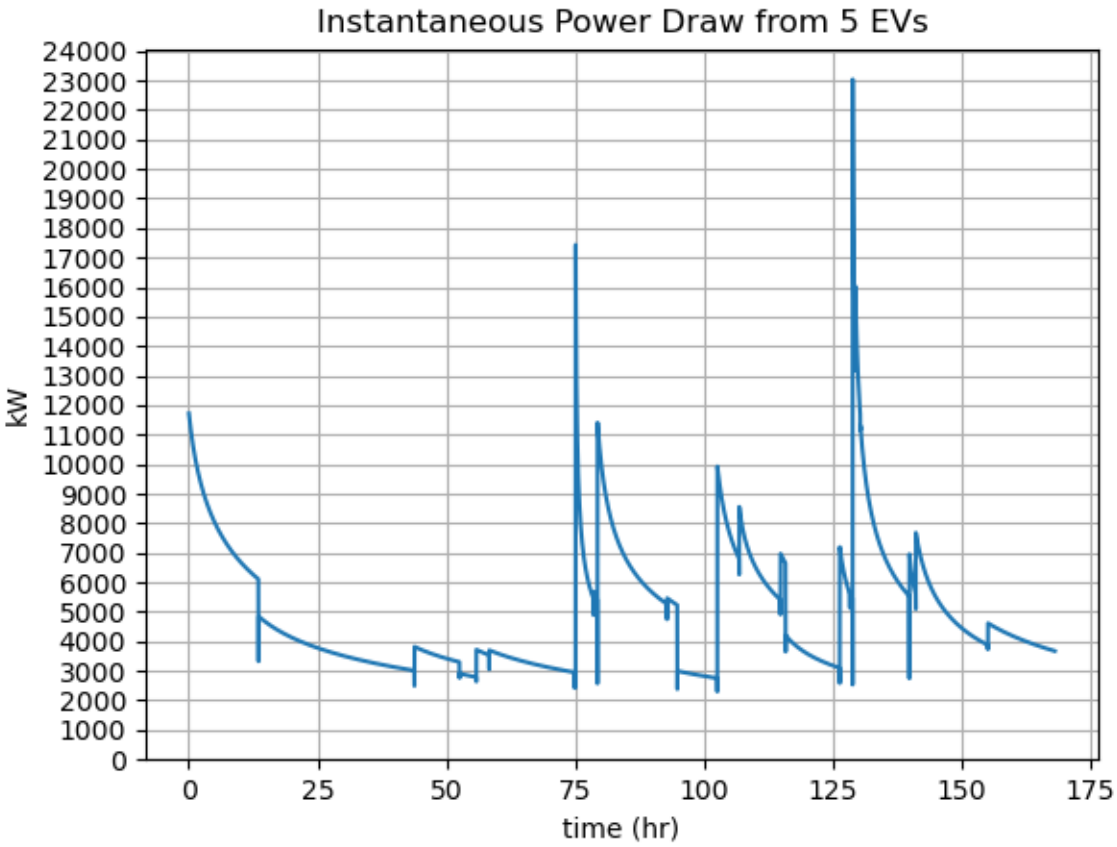
```
# The data flow graph can be a time-intensive query for large
#   federations
# Verifying dynamic configuration worked.
graph, federates_lut, handle_lut = eval_data_flow_graph(fed)
# logger.debug(pp.pformat(graph))
sub_count = h.helicsFederateGetInputCount(fed)
logger.debug(f"Number of subscriptions: {sub_count}")
subid = {}
sub_name = {}
for i in range(0, sub_count):
    subid[i] = h.helicsFederateGetInputByIndex(fed, i)
    sub_name[i] = h.helicsSubscriptionGetKey(subid[i])
    logger.debug(f"\tRegistered subscription---> {sub_name[i]}")
```

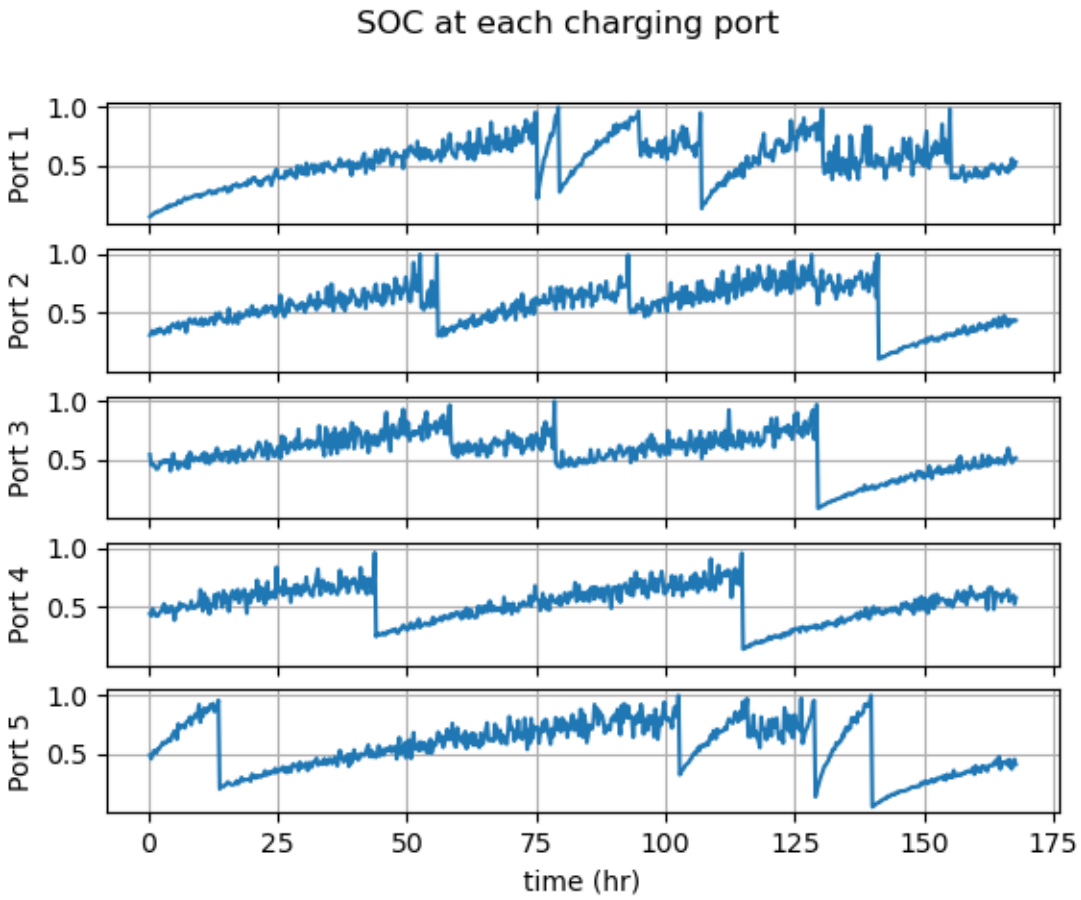
Execution and Results

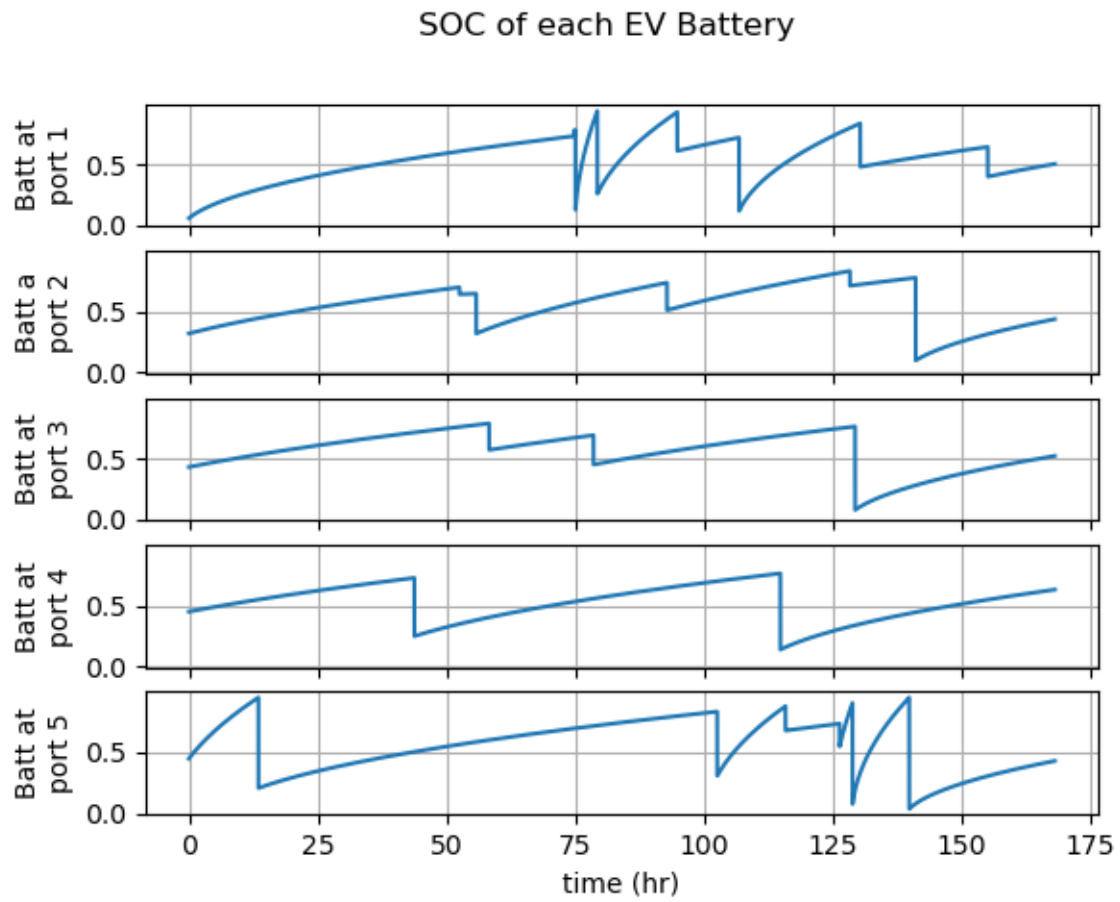
Run the co-simulation:

```
$ helics run --path=./multi_broker_runner.json
```

Since this is only a change to the configuration method of the federation, the results are identical to those in the *Advanced Default example.*


Instantaneous Power Draw from 5 EVs
kW
time (hr)
24000
23000
22000
21000
20000
19000
18000
17000
16000
15000
14000
13000
12000
11000
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0
25
50
75
100
125
150
175


SOC at each charging port
Port 1
Port 2
Port 3
Port 4
Port 5
1.0
0.5
0
25
50
75
100
125
150
175
time (hr)



The dynamic configuration can also been seen by looking at the log file for the Battery federate (`Battery.log`). The pre-configure data flow graph only showed five subscriptions, all made by the Charger federate of the Battery federates current

```
Pre-configure data-flow graph query.
Federate Charger (with id 131072) has the following subscriptions:
    Battery/EV1_current from federate Battery
    Battery/EV2_current from federate Battery
    Battery/EV3_current from federate Battery
    Battery/EV4_current from federate Battery
    Battery/EV5_current from federate Battery
Added subscription Charger/EV1_voltage
```

After analyzing the results of the data-flow graph, the Battery federate subscribing to the appropriate publications from the Charger federate, and re-running the query, the subscription list looks like this:

```
Post-configure data-flow graph query.
Federate Battery (with id 131073) has the following subscriptions:
    Charger/EV1_voltage from federate Charger
    Charger/EV2_voltage from federate Charger
    Charger/EV3_voltage from federate Charger
    Charger/EV4_voltage from federate Charger
    Charger/EV5_voltage from federate Charger
Federate Charger (with id 131072) has the following subscriptions:
```

(continues on next page)

(continued from previous page)

```
    Battery/EV1_current from federate Battery
    Battery/EV2_current from federate Battery
    Battery/EV3_current from federate Battery
    Battery/EV4_current from federate Battery
    Battery/EV5_current from federate Battery
```

Questions and Help

Do you have questions about HELICS or need help?

1. Come to office hours!
2. Post on the gitter!
3. Place your question on the github forum!

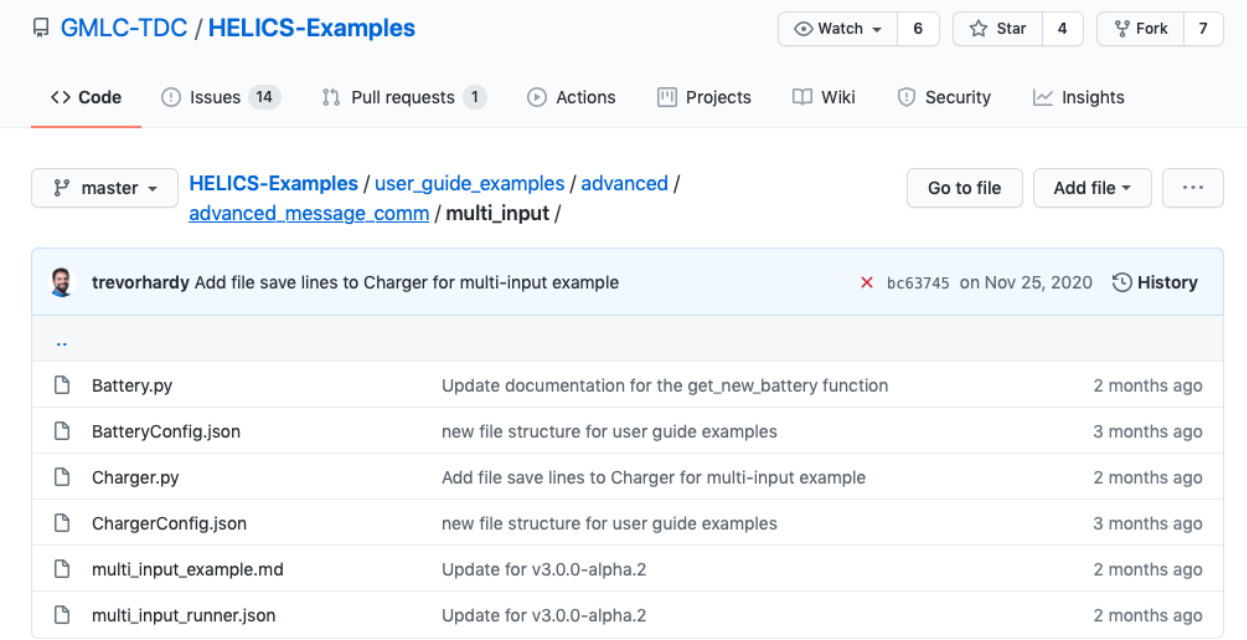
Multi-Input

This demonstrates the use of federation queries and performs dynamic configuration by using the information from the query to configure the Battery federate.

- *Where is the code?*
- *What is this Co-simulation doing?*
  - *Differences Compared to the Fundamental Examples*
    - *HELICS Differences*
  - *HELICS Components*
- *Execution and Results*

Where is the code?

The example on multi-inputs can be found here. If you have issues navigating the examples, visit the HELICS Gitter page or the user forum on GitHub.

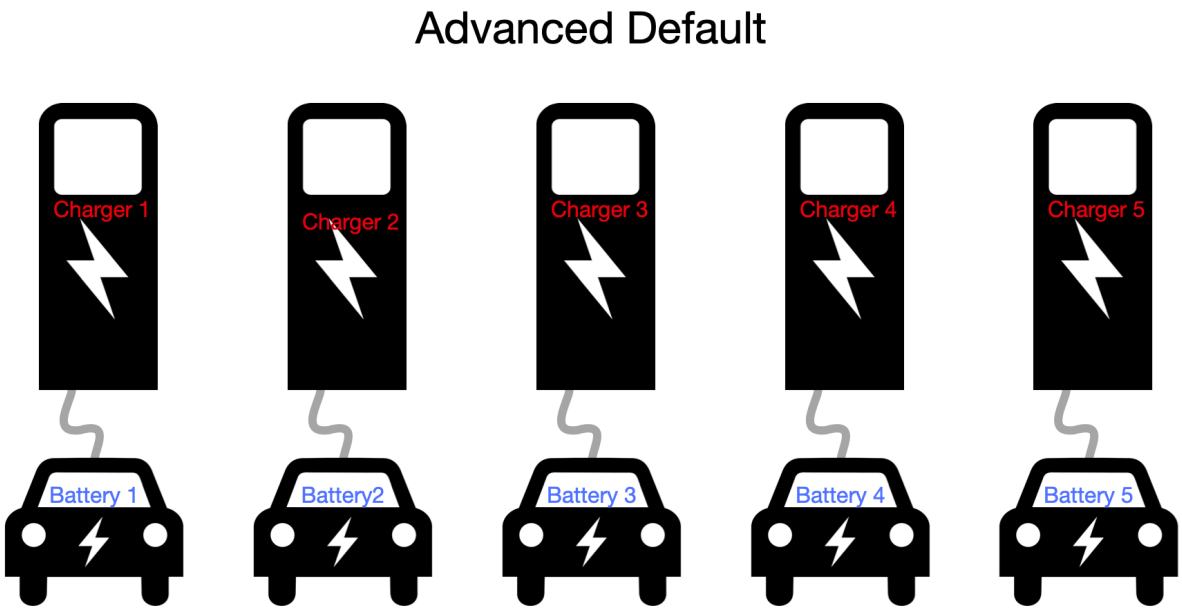


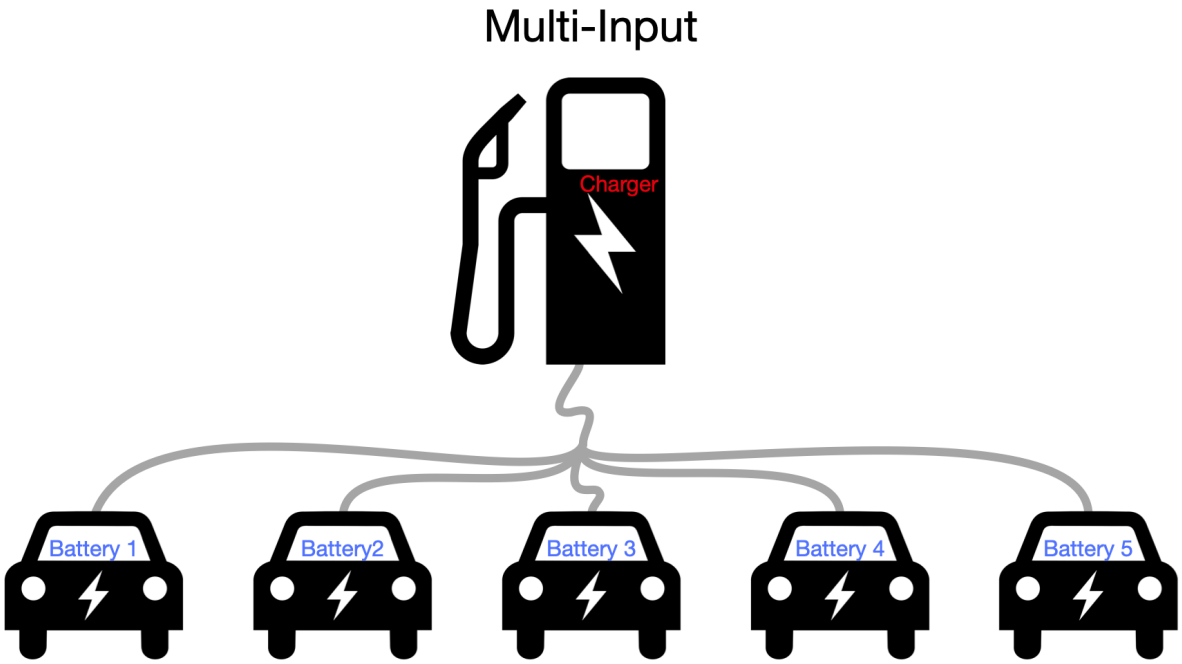
What is this co-simulation doing?

This example shows how to use inputs, allowing multiple publications to arrive at the same input interface (similar to a subscription, as you'll see) and a demonstration on one method of managing data conflicts that can arise.

Differences compared to the Advanced Default example

This example deviates fairly significantly from the *Advanced Default example* in that it only has a Battery and Charger federate. The Charger federate was modeled with one charging terminal that branches out to the five Battery terminals. That is, from the Charger federates perspective, there is only one charging voltage and one charging current even though the federation is still constructed to charge five batteries.





The difference in the model is entirely implied by the HELICS configuration; the physics of the system is modeled through the configuration and this is one valid interpretation.

Additionally, since the protocol (to use the term loosely) in the Advanced Default example for a Battery indicating it was fully charged and the Charger confirming was the removal of the charging voltage, this simple protocol won't work as written when using multi-inputs. Rather than implementing a more sophisticated protocol and to keep the code simple, we decided to just charge one battery at each terminal over the duration of the simulation. Also, due to the removal of the protocol, there is no need of the Controller federate to determine when to stop charging the battery as

the batteries self-terminate their charging.

HELICS differences

With a single charger being used to charge five batteries, each battery still publishes it's charging current as before but only has one subscription, the single charging voltage. This shows up on the Battery federate configuration:

BatteryConfig.json

```
{
  "name": "Battery",
  "loglevel": 1,
  "coreType": "zmq",
  "period": 60,
  "uninterruptible": false,
  "terminate_on_error": true,
  "wait_for_current_time_update": true,
  "publications": [
    {
      "key": "Battery/EV1_current",
      "type": "double",
      "unit": "A",
      "global": true
    },
    {
      "key": "Battery/EV2_current",
      "type": "double",
      "unit": "A",
      "global": true
    }
  ],
  "subscriptions": [
    {
      "key": "Charger/EV_voltage",
      "type": "double",
      "unit": "V",
      "global": true
    }
  ]
}
```

The Charger federate configuration is also altered, using an `input` rather than a `subscription` interface to allow all publications from the Battery federates to be received on one interface. The input has been configured to allow multiple inputs and lists the publications that should be targeted toward it and to handle these multiple inputs by summing them.

```
{
  "name": "Charger",
  "loglevel": 1,
  "coreType": "zmq",
  "period": 60,
  "uninterruptible": false,
  "terminate_on_error": true,
  "publications": [
    {
```

(continues on next page)

(continued from previous page)

```
      "key": "Charger/EV_voltage",
      "type": "double",
      "unit": "V",
      "global": true
    }
  ],
  "inputs": [
    {
      "key": "Battery/charging_current",
      "type": "double",
      "global": true,
      "multi_input_handling_method": "sum",
      "targets": [
        "Battery/EV1_current",
        "Battery/EV2_current",
        "Battery/EV3_current",
        "Battery/EV4_current",
        "Battery/EV5_current"
      ]
    }
  ]
}
```

HELICS Components

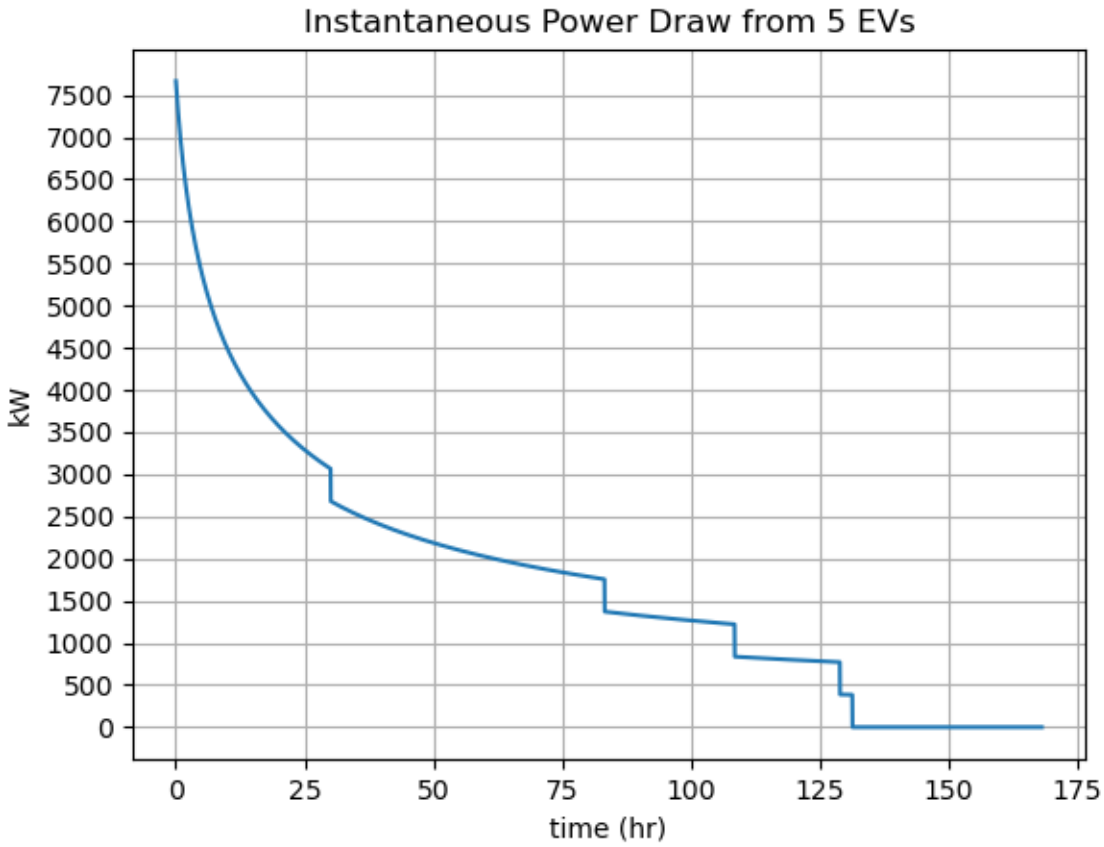
`Battery.py` and `Charger.py` have both been simplified such that only battery per charging terminal is charged over the duration of the simulation. When the battery reaches full SOC, it self-terminates charging.

Execution and Results

Run the co-simulation:

```
$ helics run --path=./multi_input_runner.json
```

The primary result of interest is still the cumulative charging power.



As the batteries are not replaced during charging, the initial charging power will be the peak power. The points in time when a battery reaches full charge, though, can be clearly seen as the discrete changes in cumulative charging power.

Questions and Help

Do you have questions about HELICS or need help?

1. Come to office hours!
2. Post on the gitter!
3. Place your question on the github forum!

Monte Carlo Co-Simulations

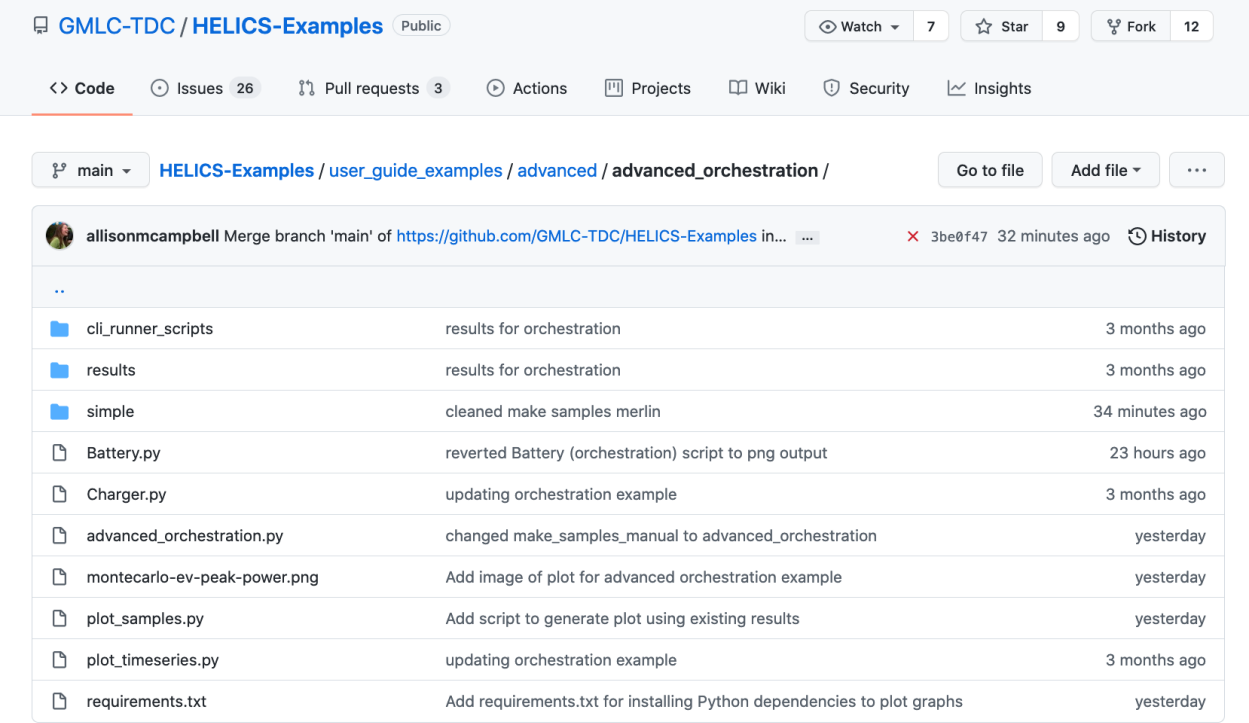
This tutorial will walk through how to set up a HELICS Monte Carlo simulation using two techniques: (1) in series on a single machine, and (2) in parallel on an HPC cluster using Merlin. We assume that you have already completed the ***orchestration tutorial with Merlin*** and have some familiarity with how Merlin works.

- *Where is the code?*
- *What is this Co-simulation doing?*
- *Probabilistic Uncertainty Estimation*
- *Execution and Results*
  - *Manual Orchestration Execution*

– *Merlin Orchestration Execution*

Where is the code?

Code for the Monte Carlo simulation and the full Merlin spec and be found in the HELICS Examples Repo. If you have issues navigating the examples, visit the HELICS Gitter page or the user forum on GitHub.



The necessary files are:

- Python program for Battery federate (Battery.py)
- Python program for Charger federate (Charger.py)
- Python program to generate HELICS runner JSON files and execute (advanced_orchestration.py)

What is this co-simulation doing?

This example walks through how to set up a probabilistic model with Monte Carlo simulations. This Monte Carlo co-simulation is built from a simple two federate example, based on the *Endpoint Federates Example*. In this example, there is a Charger federate which publishes voltage and a Battery federate which publishes current.

All of the HELICS configurations are the same as in the Endpoint example. The internal logic of the federates has been changed for this implementation. The Charger federate assumes the role of *deciding* if the Battery should continue to charge. The Battery sends a message of its current SOC (state of charge), a number between 0 and 1. If the SOC is less than 0.9, the Battery is instructed to continue charging, otherwise, it is instructed to cease charging. The Battery federate has all the logic internal for adding energy and selecting a new "battery" (charging rate) if the SOC is deemed sufficient. Energy is added to the "battery" according to the previous time interval and the charge rate of the battery. In this way, the only stochastic component to the system is the **selected charge rate**. For example, the Endpoint Example allowed the Battery federate to randomly select batteries of different sizes, and the Charger to select charge rates from a list of options. In this implementation, the battery size (capacity in kWh) is constant.

This simplification allows us to isolate a single source of uncertainty: the charge rate.

The co-simulation relies on stochastic sampling of distributions – an initial selection of vehicles for the EV charging garage. We want to ensure that we are not overly reliant on any one iteration of the co-simulation. To manage this, we can run the co-simulation *N* times, or a Monte Carlo co-simulation. The result will be a *posterior distribution* of the instantaneous power draw over a desired period of time.

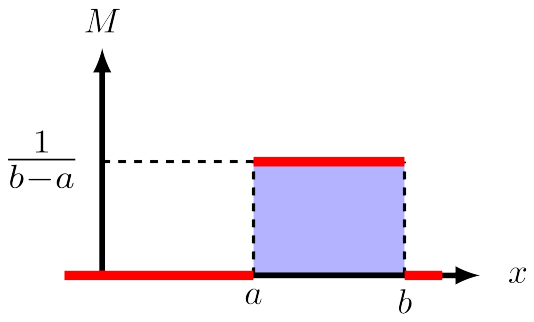
Probabilistic Uncertainty Estimation

A Monte Carlo simulation allows a researcher to sample random numbers repeatedly from a predefined distribution to explore and quantify uncertainty in their analysis. Additional detail about Monte Carlo methods can be found on Wikipedia and MIT Open Courses.

In a Monte Carlo co-simulation, a probability distribution of possible values can be used in the place of **any** static value in **any** of the simulators. For example, a co-simulation may include a simulator (federate) which measures the voltage across a distribution transformer. We can quantify measurement error by replacing the deterministic (static) value of the measurement with a random value from a uniform distribution. Probabilistic distributions are typically described with the following notation:

$M \sim U(a,b)$

Where M is the measured voltage, a is the lower bound for possible values, and b is the upper bound for possible values. This is read as, "M is distributed uniformly with bounds a and b."



The uniform distribution is among the most simple of probability distributions. Additional resources on probability and statistics are plentiful; Statistical Rethinking is highly recommended.

Monte Carlo Co-sim Example: EV Garage Charging

The example co-simulation to demonstrate Monte Carlo distribution sampling is that of an electric vehicle (EV) charging garage. Imagine a parking garage that only serves EVs, has a static number of charging ports, and always has an EV connected to a charging port. An electrical engineer planning to upgrade the distribution transformer prior to building the garage may ask the question: What is the likely power draw that EVs will demand?

Probability Distributions

Likely is synonymous for *probability*. As we are interested in a probability, we cannot rely on a deterministic framework for modeling the power draw from EVs. I.e., we cannot assume that we know a priori the exact demand for Level 1, Level 2, and Level 3 chargers in the garage. A deterministic assumption would be equivalent to stating, e.g., that 30% of customers will need Level 1 charge ports, 50% will need Level 2, and 20% will need Level 3. What if, instead of static proportions, we assign a distribution to the need for each level of charge port. The number of each level port is discrete (there can't be 0.23 charge ports), and we want the number to be positive (no negative charge ports), so we will use the Poisson distribution. The Poisson distribution is a function of the anticipated average of the value and the number of samples k. Then we can write the distribution for the number of chargers in each level, L, as

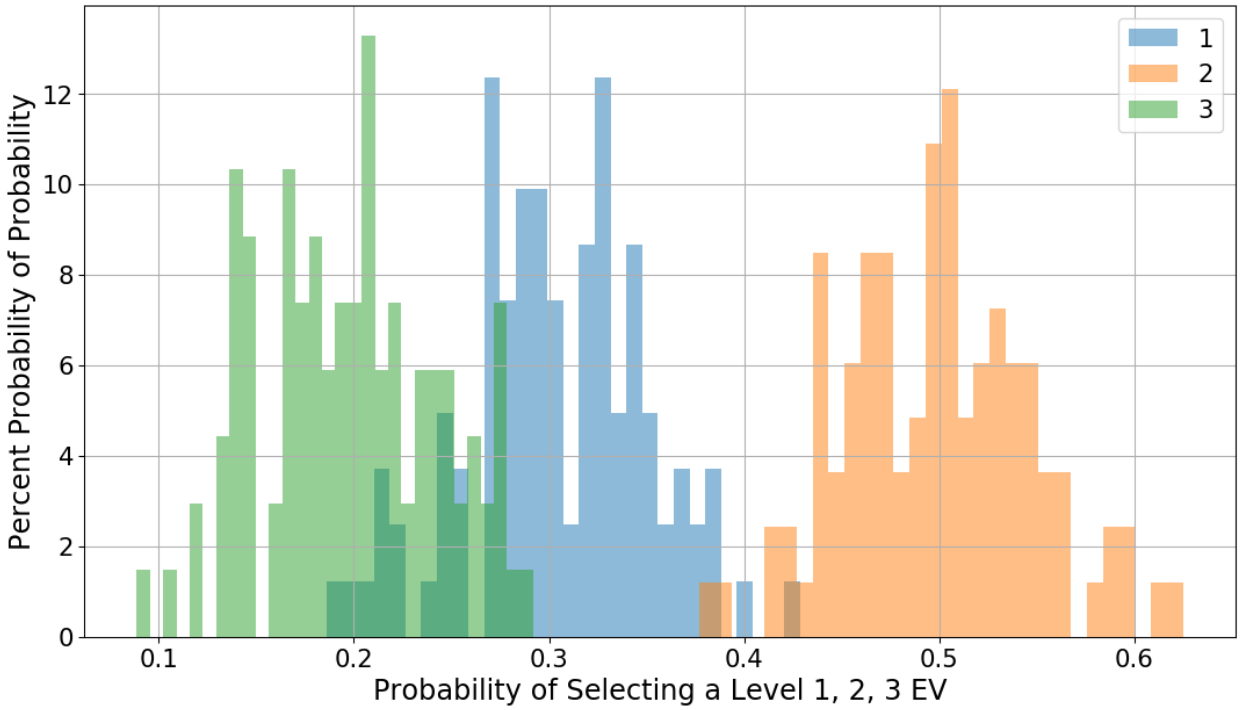
$L \sim P(k,)$

Let's extend our original assumption that the distribution of chargers is static to Poisson distributed, and let's assume that there are 100 total charging ports:

L1 ~ P(100,0.3)

L2 ~ P(100,0.5)

L3 ~ P(100,0.2)



What if we weren't entirely certain that the average values for *L1, L2, L3* are *0.3, 0.5, 0.2*, we can also sample the averages from a normal distribution centered on these values with reasonable standard deviations. We can say that:

~ N(,)

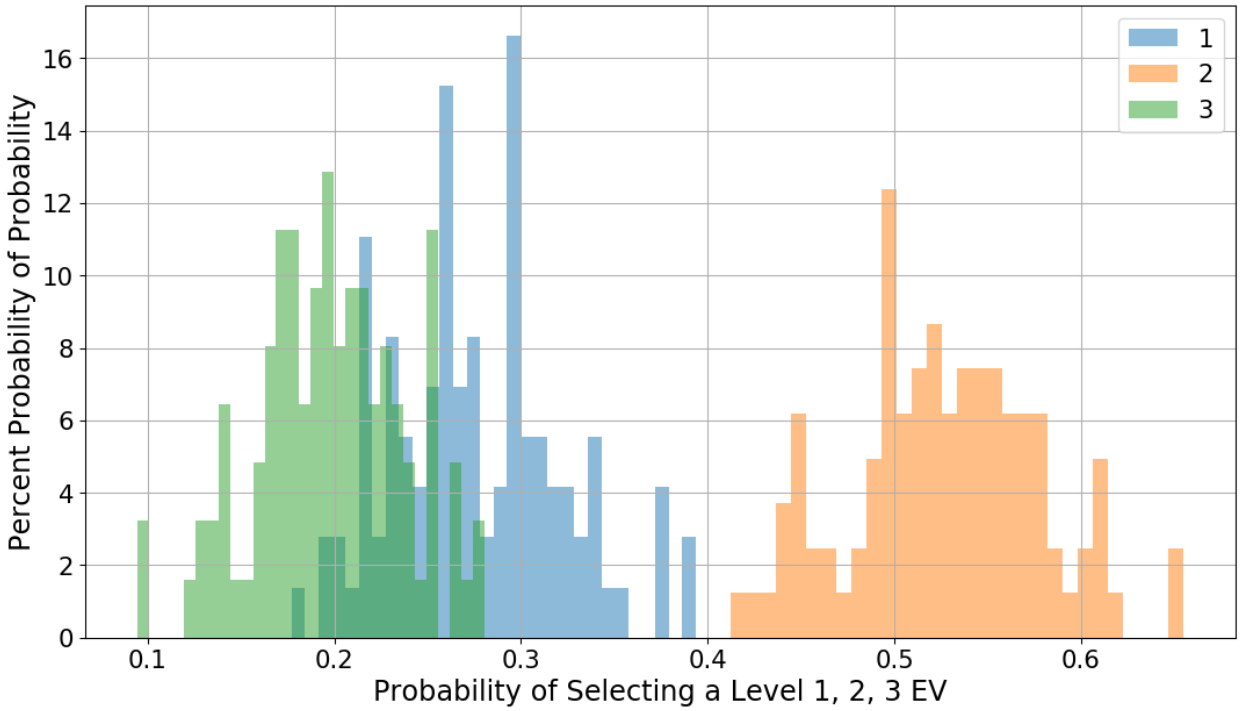
Which means that the input to *L* is distributed normally with average and standard deviation .

Our final distribution for modeling the anticipated need for each level of charging port in our $k = 100$ EV garage can be written as:

L ~ P(k,)

~ N(,)

L1	*L2*	*L3*
0.3	0.5	0.2
~ N(1,3)	~ N(1,2)	~ N(0.05,0.25)



Notice that the individual overplotted distributions in the two histograms above are different – there is more flexibility encoded into the second. The distributions in the second plot describe the following assumptions about the anticipated need for Level 1, 2, and 3 chargers:

1. The number of charging ports is discrete and positive (Poisson).
2. The variance in the number of ports should be normally distributed.
3. The average percentage of Level 1, 2, and 3 chargers is 30%, 50%, and 20%, respectively.
4. The variance around the average is uniformly distributed.
5. The variance is relatively very broad for Level 1, broad for Level 2, and very constrained for Level 3. This means we are very confident in our guess of the average percentage for Level 3 chargers, less confident for Level 2, and much less confident for Level 3.

We have described the individual distributions for each level of charging port. What we don't know, prior to running the Monte Carlo co-simulation, is how these distributions will jointly impact the research question.

Research Question Configuration

We want to address the research question: What is the likely power draw that EVs will demand?

At the beginning of the co-simulation, the distributions defined above will be sampled N times within the EV federate, where $N =$ the number of parking spots/charging ports in the garage. The output of the initial sampling is the number of requested Level 1, 2, and 3 charging ports. The SOC values for the batteries on board are initialized to a uniform random number between 0.05 and 0.5, and these SOC values are sent to the Charger federate. If the SOC of an EV battery is less than 0.9, the EV Controller federate tells the EV battery in the EV federate to continue charging. Otherwise, the EV Charger disconnects the EV battery and instructs it to sample a new EV battery from the distributions (one sample).

After the two federates pass information between each other – EV Battery sends SOC, EV Charger instructs whether to keep charging or resample the distributions – the EV Battery calculates the total power demanded in the last time interval.

Execution and Results

Execution can be done with either a simple script (provided on the repo), or with Merlin.

Manual Orchestration Execution

Manual implementation of the co-simulation is done with the helper script `advanced_orchestration.py`, with command line execution:

```
$ python advanced_orchestration.py
```

This implementation will run a default co-simulation. The default parameters are:

```
samples = 30
output_path = os.getcwd()
numEVs = 10
hours = 24
plot = 0
run = 1
```

This means that we are generating 30 JSON files with unique seeds, we are using the current operating directory as the head for the output path, we are simulating 10 EVs in the co-simulation for one day, we are not running individual plots for each simulation, and we are executing the JSON files with the HELICS runner after they have been created.

If we wanted to run a Monte Carlo co-sim with different parameters, this would be:

```
$ python advanced_orchestration.py 10 . 100 24*7 0 0
```

This execution would create 10 JSON files with unique seeds, set the current directory as the head for the output path, simulate 100 EVs for a week, not generate plots with each simulation, and not execute the JSON files with the HELICS runner (meaning the user will need to manually run each JSON file).

You may decide to adapt `advanced_orchestration.py` to suit your needs within the Merlin environment, in which case you would only need the helper script to create the JSON files. If you elect to use the HELICS runner for the generated runner JSON files using the helper script, subdirectories are created for the HELICS runner JSON files and for the csv results. Results for the default simulation are in the repo and can be used for confirming accurate execution.

```
out_json = output_path + "/cli_runner_scripts"
out_data = output_path + "/results"
```

In the runner scripts directory, there will be 30 JSON files. Each will have a unique `seed` parameter, otherwise they will all look identical:

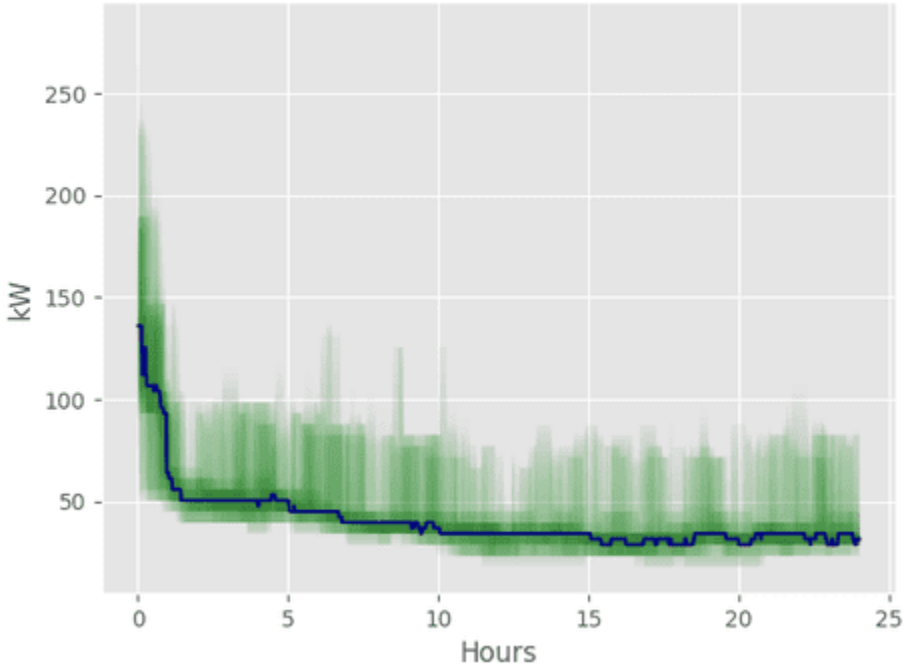
```
{
  "federates": [
    {
      "directory": "[path to local HELICS-Examples/user_guide_examples/advanced/advanced_
↪orchestration dir]",
      "exec": "helics_broker --federates=2 --loglevel=data --coretype=tcpss --port 12345
↪",
      "host": "localhost",
      "loglevel": "data",
      "name": "broker_0"
    },
    {
```

(continues on next page)

(continued from previous page)

```
      "directory": "[path to local HELICS-Examples/user_guide_examples/advanced/advanced_
↪orchestration dir]",
      "exec": "python3 Battery.py --port 12345 --seed 10 --numEVs 10 --hours 24 --plot 0␣
↪--outdir [path to local HELICS-Examples/user_guide_examples/advanced/advanced_
↪orchestration dir]/results",
      "host": "localhost",
      "loglevel": "data",
      "name": "Battery_0"
    },
    {
      "directory": "[path to local HELICS-Examples/user_guide_examples/advanced/advanced_
↪orchestration dir]",
      "exec": "python3 Charger.py --port 12345 --numEVs 10 --hours 24",
      "host": "localhost",
      "loglevel": "data",
      "name": "Charger_0"
    }
  ],
  "name": "Generated by advanced orchestration"
}
```

The final result of the default Monte Carlo co-simulation is shown below.



This is a time series density plot. Each simulation is a green line, and the blue solid line is the median of all simulations. From this plot, we can see that (after the system *initializes*, after a few hours) the maximum demand from EVs in the garage will be roughly 125 kW. We could improve the analysis by conducting an initialization step and by running the simulation for a longer time period. This type of analysis provides the engineer with information about the probability that demand for power from N EVs will be X kW. The most commonly demanded power is less than 50 kW – does the engineer want to size the power conduit to provide median power, or maximum power?

Merlin Orchestration Execution

In this specification we will be using the HELICS runner to execute each co-simulation run since this is a Monte Carlo simulation. This means that the HELICS runner will be called multiple times with different JSON runner files.

Co-simulation Reproduction

Management of multiple iterations of the co-simulation can be done by setting the seed as a function of the number of brokers, where there will be one broker for each iteration. In Python 3.X:

```
import argparse

parser = argparse.ArgumentParser(description="EV simulator")
parser.add_argument(
    "--seed",
    type=int,
    default=0,
    help="The seed that will be used for our random distribution",
)
parser.add_argument("--port", type=int, default=-1, help="port of the HELICS broker")


args = parser.parse_args()
np.random.seed(args.seed)
```

HELICS runner in Merlin

Since we are using the HELICS runner to manage and execute all the federates, we need to create these runner JSON files. There is a provided python script called `make_samples_merlin.py` (see the `simple` subfolder in the code for the example) that will generate the runner file and a csv file that will be used in the study step. The HELICS runner command will start each of these federates. In the Merlin spec, Merlin will be instructed to start each run using the JSON HELICS runner files.

Merlin Specification

Environment

In the Merlin spec we will instruct Merlin to execute N number of the Monte Carlo co-simulations. The number of samples is the number specified as the `N_SAMPLES` env variable in the env section of the Merlin spec.

```
env:
  variables:
    OUTPUT_PATH: ./UQ_EV_Study
    N_SAMPLES: 10
```

We set the output directory to `UQ_EV_Study`, this is where all the output files will be stored. Every co-simulation run executed by Merlin will have its own subdirectory in `./UQ_EV_Study`.

Merlin Step

Remember this step is for Merlin to setup all the files and data it needs to execute its jobs. In the Monte Carlo co-simulation there is a python script we created that will generate the HELICS runner files that Merlin will use when it executes a specific run. The `make_samples_merlin.py` script (located under the `simple` subfolder of the advanced orchestration example code) will also output a csv file that Merlin will use. The csv file contains all the names of the

runner files. Merlin will go down this list of file names and launch each co-simulation using the HELICS runner JSON file.

```
merlin:
  samples:
    generate:
      cmd: |
        python3 $(SPECROOT)/simple/make_samples_merlin.py $(N_SAMPLES) $(MERLIN_INFO)
        cp $(SPECROOT)/Battery.py $(MERLIN_INFO)
        cp $(SPECROOT)/Charger.py $(MERLIN_INFO)
    file: $(MERLIN_INFO)/samples.csv
    column_labels: [FED]
```

The samples the get generated should look something like below. This is the runner file that the HELICS runner will use to start the co-simulation.

```
{
  "federates": [
    {
      "directory": ".",
      "exec": "helics_broker --federates=2 --loglevel=5 --type=tcpss --port 12345",
      "host": "broker",
      "name": "broker_0",
      "loglevel": 3
    },
    {
      "directory": ".",
      "exec": "python3 Battery.py --port 12345 --seed 1",
      "host": "broker",
      "name": "Battery",
      "loglevel": 3
    },
    {
      "directory": ".",
      "exec": "python3 Charger.py --port 12345",
      "host": "broker",
      "name": "Charger",
      "loglevel": 3
    }
  ],
  "name": "Generated by make samples"
}
```

Once the samples have been created, we copy the 2 federates to the `MERLIN_INFO` directory.

Study Step

We have made it to the study step. This step will execute all 10 Monte Carlo co-simulations. There are 2 steps in the study step. The first step is the `start_parallel_sims` step. This step will use the HELICS runner to execute each co-simulation. The second step, `cleanup` depends on the first step. Once `start_parallel_sims` is complete the `cleanup` step will remove any temporary data that is no longer needed.

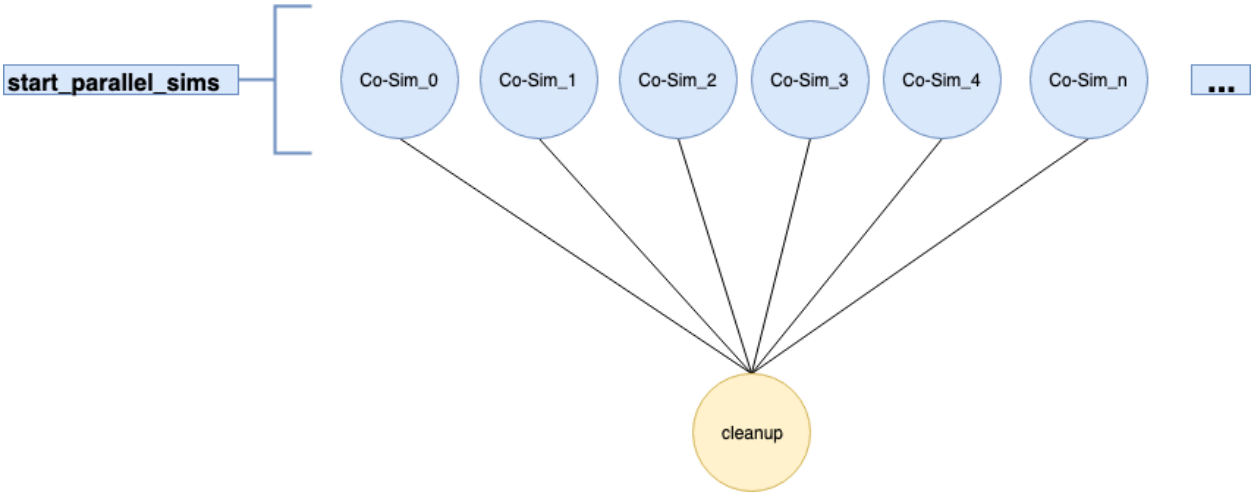
```
study:
  - name: start_parallel_sims
```

(continues on next page)

(continued from previous page)

```
    description: Run a bunch of UQ sims in parallel
    run:
      cmd: |
        spack load helics
        helics run --path=$(MERLIN_INFO)/$(FED)
        echo "DONE"
  - name: cleanup
    description: Clean up
    run:
      cmd: rm $(MERLIN_INFO)/samples.csv
      depends: [start_parallel_sims_*]
```

Below is what the DAG of the Merlin study will look like. Each of the Co-Sim_n bubbles represents the Monte Carlo simulation. Each co-sim runs in parallel with each other since there is no dependency on the output that each co-sim runs.



Questions and Help

Do you have questions about HELICS or need help?

1. Come to office hours!
2. Post on the gitter!
3. Place your question on the github forum!

Iteration

This demonstrates the use of iteration both in the initialization phase, as well as execution, in order to reach convergence between federates.

- *Where is the code?*
- *What is this Co-simulation doing?*
  - *Differences Compared to the Fundamental Examples*
  - *Iteration Components*
- *Execution and Results*
  - *Initialization Results*

    - *Time Loop Results*
- *Impact of Iteration*

Where is the code?

The example on iteration can be found here. If you have issues navigating the examples, visit the HELICS Gitter page or the user forum on GitHub.

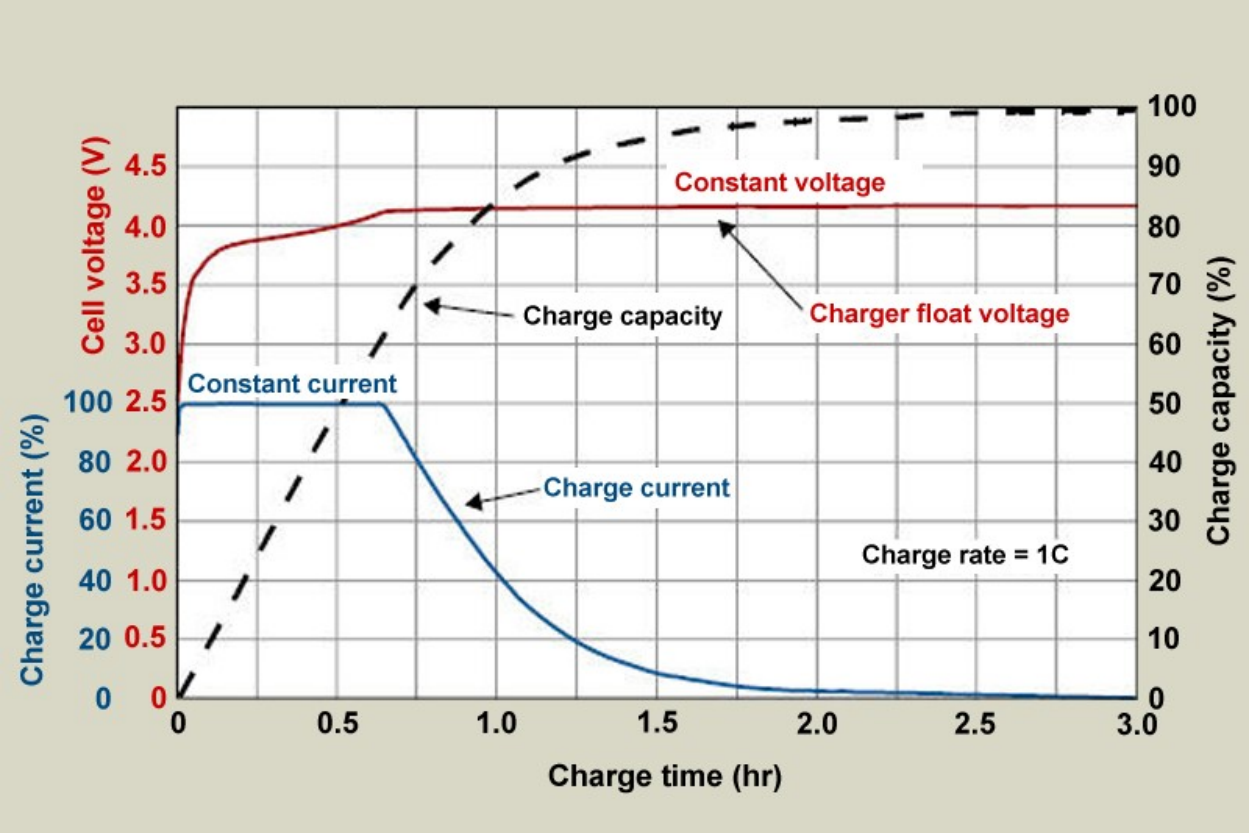
What is this co-simulation doing?

This example shows how to set up the iteration calls that support state convergence across federates. This is discussed in more detail in the *User Guide*.

Differences Compared to the Advanced Default Example

This example changes the model of the default example so that the charger voltage is a function of the battery current: $V_{ch}(I_b)$. Conversely, battery current is a function of charger voltage: $I_b(V_{ch})$. As a result, each time step requires some iteration to find the converged fixed-point, where both models are in a consistent state.

In very loose terms, the charging strategy uses constant current below a specific state of charge followed by constant voltage once rated voltage is achieved:



source: https://batteryuniversity.com/article/bu-409-charging-lithium-ion

Charger Behavior

In the default example the charger has a rated voltage and power. This has been changed to rated voltage and current:

- *Level 1*: 120V, 15A
- *Level 2*: 240V, 30A
- *Level 3*: 630V, 104A

```
def get_charger_ratings(EV_list):
    """
    This function uses the pre-defined charging powers and maps them to
    standard (more or less) charging voltages. This allows the charger
    to apply an appropriately modeled voltage to the EV based on the
    charging power level

    :param EV_list: Value of "1", "2", or "3" to indicate charging level
    :return: charging_voltage: List of charging voltages corresponding
            to the charging power.
    """
    out = []
    # Ignoring the difference between AC and DC voltages for this application
    charge_voltages = [120, 240, 630]
    charge_currents = [15, 30, 104]
    for EV in EV_list:
        if EV not in [1, 2, 3]:
            out.append({"Vr": 0, "Ir": 0})
        else:
            out.append({"Vr": charge_voltages[EV - 1], "Ir": charge_currents[EV - 1]})
        logger.debug(
            "EV {} ratings: Vr = {} Ir = {}".format(EV, out[-1]["Vr"], out[-1]["Ir"])
        )
    return out
```

The charging function, $V_{ch}(I_b)$ is implemented `voltage_update` in `Charger.py`. It considers 4 different cases (the very first `if` statement is an initialization option):

- *Case 1*: Current is within tolerance ϵ of rated current $\rightarrow$ voltage remains the same.
- *Case 2*: Current is below rated current *and* voltage is below rated voltage $\rightarrow$ voltage is increased via bisection search.
- *Case 3*: Current is below rated *and* voltage is rated voltage $\rightarrow$ retain rated voltage.
- *Case 4*: Current is greater than rated current $\rightarrow$ reduce voltage via bisection search.

```
def voltage_update(
    charger_rating, charging_current, charging_voltage=None, epsilon=1e-2, quiet=False
):
    if charging_voltage is None:
        return {"V": charger_rating["Vr"], "Vmin": 0, "Vmax": charger_rating["Vr"]}
    if abs(charging_current - charger_rating["Ir"]) < epsilon:
        # Constant current charging: do nothing
        pass
    elif (charging_current < charger_rating["Ir"]) and (
        charging_voltage["V"] < charger_rating["Vr"]
```

(continues on next page)

(continued from previous page)

```
    ):
        # increase voltage
        charging_voltage = {
            "V": (charging_voltage["Vmax"] + charging_voltage["V"]) / 2,
            "Vmin": charging_voltage["V"],
            "Vmax": charging_voltage["Vmax"],
        }
    elif (charging_current < charger_rating["Ir"]) and (
        abs(charging_voltage["V"] - charger_rating["Vr"]) < epsilon
    ):
        # constant voltage charging: do nothing
        pass
    elif charging_current > charger_rating["Ir"]:
        # decrease voltage
        charging_voltage = {
            "V": (charging_voltage["V"] + charging_voltage["Vmin"]) / 2,
            "Vmin": charging_voltage["Vmin"],
            "Vmax": charging_voltage["V"],
        }
    else:
        raise ValueError(
            "voltage_update: inputs do not match any of the expected cases"
        )

    return charging_voltage
```

Battery Model Development

In order to achieve the desired charging profile, a fictitious battery resistance $R(soc)$ is used. As this R increases the voltage must increase to maintain the rated current I_{rated} until V_{rated} is reached and held, at which point the current begins to taper.

We utilize the following design criteria:

1. The actual internal resistance of a battery is about on the order of 0.5 Ω[1], which will be used as the $R(soc = 0)$
2. We would like constant current charging until around $soc = 0.6$.
3. Charging is considered full when current drops below 3% rated.

From Criteria 2 we have $R(0.6) = V_{\text{rated}}/I_{\text{rated}}$. For the three charger types we have defined that is:

- *Level 1*: $R(0.6) = 120\text{V}/15\text{A} = 8\Omega$
- *Level 2*: $R(0.6) = 240\text{V}/30\text{A} = 8\Omega$
- *Level 3*: $R(0.6) = 630\text{V}/104\text{A} \approx 6\Omega$

Splitting the difference we fix $R(0.6) \stackrel{!}{=} 7\Omega$.

From Criteria 3 we have $R(1) = V_{\text{rated}}/(0.03 \cdot I_{\text{rated}})$. For the three charger types defined that is:

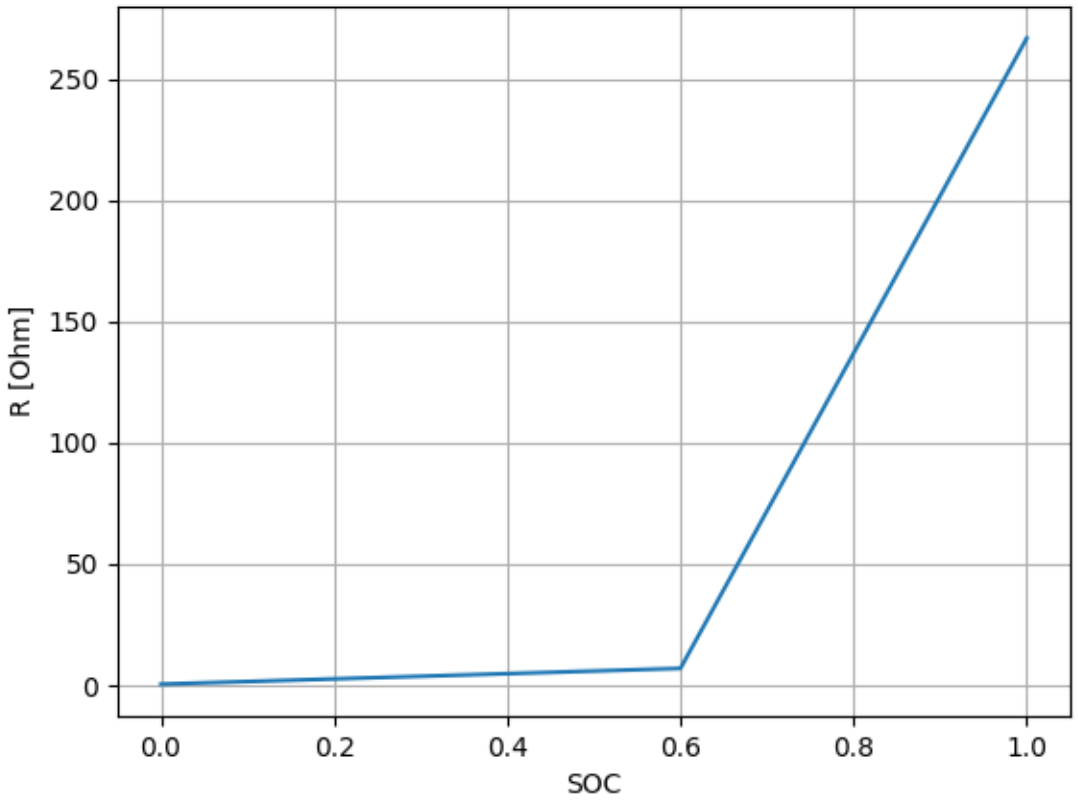
- *Level 1*: $R(0.6) = 120\text{V}/(0.03 \cdot 15\text{A}) \approx 267\Omega$
- *Level 2*: $R(0.6) = 240\text{V}/(0.03 \cdot 30\text{A}) \approx 267\Omega$
- *Level 3*: $R(0.6) = 630\text{V}/(0.03 \cdot 104\text{A}) \approx 202\Omega$

[1] See this article

Taking the maximum (so we make sure the current decays *at least* to 3% rated by full charge) we fix $R(1) \stackrel{!}{=} 267\Omega$.

The function $R(soc)$ is now created as two linear segments: one from 0 to 0.6 and the other from 0.6 to 1:

```
def effective_R(soc):
    if soc >= 0.6:
        return 650 * soc - 383
    else:
        return 10.83 * soc + 0.5
```



The current is then simply:

```
def current_update(charging_voltage, soc):
    # Calculate charging current and update SOC
    R = effective_R(soc)
    # If battery is full assume its stops charging on its own
    #  and the charging current goes to zero.
    if soc >= 1:
        return 0
    else:
        return max(0, charging_voltage / R)
```

Iteration components

The majority of the iteration related code is found in `iterutils.py` in order to reuse it in both `Battery.py` and `Charger.py`.

Before diving into the details, HELICS has iteration *requests* and *results*. The *requests* are passed as inputs and the *results* are returned (see the *User Guide section on iteration* for further details).

The relevant *requests* are:

- `NO_ITERATION`: this *forces* movement to the next time step and should therefore be avoided by *all* Federates if iteration is desired.
- `FORCE_ITERATION`: this *forces* iteration. If a federate *always* requests this the simulation will be stuck in a never ending loop. It is intended to be used only in *rare* cases.
- `ITERATE_IF_NEEDED`: in this case iteration only takes place if new values are *published*. This is the *usual* flag that should be used when iteration is desired.

The last point is ***critical***; HELICS uses the relevant outputs of other federates to determine if a given federate needs to iterate.

The relevant *results* are:

- `NEXT_STEP`: simulation is moving on.
- `ITERATING`: simulation is still iterating.

Each federate should therefore look for `NEXT_STEP` in order to advance in time and otherwise continue to iterate at the current time.

Note: An essentially equivalent method to checking the flag for `ITERATING` or `NEXT_STEP` is to check whether the currently granted time is the same the previous one. `ITERATING` ⇒ `grantedtime == grantedtime_previous`. `NEXT_STEP` ⇒ `grantedtime > grantedtime_previous`.

Initialization

The first step is to initialize the federates. In this step `helicsFederateEnterInitializingMode` is called and the federates publish their initial values.

The function is `set_pub` (with option `init=True` and logging commands taken out for clarity):

```
def set_pub(self, fed, pubid, pubvals, nametyp=None, init=False):
    if init:
        self.logger.info("=== Entering HELICS Initialization mode")
        h.helicsFederateEnterInitializingMode(fed)
    pub_count = h.helicsFederateGetPublicationCount(fed)
    for j in range(0, pub_count):
        h.helicsPublicationPublishDouble(pubid[j], pubvals[j])
```

The function call looks like:

```
# Battery.py
feditr.set_pub(fed, pubid, charging_current, "Battery", init=True)

# Charger.py
feditr.set_pub(fed, pubid, [x["V"] for x in charging_voltage], "EV", init=True)
```

Next, the federates *iterate* using `helicsFederateEnterExecutingModeIterative`. The basic structure is:

- Call `helicsFederateEnterExecutingModeIterative`.

  - If the result is `NEXT_STEP` then we're done.

```
itr = 0
itr_flag = h.helics_iteration_request_iterate_if_needed
while True:
    itr_status = h.helicsFederateEnterExecutingModeIterative(fed, itr_flag)
    if itr_status == h.helics_iteration_result_next_step:
        break
```

- Get subscriptions. Done via the `get_sub` function:

```
def get_sub(self, fed, subid, itr, valarray, valinit, nametyp, proptyp):
    sub_count = h.helicsFederateGetInputCount(fed)
    for j in range(0, sub_count):
        x = h.helicsInputGetDouble((subid[j]))
        if itr == 0:
            valarray[j] = [valinit] * 2
        else:
            valarray[j].insert(0, valarray[j].pop())
        valarray[j][0] = x
```

  Here `valarray` stores the current *and* previous value to check for convergence. The function call looks like:

```
# Battery.py
feditr.get_sub(fed, subid, itr, charging_voltage, vinit, "Battery", "voltage")

# Charger.py
feditr.get_sub(fed, subid, itr, charging_current, iinit, "EV", "current")
```

- check error
  - If the error is sufficiently small loop back and ***do not publish***, otherwise proceed to update and publishing, which will trigger another iteration. The function to check the error is `check_error`:

```
def check_error(self, dState):
    return sum([abs(vals[0] - vals[1]) for vals in dState.values()])
```

  The function call looks like[2]:

```
error = feditr.check_error(charging_voltage)
if (error < epsilon) and (itr > 0):
    # no further iteration necessary
    continue
else:
    pass
```

- perform state update based on subscription values.

```
# Battery.py
charging_current[j] = current_update(charging_voltage[j][0], current_soc[j])

# Charger.py
charging_voltage[j] = voltage_update(
```

(continues on next page)

[2] For `Battery.py` in `Charger.py` the input is `charging_current` instead of `charging_voltage`.

(continued from previous page)

```
    charger_ratings[j], charging_current[j][0], charging_voltage[j]
)
```

- publish new outputs and increment iteration

```
# Battery.py
feditr.set_pub(fed, pubid, charging_current)
itr += 1

# Charger.py
feditr.set_pub(fed, pubid, [x["V"] for x in charging_voltage])
itr += 1
```

Time Loop

The time loop looks almost identical to the initialization loop, except that instead of calling `helicsFederateEnterExecutingModeIterative`, the call is to `helicsFederateRequestTimeIterative`:

```
itr = 0
itr_flag = h.helics_iteration_request_iterate_if_needed
while True:
    grantedtime, itr_state = h.helicsFederateRequestTimeIterative(
        fed, requested_time, itr_flag
    )
    if itr_state == h.helics_iteration_result_next_step:
        break  # Iteration complete!
    else:
        pass  # Iterating
```

Similar to the initialization, it is ***critical*** to publish ***before*** the first time request, otherwise, HELICS will see no new data, and return `NEXT_STEP`. The general simulation code is therefore:

```
while grantedtime < total_interval:
    # Publication needed so we can actually iterate
    feditr.set_pub(fed, pubid, publication_values)

    # Time request for the next physical interval to be simulated
    requested_time = grantedtime + update_interval

    [...]
```

Note that this essentially means, that we publish our final values from time t as the very first thing in time $t + \Delta t$.

Execution Results

Initialization Results

Two figures are produced following the initialization iteration, which show how the currents and voltages converge. Note that all batteries are in the constant current phase of charging, as such they all converge to their rated current (30A for EV1-EV4 and 104A for EV5). The voltages meanwhile are *not* nominal (240V or 630V) but rather determined based on the effective impedance, which is dependent on the initialized soc.

Time Loop Results

Four figures are produced once the co-simulation runs its course, two from Battery.py and two from Charger.py.

Battery Results

The Battery results show the charging current in each battery and the development of the SoC over time. As desired, the charging current exhibits the constant current followed by a decay characteristic, and the SoC rise to 1.

Charger Results

The Charger results show how the voltage rises to its rated value and then remains there. The total power plot is also quite interesting. Since initially the current remains fixed while the voltage rises, the total power draw increases. Once the charging mode switches to constant voltage and the current decreases, the power draw follows suit.

Impact of Iteration

It is useful to think about *why* iteration would be necessary in a simulation in the first place. In a world governed by differential equations, iteration essentially smooths out timescales below the range of interest. It allows us to assume that quicker dynamics have already settled to their steady state. Power systems engineers might be quite familiar with this concept. The power flow, which assumes steady state of voltage and current dependencies, requires iteration because of the use of constant power loads. Dynamic simulations (in their simplest form), however, do not iterate, because all loads are converted to impedances. More advanced dynamic simulations do iterate because they are attempting to account for the impact of electromagnetic phenomena (e.g. power electronics control loops) that have already settled.

In this example, the charger needs to find the appropriate voltage to achieve rated current. In reality, there are a multitude of possible control algorithms that might be implemented. The use of iteration here is an acknowledgement that this fixed point *will* be found, but at a timescale that is effectively "instantaneous" w.r.t. the interval of one hour used in the simulation. That is $\Delta t_{\text{control}} \ll \Delta t$.

To illustrate this, a second copy of the iteration is available in the repository (`Battery_noitermain.py`, `Charger_noitermain.py`, and `advanced_iteration_runner_noitermain.json`) with the iteration turn off (except for initialization) using the flag `iterative_mode = False`. A comparison of the two runs is presented below:

With Iteration	Without Iteration

Without iteration the "voltage hunting" that happens during the constant-current charging phase takes place over hours instead of *instantaneously* as is the case *with* iteration. During the constant voltage phase the two runs are identical, since there is no longer an interdependence between the federates.

Finally, it is worth noting that the oscillation observed here is a function of the convergence algorithm implemented (see function `voltage_update`). A more sophisticated algorithm may lead to less oscillation. For example, if the charger had a model of the battery SOC and its dependence on the equivalent resistance, it could estimate the current and thus obtain the right voltage immediately. The point is, that the specific charger control architecture is not of interest in this simulation and iteration allows to abstract out the implementation without completely neglecting certain model interdependencies and focus on behavior in the time frame of interest.

Questions and Help

Do you have questions about HELICS or need help?

1. Come to office hours!
2. Post on the gitter!
3. Place your question on the github forum!

FMUs with HELICS

This demonstrates the use of HELICS-FMI to integrate an FMU into a HELICS-based co-simulation. Functional Mock-Up Interface (FMI) is an existing method of integrating models that is similar to HELICS in some ways. FMI provides a way to specify a model and, with the help of a supporting tool, compile it into a binary and provide a companion specification of the interface to that model. In some cases, a solver for the model is also distributed along with the model, effectively making it a highly specialized simulation tool. These elements can be bundled up into a distributable unit called a "functional mock-up unit" (FMU) that can be distributed without revealing the inner workings of a model. HELICS provides a way to allow one or more FMUs to participate in a HELICS co-simulation using the HELICS-FMI application.

- *Where is the code?*
- *What is this Co-simulation doing?*
- *HELICS components*
- *Building the FMU*
- *Configuring the co-simulation*
- *Execution and Results*

Where is the code?

The code for the Advanced examples can be found in the HELICS-Examples repository on GitHub. The example on using an FMU in a HELICS-based co-simulation can be found here. If you have issues navigating the examples, visit the HELICS Gitter page or the user forum on GitHub.

Note that this code contains a component (the FMU) that has been compiled for Windows and thus will only function on Windows. The FMU source code has been included and can be compiled into an FMU for other platforms with the appropriate tools.

What is this co-simulation doing?

This example shows you how to take an existing FMU and incorporate it into a HELICS-based co-simulation. This effectively allows FMUs to act as HELICS federates.

The Functional Mock-up Interface (FMI) is a modeling language popularly implemented by the commercial tool Modelon (*nee* Modelica) but also has an open-source implementation OpenModelica. The modeling language allows for a black-box description of a model's interfaces to support co-simulation of various modeled entities through the creation

of an FMU. The FMU, when used for co-simulation, consists of two main components: 1. an XML file describing the data exchange interfaces and 2. a binary version of the model with callable FMI-defined functions that simulate the model.

FMI effectively defines another means of performing a co-simulation using FMUs and the HELICS team has created a means by which these FMUs can join a HELICS-based co-simulation. To integrate the FMU into the example, HELICS-FMI acts as a bridge that is able to execute the "SimpleBattery.fmu" FMU. HELICS-FMI takes care of executing the FMU using the inputs from the rest of the co-simulation and providing the outputs on its behalf.

HELICS components

To enable an FMU to act as a HELICS federate, the "helics_fmi" application has been developed. This application has the ability to read an FMU, set up the defined data-exchange interfaces as HELICS value interfaces, and call the necessary simulation functions for the FMU to push it forward in simulated time. This puts HELICS in the role of "master algorithm" (to use the FMI parlance).

As a more specialty item, helics_fmi is not included as a part of the standard HELICS distribution and must be built from source. The source code can be found in the "helics_fmi" repository and uses a CMake build process like the main HELICS library. You can follow the instructions here on building the main HELICS library but work from the helics_fmi repo.

Building the FMU

Distributed with the example is the source code for the FMU battery model. This source code needs to be compiled into an FMU for use in this example. Provided with this example is the FMU for Windows; if running this example on another platform the FMU will have to be compiled on that platform to produce a valid binary.

Configuring the Co-Simulation

With helics_fmi and FMU built, all that remains is providing command line options for helics_fmi. Looking at the runner JSON for this example, the following options are used:

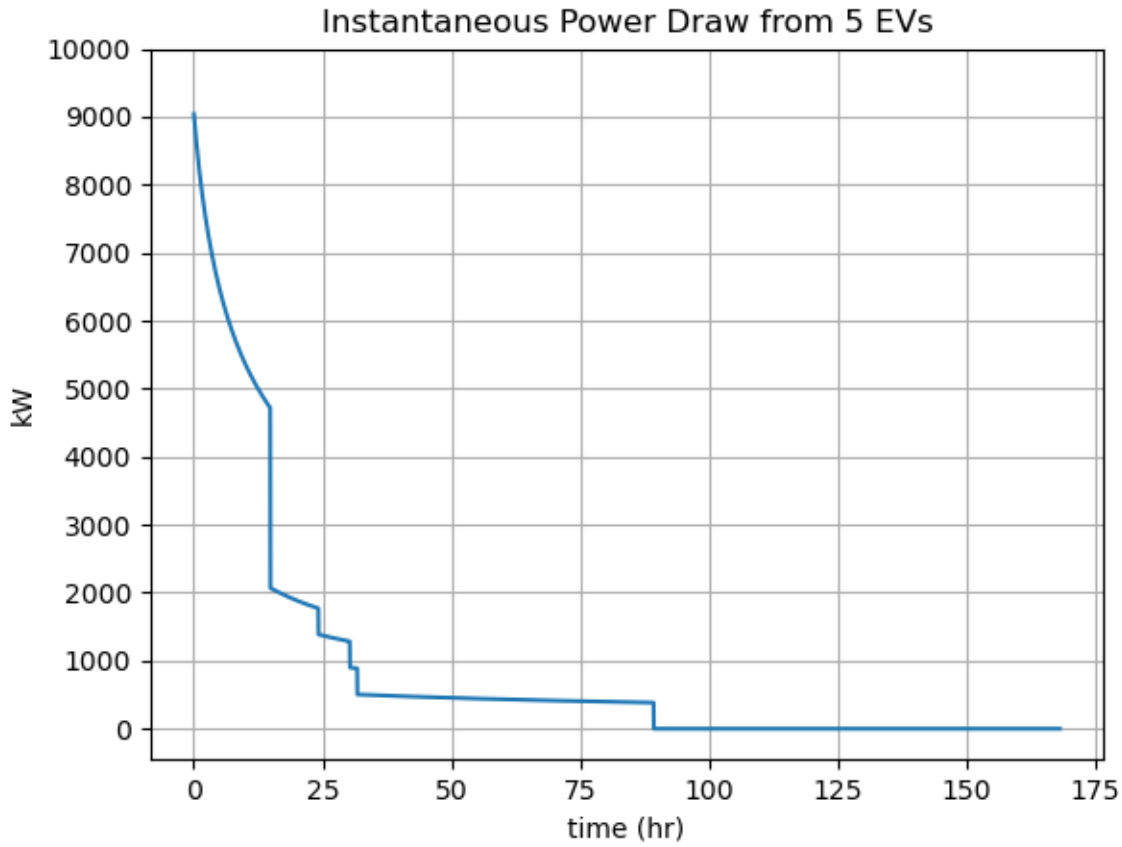
- `stoptime` - The simulated time in seconds for ending the FMU simulation
- `step` - Simulation step size, equivalent to `period` in HELICS configuration
- `name` - Federate name
- `flags` - Special runtime flags to address particular FMU needs
- `set` - Allow for interaction with the interface values defined by the FMU. In this case it is used to define initialization values.

Execution and Results

To execute this example, use the provided runner file:

```
helics run --path=runner.json
```

The FMU has been designed to replicate the model used in this example suite and the results should be identical to those provided in the fundamental default example:



Questions and Help

Do you have questions about HELICS or need help?

1. Come to office hours!
2. Post on the gitter!
3. Place your question on the github forum!

Asynchronous (non-blocking) Time Request

Generally, when federates make a time request they do so by calling `helicsFederateRequestTime()`. This is a blocking call; the HELICS library will wait until it has reached the time it determines should be granted to the federate and then will do so, returning the granted time. In many cases, this blocking call is acceptable as a time request generally means a federate has nothing to do until it receives new inputs from the rest of the federation as it has completed updating its internal model based on the last received inputs from the federation.

There are cases, though, where a federate may want to make a time request and then continue working on another computational task instead of waiting in an idle state for the time request to be granted. HELICS provides an alternative time request API to accomplish this: `helicsFederateRequestTimeAsync()`.

Using the Asynchronous Time Request

Making the asynchronous time request is simple: to make the non-blocking time request use the `helicsFederateRequestTimeAsync()` API. This API does not return a value and does not block the federate operation. The code following this API will begin immediately executing while the rest of the federation advances through simulated time. When this federate is done with its extra work, it uses the `helicsFederateRequestTimeComplete()` API call. This is a blocking call (just like `hliecsFederateRequestTime()` was), though it is possible the state of the federation is such that HELICS is ready to immediately grant a time.

There is also a similar set of APIs when working with the iterative time requests: `helicsFederateRequestTimeIterativeAsync()` and `helicsFederateRequestTimeIterativeComplete()`.

Example Explanation

The advanced default example has been very slightly modified to demonstrate the use of `helicsFederateRequestTimeAsync()` and `helicsFederateRequestTimeComplete()`. In this case, Battery.py has been edited to always make the async time request and then immediately call a frivolous function that, once a simulated day, delays the time request by one wall-clock second and sends a message to the logs. (Obviously, in a real-world application the work done in this function would not be frivolous.) Running this example and looking at the logs shows this message, indicating the function is being run after the async time request but before completing the time request.

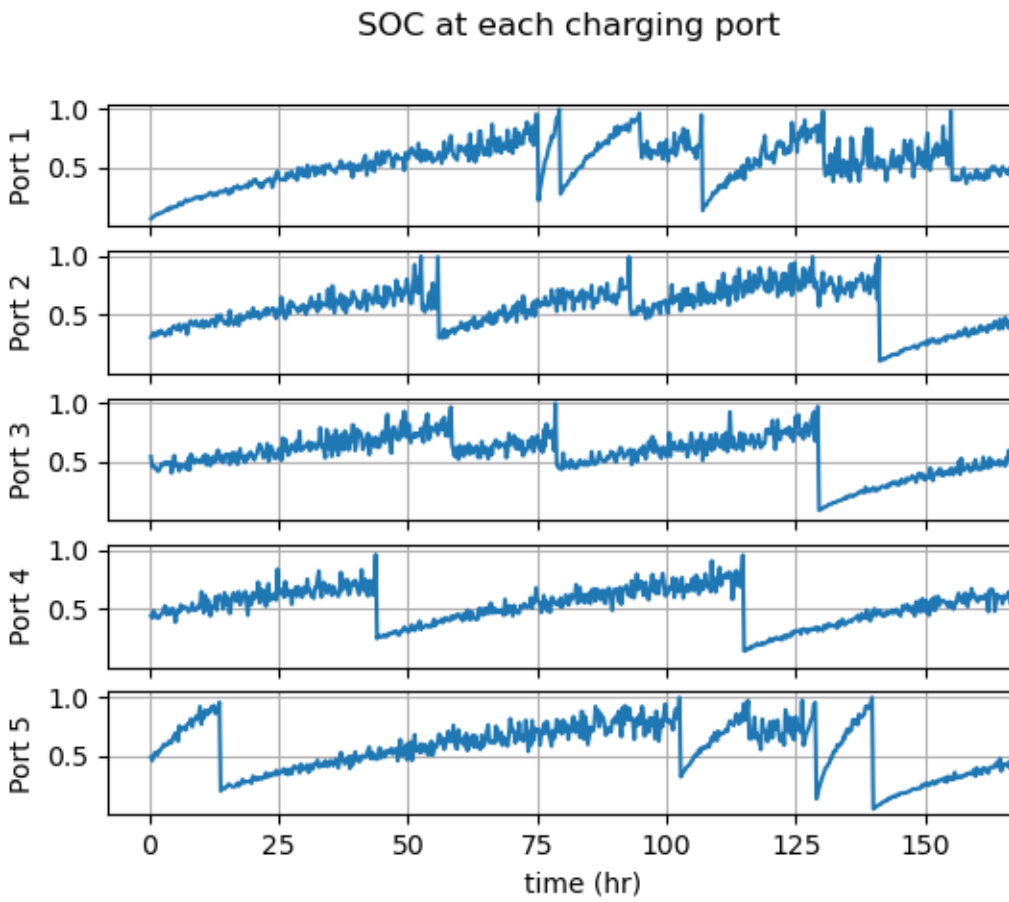
The results of this example are identical to those of the Advanced Default example as there is no impact on the simulated results, just the wall-clock time it takes to run the simulation.

Running the example

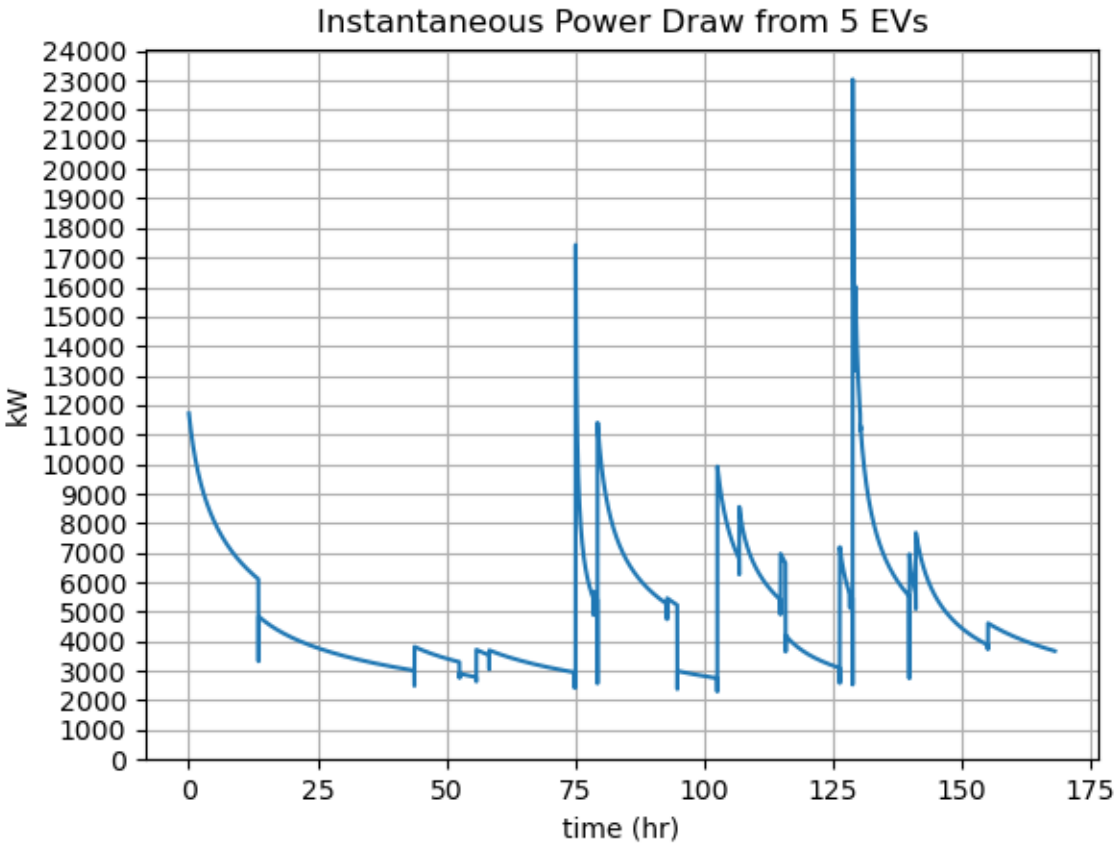
The example can be found in the HELICS Examples repopsitory.

To run the example:

```
$ helics run --path=advanced_async_runner.json
```



And the results should look like this:



Questions and Help

Do you have questions about HELICS or need help?

1. Come to office hours!
2. Post on the gitter!
3. Place your question on the github forum!

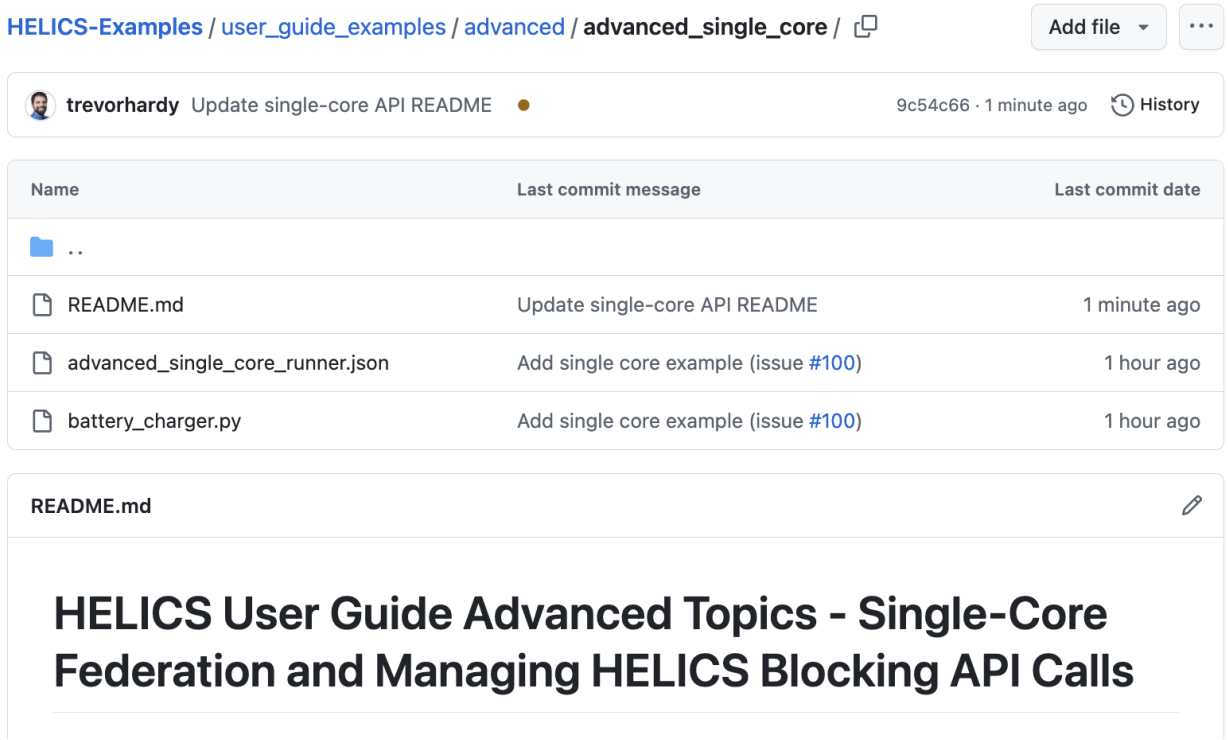
Multiple Federates on a Single Core and Managing Blocking APIs

This example demonstrates the capability of HELICS to support multiple federates on a single HELICS core. Along the way, it uses the "Async" variations of several HELICS APIs to manage multiple federates in a single executable.

- *Where is the code?*
- *What is this co-simulation doing?*
  - *Differences compared to the Fundamental Default example*
  - *HELICS differences*
- *Execution and Results*
- *Questions and Help*

Where is the code?

The example for using multiple federates on a single HELICS core can be found here. If you have issues navigating the examples, visit the HELICS Gitter page or the user forum on GitHub.



What is this co-simulation doing?

This example demonstrates the HELICS capability of supporting multiple HELICS federates on a single HELICS core. A HELICS core is the connection between a HELICS federate (an instance of a simulator) and the rest of the co-simulation. See the *HELICS Terminology* page for further details. In most cases, each federate is on its own core; this is the default behavior when creating a federate via a JSON config or the HELICS APIs. It is possible, though, to have multiple federates use the same core.

The HELICS auto-test suite commonly does this as a convenient way to create a federation in a single executable file. As this example shows, managing multiple federates in a single executable creates complications in managing parallel code paths. In fact, it could be argued that the point of HELICS is to help avoid the challenges of such manual management. Nevertheless, the capability exists and can be helpful in some use cases.

Additionally, this example demonstrates the use of HELICS "Async" API calls that are useful in working around the blocking calls that are common in HELICS federates, with `helicsFederateRequestTime()` being the most common. By using the "Async" variations of these API calls, it is possible to further parallelize the execution of multiple federates within a single executable.

Differences compared to the Fundamental Default example

Compared to the Fundamental Default example, the Battery and Charger functionality have been integrated into a single executable. The physics being simulated in each federate is maintained, but the order of execution within the federate code and the APIs used have been modified. There are still two federates as far as HELICS is concerned, but they both use the same HELICS core.

In this example, there is no federation outside of the Battery and Charger federates, so integrating the only two federates

into a single executable is not something you would normally need to do. This is an example of a capability.

HELICS differences

The source code for the integrated Battery-Charger federate shows that configuration is done via APIs to allow both federates to use the same core. To achieve this, all that is necessary is for the `federateInfo` object to have identical core names for every federate on the same core; this is achieved in this example by using the `helicsFederateInfoSetCoreName()` API. In this example, the core name is arbitrarily chosen as `common_core`.

One of the big challenges of putting multiple federates into a single executable is managing the blocking calls in HELICS: `helicsFederateEnterInitializingMode()`, `helicsFederateEnterExecutingMode()`, and `helicsFederateRequestTime()`. Any time a blocking call is made, execution halts until HELICS responds. For example, if the Battery federate in this single executable calls `helicsFederateRequestTime()`, execution halts until HELICS determines the time it can grant Battery. If HELICS needs Charger to also request a time before it can determine the time to grant Battery, and if the Charger time request occurs after Battery's, the executable can deadlock.

To resolve this deadlock, HELICS provides "Async" variants of these API calls. In these calls, a federate makes the API call, such as `helicsFederateRequestTimeAsync()`, and control is immediately returned to the code. No return value, such as the granted time, is provided at that point. From the HELICS standpoint, the time request for that federate has been made, with the granted time to be determined and communicated later. Because control returns immediately, this example can make a similar time request for the other federate.

So how does HELICS communicate the granted time to a federate that used `helicsFederateRequestTimeAsync()`? When the federate is ready to complete its time request, it calls `helicsFederateRequestTimeComplete()`; this is a blocking call and returns the granted time. Federates can also ask HELICS if a time grant has been calculated and is ready for the federate via `helicsFederateIsAsyncOperationCompleted()`; this works for all "Async" API calls.

The "Async" versions of the API calls are useful for other reasons. Say you have a federate that, after being granted time, getting updated values on its inputs, and updating its internal model, still has some bookkeeping to do: putting values in a database or calculating metrics for later post-processing. Rather than holding up the entire federation while it does this administrative work, it can use `helicsFederateRequestTimeAsync()` to make the time request, proceed to do its bookkeeping, and then call `helicsFederateRequestTimeComplete()` when the administrative tasks are complete.

In this example, there is a further wrinkle that complicates the sequencing of the time grants: the original Fundamental Default example had the Battery federate set the `wait_for_current_time_update` flag. This flag indicates that this federate needs to be the last federate granted a given simulation time. That is, HELICS needs to guarantee that all other federates have been granted a future time and that whatever values they produce at the simulation time in question have taken place. The use of this flag complicates the execution of these two federates in a single executable.

To achieve proper execution of the two federates with `wait_for_current_time_update` set by the Battery federate, a more complicated execution pattern must take place. Here's some pseudo-code to give the general idea:

```
h.helicsFederateRequestTimeAsync(charger_fed, charger_requested_time)
while charger_grantedtime < total_interval:
  h.helicsFederateRequestTimeAsync(battery_fed, battery_requested_time)
  charger_grantedtime = h.helicsFederateRequestTimeComplete(charger_fed)
  charger_update_internal_model()
  charger_publish_updated_values()
  h.helicsFederateRequestTimeAsync(charger_fed, charger_requested_time)
  battery_grantedtime = h.helicsFederateRequestTimeComplete(battery_fed)
  battery_update_internal_model()
  battery_publish_updated_values()
```

As you can see in the code, Charger must update its internal model to complete its work at a given simulation time and request the next simulation time before Battery is able to be granted the previous simulation time. This means that Charger must update its internal model, publish values, and request the next time between Battery's

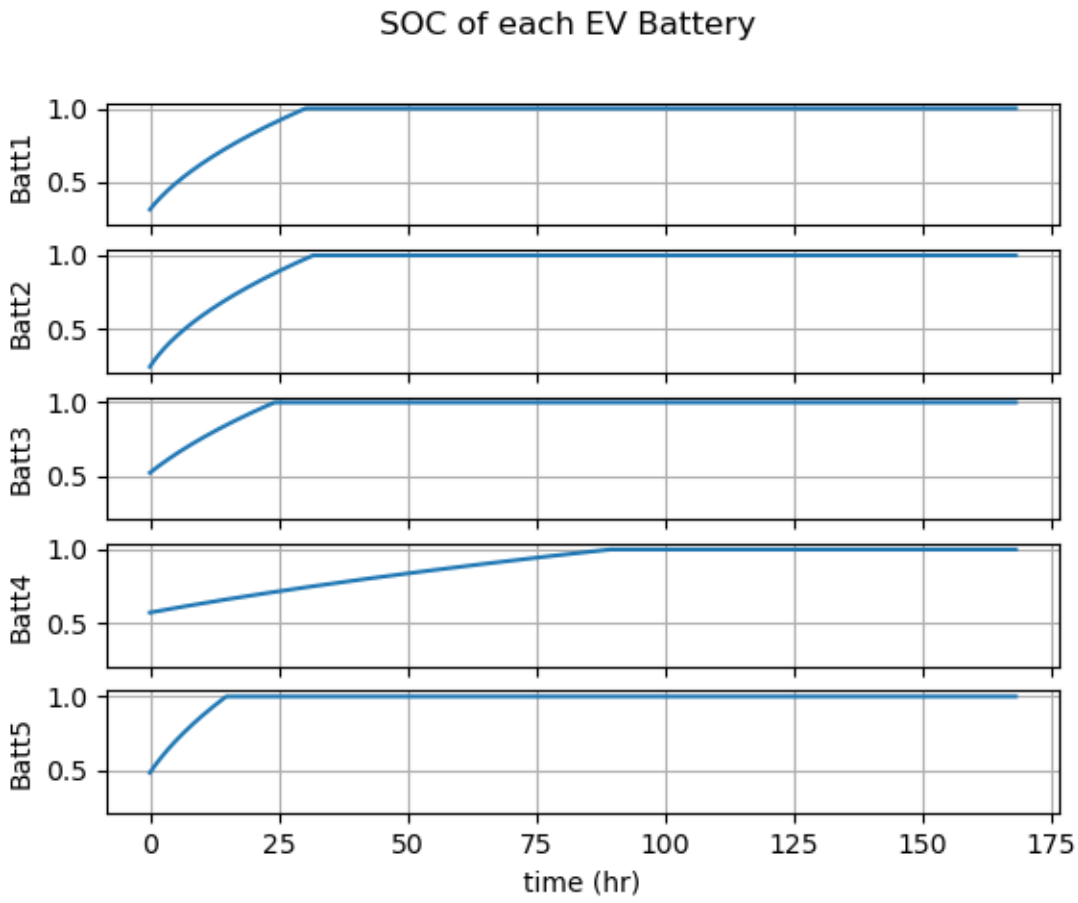
`helicsFederateRequestTimeAsync()` and `helicsFederateRequestTimeComplete()`. Similar complications exist at the beginning of execution around entering executing mode and at the end of the co-simulation when killing off the federates.

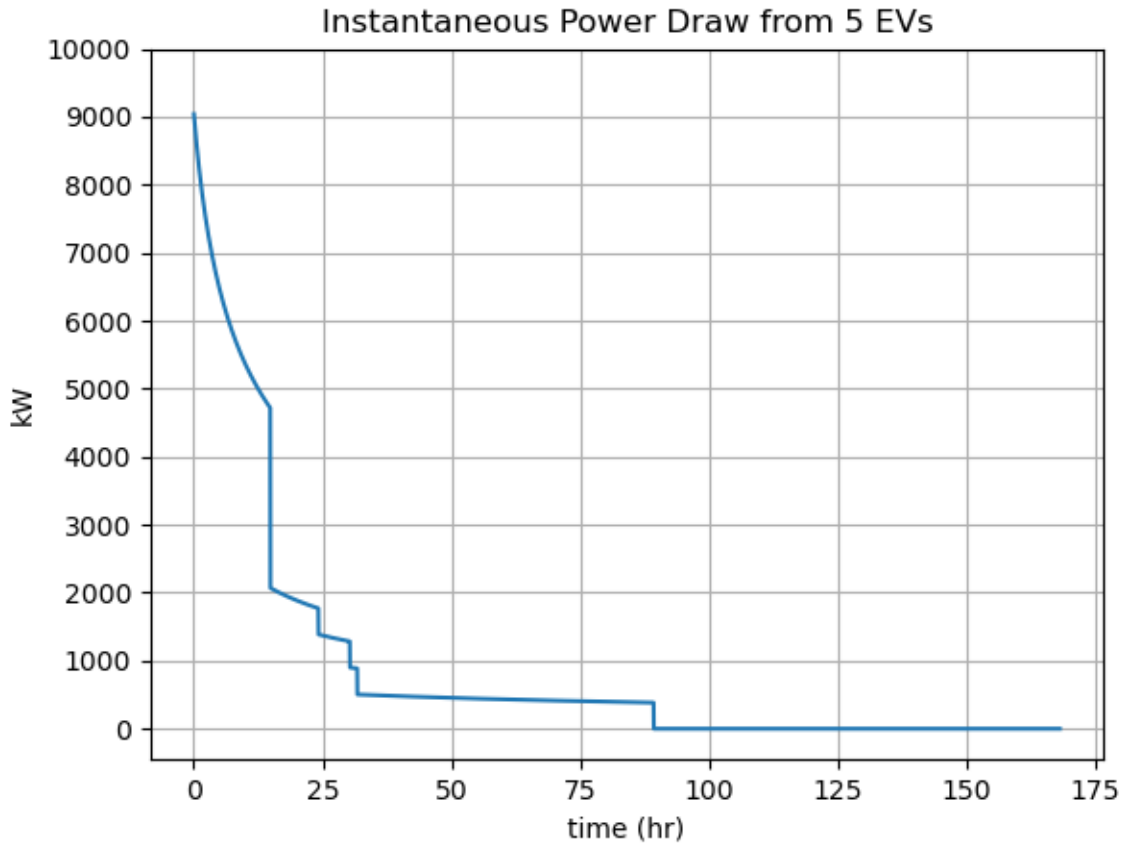
Execution and Results

Run the co-simulation:

```
helics run --path=./single_core_runner.json
```

As designed, the results of this co-simulation are identical to those of the Fundamental Default example.





Questions and Help

Do you have questions about HELICS or need help?

1. Come to office hours!
2. Post on the gitter!
3. Place your question on the github forum!

helics_connector

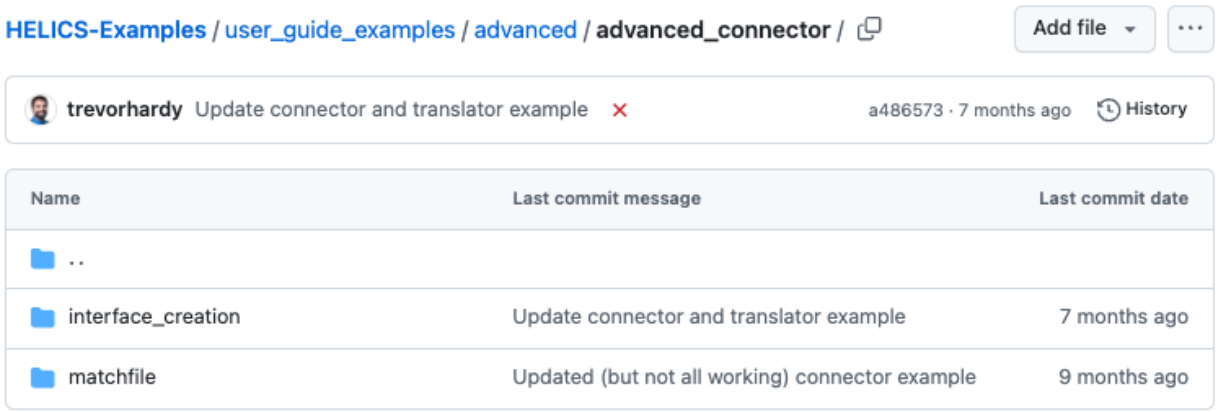
This example demonstrates the use of the "helics_connector" app to create interface connection between federates outside of the configuration JSON or API calls.

- *helics_connector*
  - *Where is the code?*
  - *What is this co-simulation doing?*
    - *Differences compared to the Advanced Default example*
      - *HELICS Differences*
    - *HELICS components*
      - *Configuration by match-file*
      - *Match-file specification*

        - *Configuration by interface creation*
        - *Query callback*
        - *Configuration command response*
        - *Interface creation timing*
    - *Execution and Results*
    - *Questions and Help*

Where is the code?

Example code for using `helics_connector` can be found here. If you have issues navigating the examples, visit the HELICS Gitter page or the user forum on GitHub.



What is this co-simulation doing?

This example shows how to use helics_connector to create connections between HELICS federate interfaces without specifying said connections via the JSON configuration or similar APIs. Specifically, only publications (with no targets defined), outputs (with no source defined), and endpoints need to be defined in the JSON. helics_connector then uses a match-file to establish the connections between publication outputs and inputs.

Differences compared to the Advanced Default example

This example has the same federates interacting in the same ways as in the *Advanced Default example*. The only difference is the use of helics_connector to create the data exchanges between the federates.

HELICS Differences

By configuring the federates without any targeting in the input or output, helics_connector uses existing public APIs to establish the data exchanges between federates. No core library changes were made to add this functionality and it is entirely possible for other HELICS users to write their own versions of helics_connector to satisfy their particular needs.

HELICS components

Configuration by match-file

The only difference between the federation as defined in the "advanced default" example and this example is how the configuration is defined. The advanced_default example uses subscriptions instead of inputs in the configuration JSON

while this example uses inputs. For example, a portion of the Battery federate configuration for the advanced_default example and this example are shown below; the advanced_default configuration uses subscriptions and specifies a "key" as a data source where this example uses inputs and has no "key" value.

```
"subscriptions":[
    {
      "key":"Charger/EV1_voltage",
      "type":"double",
      "unit":"V",
      "global": true
    }
]
```

```
"inputs": [
    {
      "global": true,
      "name": "Battery/EV100_input_voltage",
      "type": "double",
      "unit": "V"
    }
]
```

Match-file specification

The match-file can be specified either using a plain textfile or JSON. The simplest format specifies the individual connections between federates, one connection per line:

```
Charger/EV1_output_voltage Battery/EV1_input_voltage from_to
Charger/EV2_output_voltage Battery/EV2_input_voltage from_to
Charger/EV3_output_voltage Battery/EV3_input_voltage from_to
Charger/EV4_output_voltage Battery/EV4_input_voltage from_to
Charger/EV5_output_voltage Battery/EV5_input_voltage from_to
Charger/EV1_input_current Battery/EV1_output_current to_from
Charger/EV2_input_current Battery/EV2_output_current to_from
Charger/EV3_input_current Battery/EV3_output_current to_from
Charger/EV4_input_current Battery/EV4_output_current to_from
Charger/EV5_input_current Battery/EV5_output_current to_from
```

In JSON, the format looks like:

```
{
  "connections": [
    ["Charger/EV1_output_voltage", "Battery/EV1_input_voltage", "from_to"],
    ["Charger/EV2_output_voltage", "Battery/EV2_input_voltage", "from_to"],
    ["Charger/EV3_output_voltage", "Battery/EV3_input_voltage", "from_to"],
    ["Charger/EV4_output_voltage", "Battery/EV4_input_voltage", "from_to"],
    ["Charger/EV5_output_voltage", "Battery/EV5_input_voltage", "from_to"],
    ["Charger/EV1_input_current", "Battery/EV1_output_current", "to_from"],
    ["Charger/EV2_input_current", "Battery/EV2_output_current", "to_from"],
    ["Charger/EV3_input_current", "Battery/EV3_output_current", "to_from"],
    ["Charger/EV4_input_current", "Battery/EV4_output_current", "to_from"],
    ["Charger/EV5_input_current", "Battery/EV5_output_current", "to_from"]
  ]
```

(continues on next page)

(continued from previous page)

```
}
```

The "from-to" or "to-from" is used to indicate which side of the connection is un-named and thus is the producer of the data ("from") and which side is named and is the target for the data ("to"). helics_connector goes through and finds all **unconnected** interfaces on the federates themselves, finds the corresponding interface indicated as a "from" connection in the match-file, and then connects them with any existing entry in the match-file on the "to" side (even if they already have a target).

Alternatively, the match-file can be specified using regular expressions. This can be a convenient way to specify a large number of interface connections if they follow a naming convention. The regular expression format takes advantage of the ability to name terms in the expression, allowing terms in one part of the expression to be used later on in the expression. In the case, the above match-file looks like this:

```
# this is a comment
REGEX:Charger/EV(?<ev_num>\d*)_output_voltage, REGEX:Battery/EV(?<ev_num>)_input_voltage,
↪ from_to
REGEX:Charger/EV(?<ev_num>\d*)_input_current, REGEX:Battery/EV(?<ev_num>)_output_current,
↪ from_to
```

In this case, "ev_num" is the name given to the numerals that appear after the characters "EV" and are used in both first and second terms of the match-file. Each regular expression must be prefixed with the "REGEX:" string.

Its possible to mix direct connections and regular expressions in the same matchfile but not because the helics_connector only works on unconnected interfaces, there is an implied precedence in the match-file that works top down. That is, connections at the top of the file over-ride those specified at the bottom of the file. Once an interface marked as a "from" is connected, it is not eligible for connections defined later in the file.

Configuration by interface creation

A second, more complex example is also included where helics_connector facilitates a custom communication protocol between federates where the interfaces themselves are created on demand. In this case, the federates using this protocol must be written to respond to a specific query where they indicate the interfaces they could create and then respond to a specific command from helics_connector by creating the interfaces indicated by helics_federate. In both cases helics_connector uses a match-file specification to define which interfaces to create.

The configuration JSON of the federate is quite different as no interfaces are defined and only timing information is included:

```
{
  "name": "Battery",
  "core_type": "zmq",
  "log_level": "warning",
  "period": 60.0,
  "uninterruptible": false,
  "terminate_on_error": true,
  "wait_for_current_time_update": true
}
```

Query callback

The federate code needs to include the necessary functionality to define a callback to handle custom queries. The example shown here is in Python and the HELICS Python library is C-based so setting up the callback correctly is more challenging than most of the Python code in these examples.

To pass information into the callback (which is never explicitly called and thus it is not possible to pass specific parameters), a "UserData" class is defined that allows the inclusion of arbitrary data (in this case, the number of EVs). This has been defined as a value that is passed in when the object is instantiated ("num_EVs"). This value is used to define a class attribute, thus preserving it in the object itself.

```
class UserData(object):
    def __init__(self, num_EVs):
        self.num_EVs = num_EVs
        userdata = h.ffi.new_handle(self)
        self._userdata = userdata
```

The function that is called when the query is made. In our example, we called it "query_callback" and it has a decorator of the callback function in C that must match the signture of said function.

```
@h.ffi.callback(
    "void query(const char *query, int querySize, HelicsQueryBuffer buffer, void *user_
↪data)"
)
def query_callback(query_ptr, size: int, query_buffer_ptr, user_data):
    ...
```

The query being sent to this federate is defined in the "query_buffer_ptr" variable but must be decoded into a string by declaring it a HELICS buffer object and then decoding it into a string:

```
query_buffer = h.HelicsQueryBuffer(query_buffer_ptr)
query_str = h.ffi.string(query_ptr, size).decode()
```

After doing all this work, we can see that the query sent is "potential_interfaces"; this is the query sent by helics_connector to find out what potential interfaces each federate can create. To answer this query we need to pull in the number of EVs passed in through the UserData object

```
num_EVS = h.ffi.from_handle(user_data).num_EVs
```

The query response is a JSON dictionary composed of three lists: the federates publications, inputs, and endpoints. In this case, the name of the interfaces can be created easily but must match the names used in the helics_connector match-file. The dictionary is used to fill the HELICS buffer object that was sent in, effectively responding to the query.

```
for EVnum in range(1, num_EVs + 1):
    pubs.append(f"Battery/EV{EVnum}_output_current")
    inputs.append(f"Battery/EV{EVnum}_input_voltage")
response_dict = {"publications": pubs, "inputs": inputs, "endpoints": []}
query_response = json.dumps(response_dict)
h.helicsQueryBufferFill(query_buffer, query_response)
```

With "UserData" and "query_callback" defined, they now just need to be included in the main body of the federate code. An instance of the UserData object is created ("user_data") using the number of EVs and a handle (pointer) to the object is defined. Lastly, the query callback is defined via a HELICS API, referencing both "query_callback" and "user_data".

```
user_data = UserData(num_EVs)
user_data_handle = h.ffi.new_handle(user_data)
h.helicsFederateSetQueryCallback(fed, query_callback, user_data_handle)
```

Configuration command response

After responding to the query, the federate also needs to respond to the command sent by helics_connection to define the interfaces it needs to create. helics_connector sends a JSON as a command with the "command" field of that object being "register_interfaces". The rest of the object has three fields: the publications, inputs, and endpoints the object needs to create.

```
{
  "command": "register_interfaces",
  "inputs": [
    "Battery/EV5_input_voltage",
    "Battery/EV4_input_voltage",
    "Battery/EV3_input_voltage",
    "Battery/EV2_input_voltage",
    "Battery/EV1_input_voltage"
  ],
  "publications": [
    "Battery/EV5_output_current",
    "Battery/EV4_output_current",
    "Battery/EV3_output_current",
    "Battery/EV2_output_current",
    "Battery/EV1_output_current"
  ]
}
```

The example code checks to make sure the interfaces are provided as lists and then creates the interfaces as a part of the custom "register_interfaces_from_command" function. In this case, we assume all inputs are doubles; a more general command could include data types.

```
if isinstance(cmd["publications"], list):
    for pub in cmd["publications"]:
        h.helicsFederateRegisterGlobalPublication(fed, pub, h.HELICS_DATA_TYPE_DOUBLE)
if isinstance(cmd["inputs"], list):
    for inp in cmd["inputs"]:
        h.helicsFederateRegisterGlobalInput(fed, inp, h.HELICS_DATA_TYPE_DOUBLE)
```

Interface creation timing

The above defined query response and command protocol all takes place during the initialization phase of the federate operation. To accomplish multiple asynchronous events in initializing mode, the "iterative" version of entering initialization mode needs to be used. Due to the way HELICS grants time in initializing mode, the query is guaranteed to be available after enter initializing mode. This is handled via the callback and not explicit call needs to be made. To guarantee the command message is available for acting on, the same iterative initializing mode call is made again. After this, the command is deciphered, the interfaces created, and, finally, the federate enters executing mode.

```
h.helicsFederateEnterInitializingModeIterative(fed)
# Query is guaranteed to be available after this call but may be
# available earlier. Callback responds whenever the query comes in.
h.helicsFederateEnterInitializingModeIterative(fed)
command = h.helicsFederateGetCommand(fed)
if len(command) == 0:
    raise TypeError("Empty command.")
try:
    logger.debug(f"command string: {command}")
```

(continues on next page)

(continued from previous page)

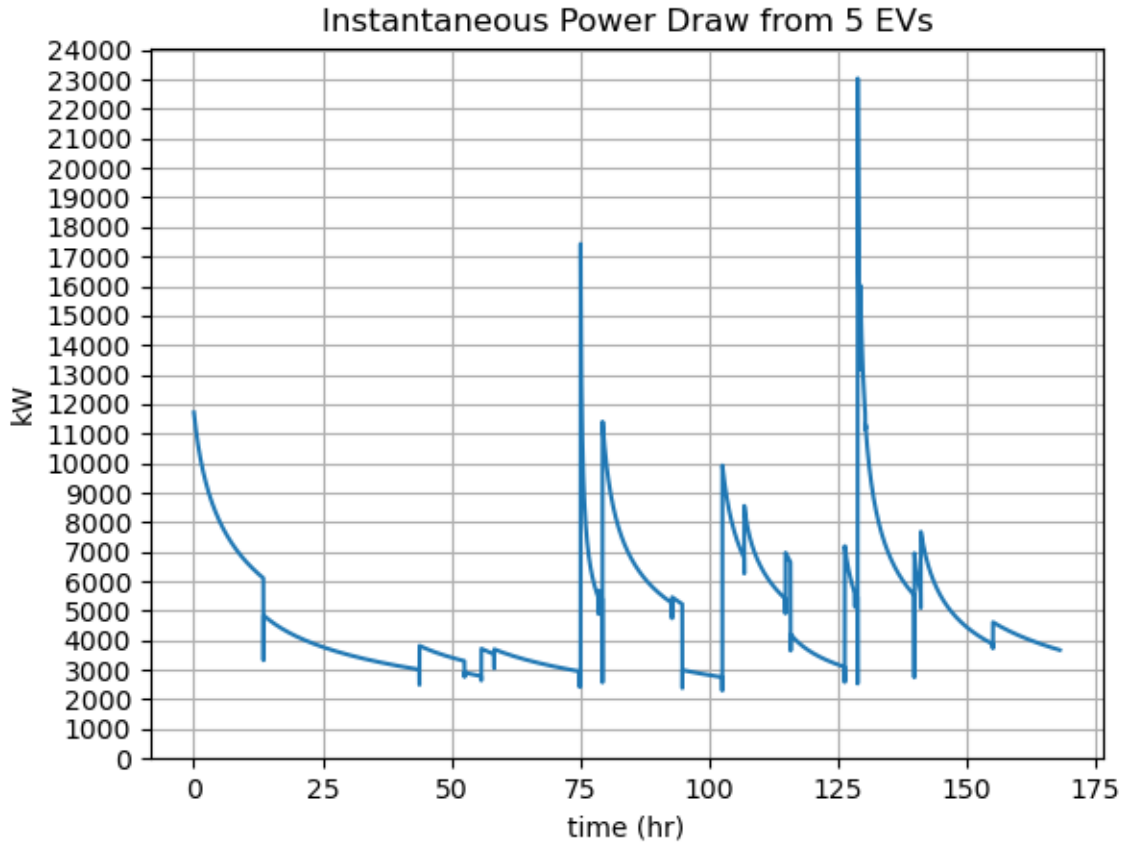
```
    cmd = json.loads(command)
except:
    raise TypeError("Not able to convert command string to JSON.")
register_interfaces_from_command(fed, cmd)  # custom function
h.helicsFederateEnterExecutingMode(fed)
```

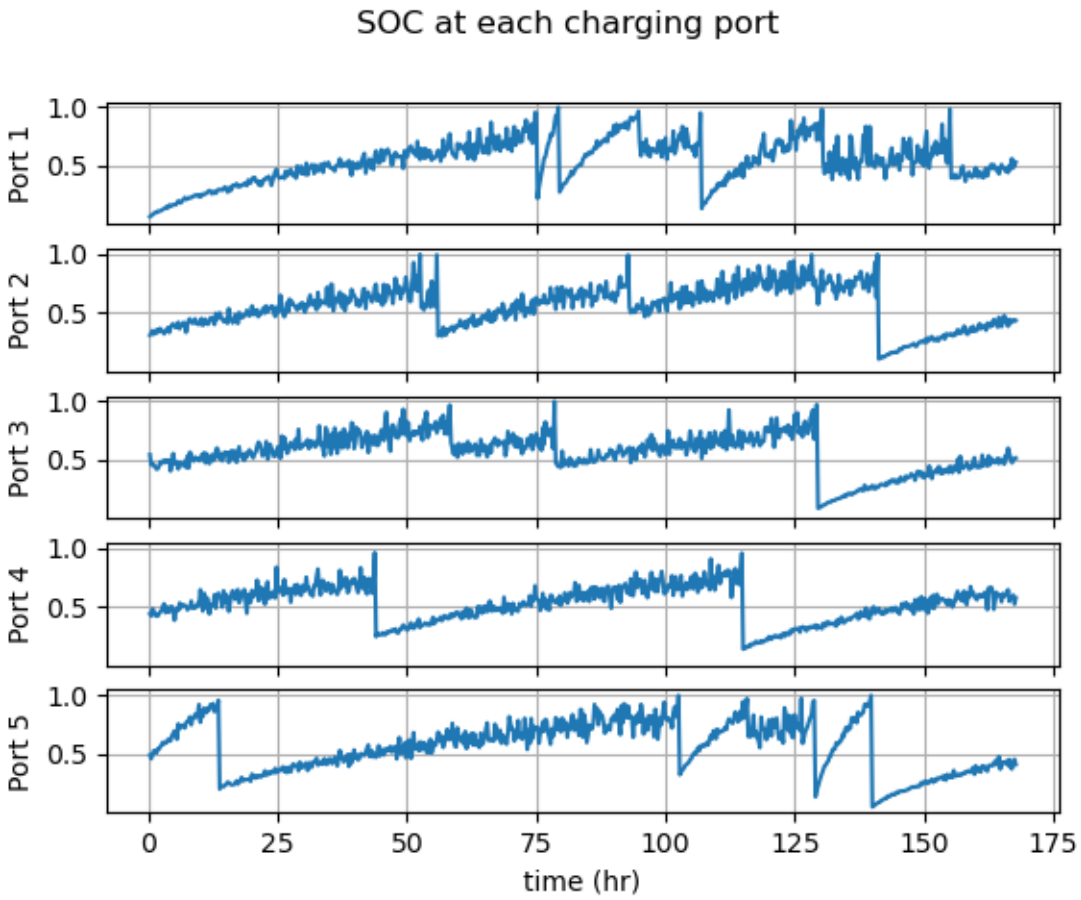
Execution and Results

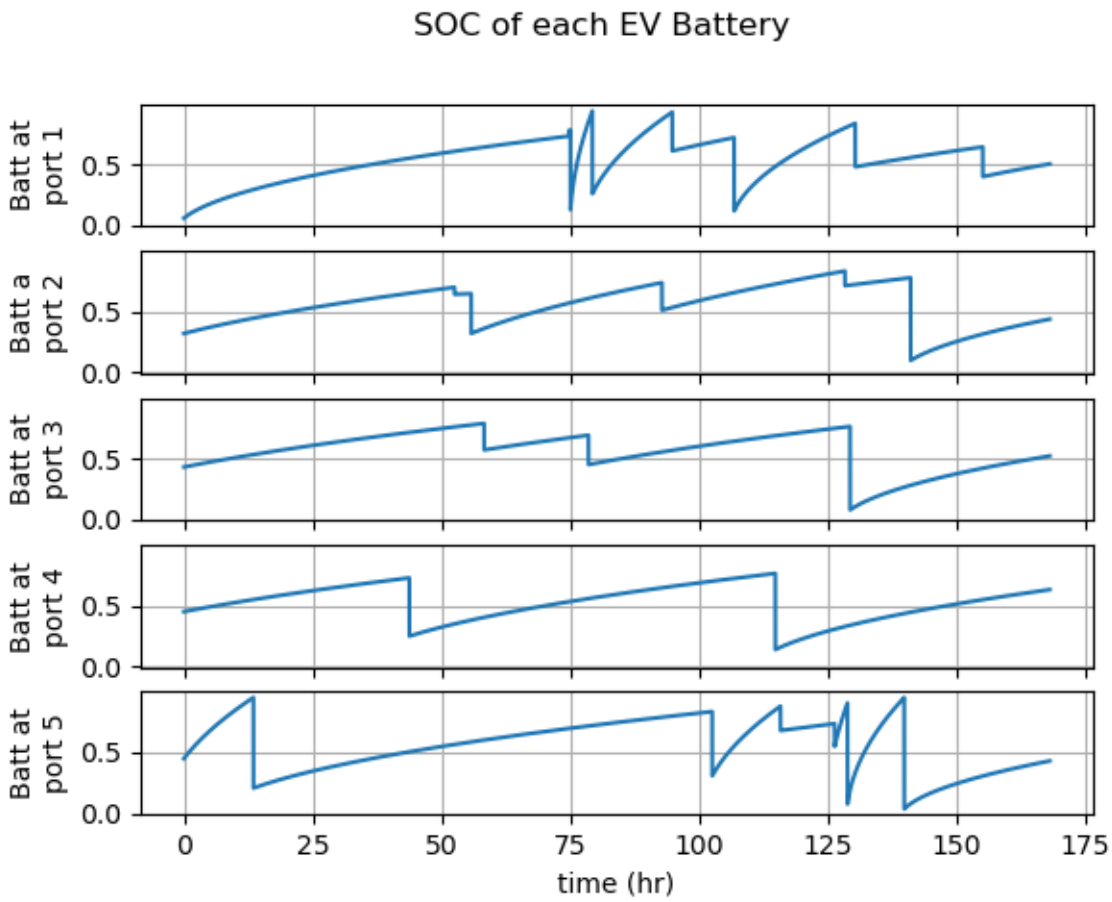
Run the co-simulation, use one of the following:

```
$ helics run --path=./advanced_matchfile_direct_runner.json
$ helics run --path=./advanced_matchfile_regex_runner.json
$ helics run --path=./advanced_connector_interface_creation_runner.json
```

Since this is only a change to the configuration method of the federation, the results are identical to those in the *Advanced Default example.*




SOC at each charging port
Port 1
Port 2
Port 3
Port 4
Port 5
1.0
0.5
0
25
50
75
100
125
150
175
time (hr)



Questions and Help

Do you have questions about HELICS or need help?

1. Come to office hours!
2. Post on the gitter!
3. Place your question on the github forum!

Dynamic Federations

This example demonstrates the capability of HELICS to allow federates to join the federation after simulation time zero has been granted and to leave the federation before other federates complete.

- *Where is the code?*
- *What is this co-simulation doing?*
  - *Differences compared to the Advanced Default example*
  - *HELICS differences*
  - *Implementation*
- *Execution and Results*
- *Questions and Help*

Where is the code?

The example code for showing the operation of a dynamic federation can be found here. If you have issues navigating the examples, visit the HELICS Gitter page or the user forum on GitHub.

What is this co-simulation doing?

HELICS can allow federates to join the federation late (after others have already entered executing mode and been granted simulation time zero) and leave early. This example demonstrates that functionality. In this case, EV 5 joins the co-simulation late and leaves the co-simulation early.

Differences compared to the Advanced Default example

Compared to the Advanced Default example, this example has EV 5 joining the federation late and leaving the federation early.

HELICS differences

HELICS has always allowed a federate to leave a federation early. That is, it has never been a requirement that all federates end their part of the co-simulation at the same simulated time. In most cases, federates will leave the co-simulation at the same or nearly the same simulated time so the model retains consistency.

The ability for a federate to join a co-simulation late was first fully supported in HELICS v3.4.0. Joining a federation late is more disruptive to a co-simulation because HELICS must incorporate any new publications, subscriptions, and endpoints into the dependency graph that it uses to determine which federates should be granted time. There is a computational cost to having federates join the federation and recalculate the dependency graph, so there can be a non-trivial performance cost depending on the details of the federation.

To enable full dynamic co-simulation, `helics_broker` needs to be started with the command-line `--dynamic` switch. For this example, the full command line looks like:

```
helics_broker -f3 --dynamic
```

Implementation

`Battery.py` has been modified to model both the EVs that join the federation at the beginning of the co-simulation and EV 5, which joins late. The `--late` flag is set for the latter. To help with data management and message passing, the variable `not_charging_value` is used as a semaphore to indicate that EV 5 is not attached to its charger, preventing `Charger.py` from misbehaving while EV 5 is not part of the co-simulation.

Ideally, a separate HELICS publication and subscription would be used to indicate whether the EV was present instead of using a semaphore. This would require enough rework in this example, and likely in the other examples to keep the suite consistent, that we decided not to undertake it. For more serious modeling efforts, adding such a signal would simplify the existing logic for Battery and Charger and make the code easier to understand and maintain.

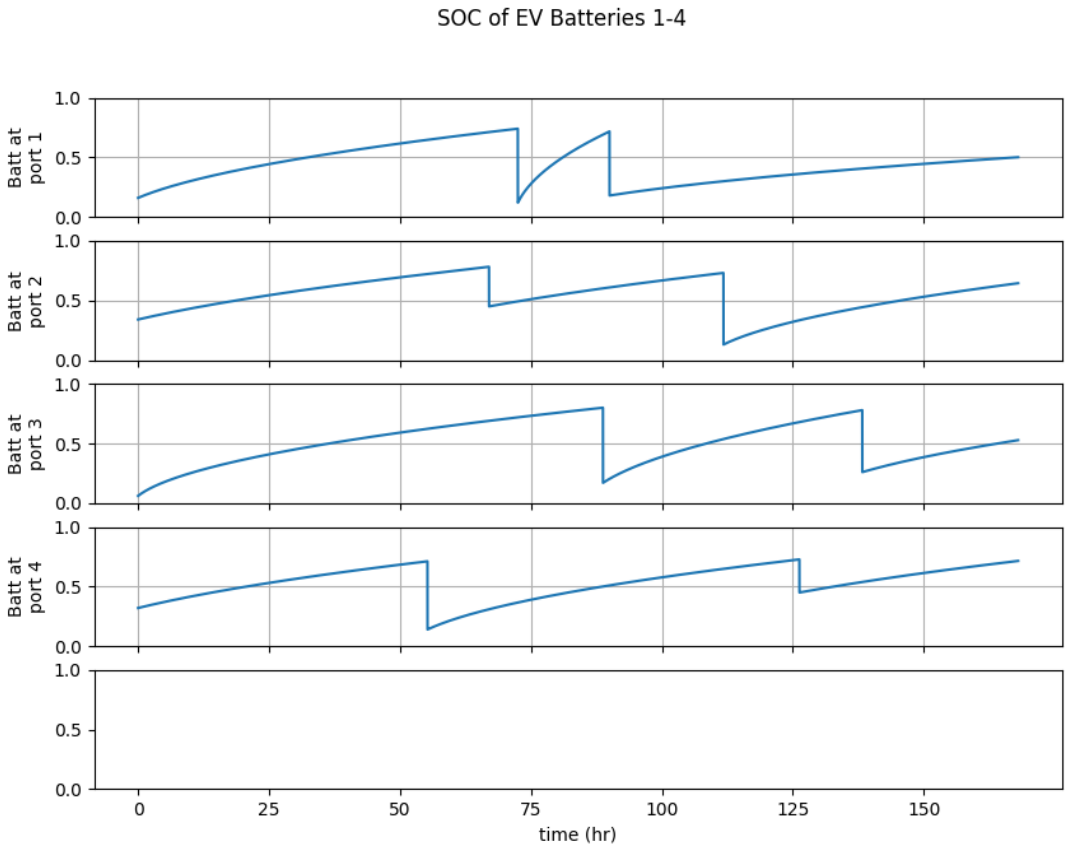
Execution and Results

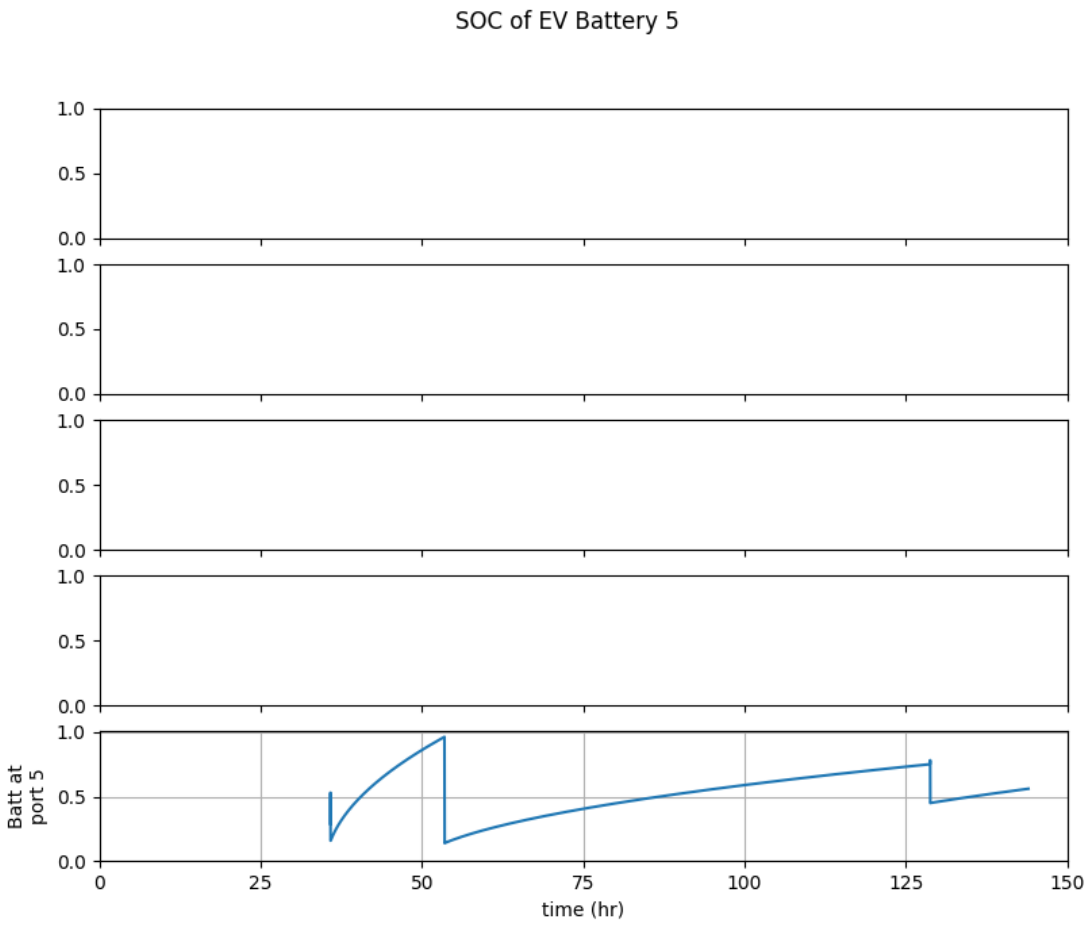
Run the co-simulation:

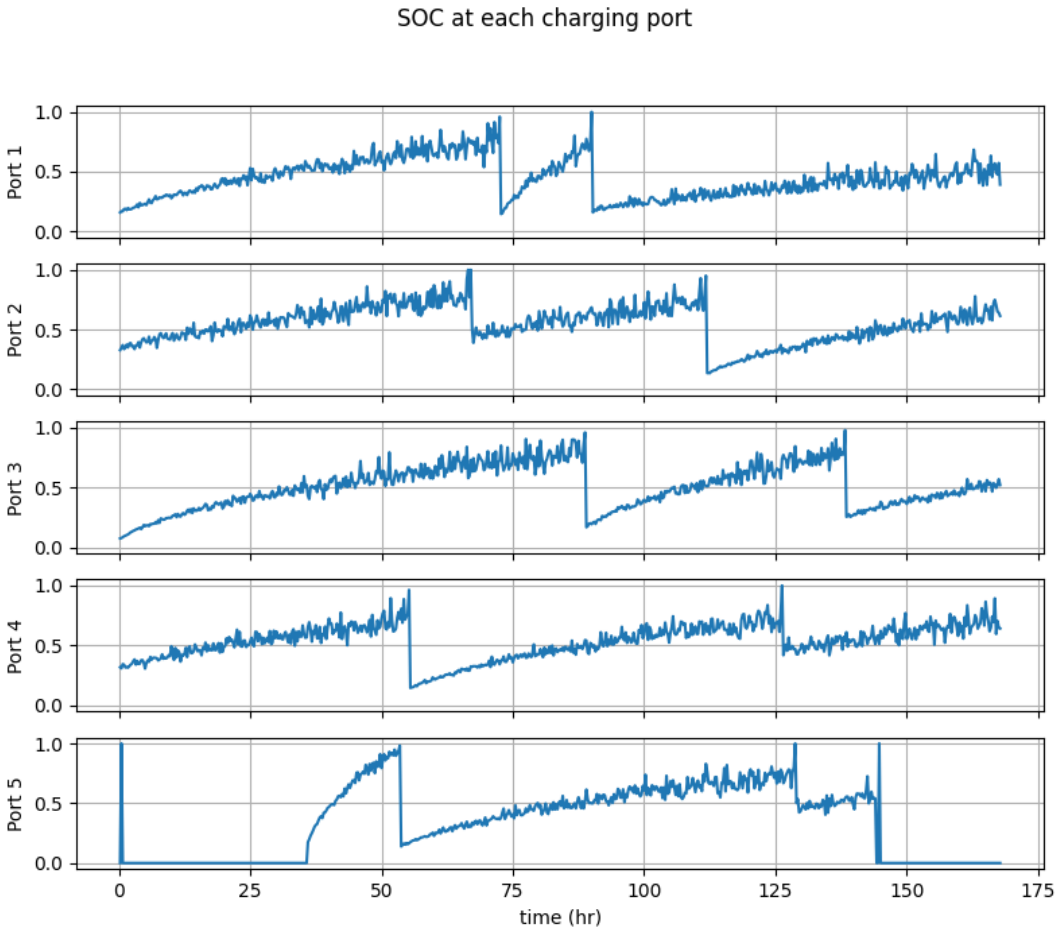
```
helics run --path=./dynamic_federation_runner.json
```

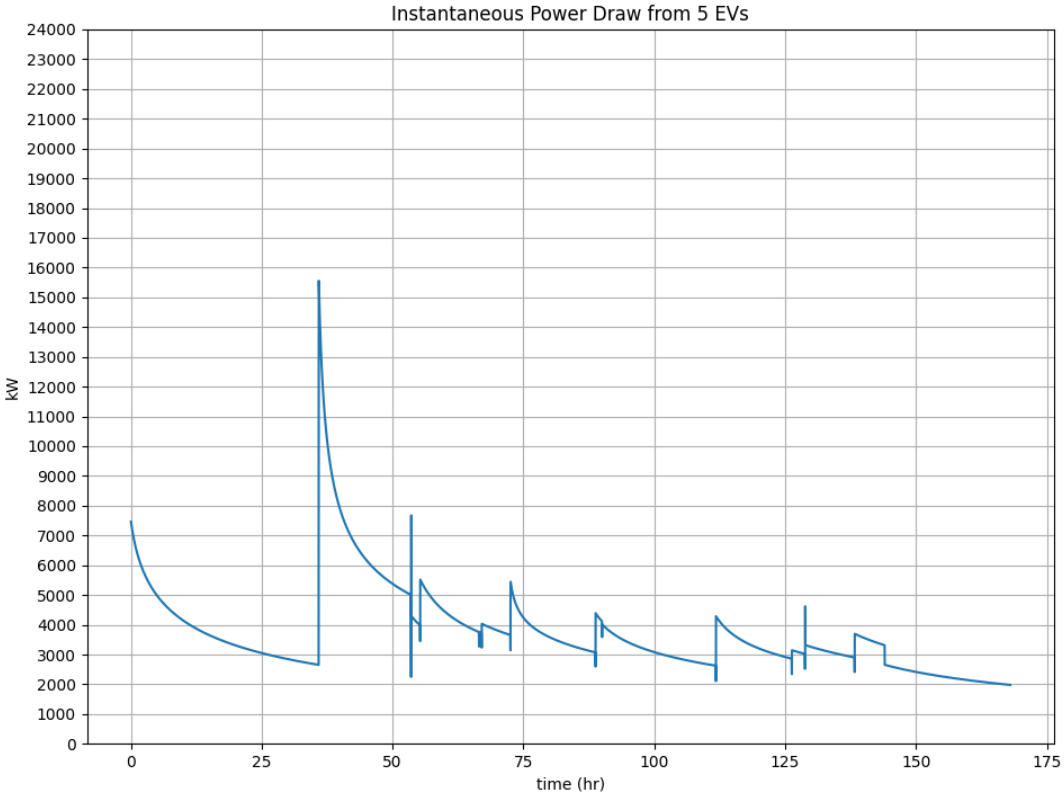
With EV 5 not being present for the full co-simulation, the resulting graphs differ from those in the *Advanced Default example*. Note that the x-axis showing the result of just the EV 5 SOC does not cover the same period of time as the graph showing EVs 1-4.

Also worth noting are the blips in the EV 5 SOC as shown in the bottom graph. This is not unexpected and is likely something that could be mitigated in federate code. If you're running a dynamic federation, managing the HELICS data exchanges at these transition points, as well as before the federate joins and after the federate leaves, will take extra consideration and effort.




SOC of EV Battery 5
1.0
0.5
0.0
Batt at port 5
0
25
50
75
100
125
150
time (hr)


SOC at each charging port
Port 1
Port 2
Port 3
Port 4
Port 5
1.0
0.5
0.0
0
25
50
75
100
125
150
175
time (hr)



Questions and Help

Do you have questions about HELICS or need help?

1. Come to office hours!
2. Post on the gitter!
3. Place your question on the github forum!

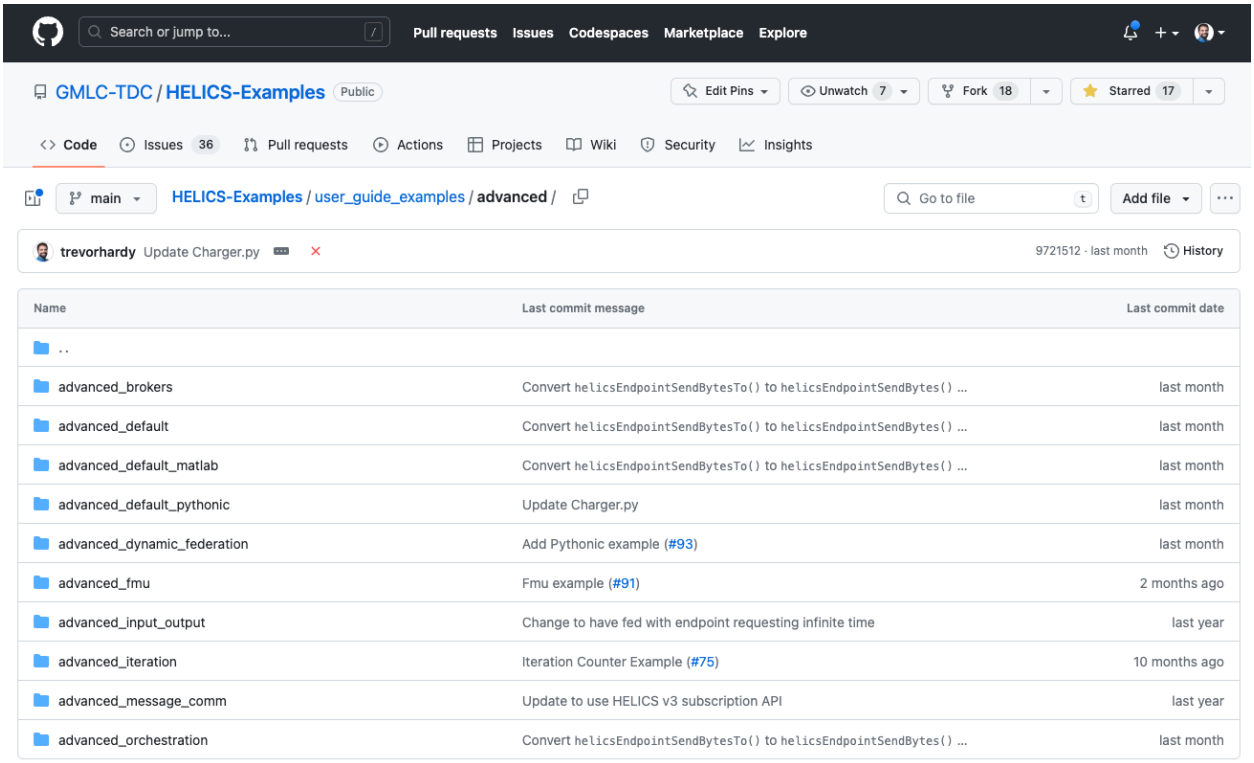
The Advanced Examples are modular, each building upon the *base example*. Users are encouraged to dive into a specific concept to build their HELICS knowledge once they have mastered the default setup in the *base example*.

This page describes the model for the Advanced Examples. This model is topically the same as the Fundamental Examples, but more complex in its execution. This allows the research question to become more nuanced and for the user to define new components to a HELICS co-simulation.

- Where is the Code?
- What is this Co-simulation Doing?
  - Differences Compared to the Fundamental Examples
    - HELICS Differences
    - Research Question Complexity Differences
- HELICS Components
  - Federates with infinite time

- Initial time requests and model initialization

The code for the Advanced examples can be found in the HELICS-Examples repository on github. If you have issues navigating the examples, visit the HELICS Gitter page or the user forum on GitHub.



The Advanced Examples are similar in theme to the *Base Example* in that both are looking at power management for an EV charging garage. The implemented federates, however, are slightly more sophisticated and include a new centralized charging controller federate.

- **Battery.py** - Models a set of the EV batteries being charged. The EV is randomly assigned to support a particular charging level and receives an applied charging voltage based on that level. Using the applied voltage and the current SOC (initially randomly assigned), a charging current is calculated returned to the charger.
- **Charger.py** - Models a set of EV charging terminals all capable of supporting three defined charging levels: level 1, 2, and 3. Applies a charging voltage based on the charging terminal power rating and (imperfectly) measures the returned current. Based on this current, it estimates the SOC and sends that information to the controller. When commanded to terminate charging it removes the applied charging voltage.
- **Controller.py** - Receives periodic updates about the SOC of each charging vehicle and when they are considered close enough to full, command the charger to terminate charging.

Every time charging is terminated on an EV, a new EV to take its place is randomly assigned a supported charging level and initial SOC.

There are a few important distinctions between the Fundamental Examples and the Advanced Examples, which can be grouped into **HELICS** differences and **research question complexity** differences.

1. **Communication:** Both physical value exchanges and abstract information exchanges are modeled. The exchange of physical values takes place between the Battery and Charger federates (this was also introduced in a slimmed-down fashion in the *Fundamental Communication Example*). The message exchange (control signals, in this case) takes place between the Charger and Controller federates. For a further look at the difference between these two messaging mechanisms see our User Guide page on *value federates* and *message federates.*

2. **Timing:** The Controller federate has no regular update interval. The Controller works in pure abstract information and has no regular time steps to simulate. As such, it requests a maximum simulated time supported by HELICS (`HELICS_TIME_MAXTIME`) and makes sure it can be interrupted by setting `uninterruptible` to `false` in its configuration file. Any time a message comes in for the Controller, HELICS grants it a time, the Controller performs the required calculation, sends out a new control signal, and requests `HELICS_TIME_MAXTIME` again.

In the *Fundamental Base Example*, a similar research question is being addressed by this co-simulation anlaysis: estimate the **instantaneous power draw** from the EVs in the garage. And though you may have similar questions, there are several complicating changes in the new model:

1. A **third federate** (where previously there were only two) models responsibilities of a charger. The charger stops charging the battery by removing the charging voltage rather than the battery stopping the charging process. The Battery federate synthesizes an EV battery when the existing EV is considered fully charged.

2. The measurement of the charging current (used to calculate the actual charging power) has some **noise** built into it. This is modeled as random variation of the charging current in the federate itself and is a percentage of the charging current. The charging current decreases as the SOC of the battery increases leading to a noisier SOC estimate by the Charger federate at higher SOCs. This results in the Controller tending to terminate charging prematurely as a single sample of the noisy charging current can lead to over-estimation of the SOC.

3. We can now model both **physics** and **measurement of physics**. There are two SOC values modeled in this co-simulation: the "actual" SOC of the battery modeled in the Battery federate and the estimate of the SOC as measured by the Charger federate. Both federates calculate the SOC in the same manner: use the effective resistive load of the battery, R, and a pre-defined relationship between R and SOC. You can see that both the Battery and Charger federates use the exact same relationship between SOC and effective R (SOC of zero is equivalent to an effective resistance of 8 ohms; SOC of 1 has an effective resistance of 150 ohms). Due to the noise in the charger current measurement, there is error built into its calculation of the SOC and therefore should be considered an estimate of the SOC.

This existence of two values for one property is not uncommon and is as much a feature as a bug. If this system were to be implemented in actual hardware, the only way that a charger would know the SOC of a battery would be through some kind of external **measurement**. And certainly there would be times where the charger would have even less information (such as the specific relationship between SOC and effective resistance) and would have to use historical data, heuristics, or smarter algorithms to know how to charge the battery effectively. Simulation allows us to use two separate models and thus independently model the actual SOC as known by the battery and the estimated SOC as calculated by the charger.

Since the decision to declare an EV fully charged has been abstracted away from the Charger and Battery (to the Controller), a slightly different procedure is used to disconnect a charged EV from the charger and replace it with a new one to be charged. In this advanced example, a mini-protocol has been designed and implemented:

1. The Charger receives a message from the Controller indicating the EV should be considered fully charged.

2. The Charger reduces the Charging voltage to zero volts and publishes it.

3. The Battery, detecting this change in charging voltage, infers that it is fully charged. The Battery federate instantiates a new EV with a battery at a random initial state of charge. The Battery federate also calculates a charging current of zero amps and publishes it.

4. The Charger federate, seeing a charging current of zero amps, infers a new EV has been set up to charge, randomly assigns one of three charging powers, and publishes this new charging voltage.

At this point the co-simulation proceeds as previously defined. The Battery uses its internal knowledge of the state-of-charge to define the charging current which the Charger uses to estimate the state-of-charge and sends on to the Controller. The Controller sends back a message to the Charger based on this state-of-charge estimate indicating whether the EV should continue to be charged or not.

The HELICS components introduced in the Fundamental Examples are extended in the Advanced Examples with additional discussion of timing and initialization of federates. These new components enter into the sequence as follows:

1. Register and Configure Federates
2. Initialization
3. Enter Execution Mode
4. Define Time Variables
5. Tell Controller federates to request `h.HELICS_TIME_MAXTIME`
6. Initiate Time Steps for the Time Loop
7. Send and Receive Communication between Federates
8. Finalize Co-simulation

Federates which are abstractions of reality (e.g., controllers) do not need regular time interval updates. These types of federates can be set up to request `HELICS_TIME_MAXTIME` (effectively infinite time) and only update when a new message arrives for it to process. This component is placed prior to the main time loop.

```
hours = 24 * 7  # one week
total_interval = int(60 * 60 * hours)
grantedtime = 0
starttime = int(h.HELICS_TIME_MAXTIME)
logger.debug(f"Requesting initial time {starttime}")
grantedtime = h.helicsFederateRequestTime(fed, starttime)
logger.debug(f"Granted time {grantedtime}")
```

As in the *Base Example*, the EV batteries are assumed connected to the chargers at the beginning of the simulation and information exchange is initiated by the Charger federate sending the charging voltage to the Battery federate. In the Advanced Examples, this is a convenient choice as the charging voltage is constant and thus is never a function of the charging current. In a more realistic model, it's easy to imagine that the charger has an algorithm that adjusts the charging voltage based on the charging current to, say, ensure the battery is charged at a particular power level. In that case, **the dependency of the models is circular**; this is common component that needs to be addressed.

If the early time steps of the simulation are not as important (a model warm up period), then ensuring each federate has a default value it will provide when the input is null (and assuming the controller dynamics are not overly aggressive) will allow the models to bootstrap and through several iterations reach a consistent state. If this is not the case then HELICS does have a provision for getting models into a consistent state prior to the start of execution: initialization mode (see an example in Battery.py of the *query example* to see the use of the initialization mode API). This mode allows for this same iteration between models with no simulated time passing. It is the responsibility of the modeler to make sure there is a method to reach and detect convergence of the models and when such conditions are met, enter execution mode as would normally be done. We've put together an *example on iteration* to demonstrate one way of managing convergence.

Using the *Advanced Default Example* as the starting point, the following examples have also been constructed:

- ***Iteration*** - Setting up federates so that they can iterate without advancing simulation time to achieve a more consistent state.
- ***Orchestration Tool (Merlin)*** Demonstration of using Merlin to handle situations where a HELICS co-simulation is just one step in an automated analysis process (*e.g.* uncertainty quantification) or where assistance is needed deploying a large co-simulation in an HPC environment.
- **Multiple Brokers**
  - ***Connecting Multiple Core Types (Multi-Protocol Broker)*** - Demonstration of how to configure a multi-protocol broker
  - ***Broker Hierarchies*** - Purpose of broker hierarchies and how to configure a HELICS co-simulation to implement one.

  - ***Simultaneous co-simulations*** - Demonstration of how to run multiple independent federations simultaneously on a single compute node.
  - ***Multi-Protocol Brokers (Multi-broker) for Multiple Core Types)*** What to do when one type of communication isn't sufficient.
  - ***Multi-compute-node Co-simulation*** - Executing a co-simulation across multiple compute nodes.
- ***Queries*** - Demonstration of the use of queries for dynamic federate configuration.
- ***Multi-Source Inputs*** - Demonstration of use and configuration of a multi-sourced input value interface.
- ***Orchestration Tool (Merlin)*** Demonstration of using Merlin to handle situations where a HELICS co-simulation is just one step in an automated analysis process (*e.g.* uncertainty quantification) or where assistance is needed deploying a large co-simulation in an HPC environment.

2.4.3 Examples in Supported Languages

User Guide Advanced Default Example

The User Guide Examples are primarily written in Python due to its popularity but HELICS provides language bindings for many other languages. To demonstrate this, *one of the examples* has been written in each of the supporting languages to show how the APIs look in each of those languages but achieve similar functionality.

- Python (C-API)
- Python (Pythonic API) - forthcoming
- MATLAB
- C - forthcoming
- C++ - forthcoming
- Julia - forthcoming
- Java - forthcoming
- C# - forthcoming

Other Examples

2.4.4 Miscellaneous Examples

This section provides a collection of miscellaneous examples that we hope will help users

GridLAB-D Example 1

This example is from an old version of the User Guide and shows how GridLAB-D can be used with HELICS to implement functionality that would otherwise not be possible without editing GridLAB-D source code. The example repeats some of the material in the *Fundamentals section of the User Guide* but, by including GridLAB-D, provides a more concrete example of how to use HELICS to answer power system problems.

Example 1a: GridLAB-D as a Value Federate

HELICS messages that are value-oriented are the most common type of messages. As mentioned in the *federate introduction*, value messages are intended to be used to represent the physics of a system, linking federates at their mutual boundaries and allowing a larger and more complex system to be represented than would be the case if only one simulator was used.

Value Federate Interface Types

There are four interface types for value federates that allow the interactions between the federates (a large part of co-simulation/federation configuration) to be flexibly defined. The difference between the four types revolve around whether the interface is for sending or receiving HELICS messages and whether the sender/receiver is defined by the federate (technically, the core associated with the federate):

- **Publications** - Sending interface where the federate core does not specify the intended recipient of the HELICS message
- **Named Inputs** - Receiving interface where the federate core does not specify the source federate of the HELICS message
- **Directed Outputs** - Sending interface where the federate core specifies the recipient of HELICS message
- **Subscriptions** - Receiving interface where the federate core specifies the sender of HELICS message

In all cases the configuration of the federate core declares the existence of the interface to use for communicating with other federates. The difference between "publication"/"named inputs" and "directed outputs"/"subscriptions" is where that federate core itself knows the specific names of the interfaces on the receiving/sending federate core.

The message type used for a given federation configuration is often an expression of the preference of the user setting up the federation. There are a few important differences that may guide which interfaces to use:

- **Which interfaces does the simulator support?** - Though it is the preference of the HELICS development team that all integrated simulators support all four types, that may not be the case. Limitations of the simulator may limit your options as a user of that simulator.
- **Is portability of the federate and its configuration important?** - Because "publications" and "named inputs" don't require the federate to know who it is sending HELICS messages to and receiving HELICS messages from as part of the federate configuration, it affords a slightly higher degree of portability between different federations. The mapping of HELICS messages still needs to be done to configure a federation, its just done separately from the federate configuration file via a broker or core configuration file. The difference in location of this mapping may offer some configuration efficiencies in some circumstances.

Though all four message types are supported, the remainder of this guide will focus on publications and subscriptions as they are conceptually easily understood and can be comprehensively configured through the individual federate configuration files.

Federate Configuration Options via JSON

For any simulator that you didn't write for yourself, the most common way of configuring that simulator for use in a HELICS co-simulation will be through the use of an external JSON configuration file. TOML files are also supported but we will concentrate on JSON for this discussion. This file is read when a federate is being created and initialized and it will provide all the necessary information to incorporate that federate into the co-simulation.

As the fundamental role of the co-simulation platform is to manage the synchronization and data exchange between the federates, you may or may not be surprised to learn that there are generic configuration options available to all HELICS federates that deal precisely with these. In this section, we'll focus on the options related to data exchange as pertaining to value federates, those options and in *Timing section* we'll look at the timing parameters.

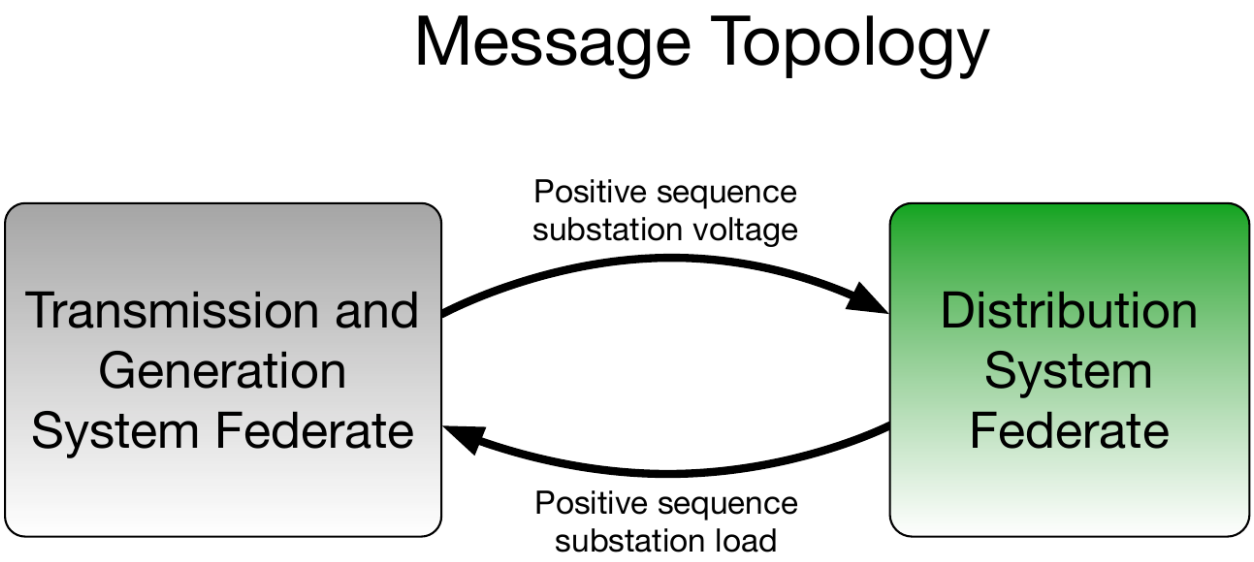
Let's look at a generic JSON configuration file as an example with the more common parameters shown; the default values are shown in "[ ]". (Further parameters and explanations can be found in the *federate configuration* guide.

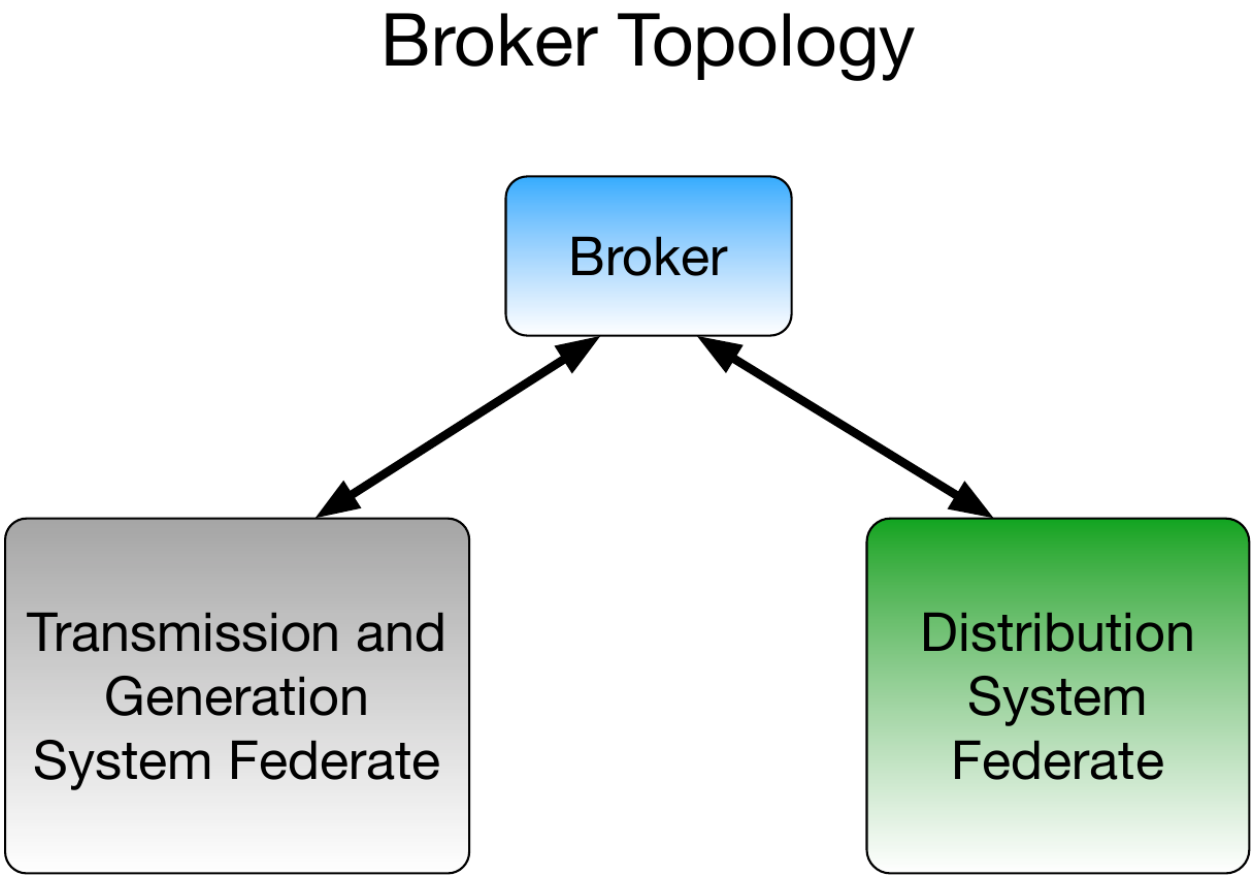
Example 1a - Basic transmission and distribution powerflow

To demonstrate how a to build a co-simulation, an example of a simple integrated transmission system and distribution system powerflow can be built; all the necessary files are found HERE but to use them you'll need to get some specific software installed; here are the instructions:

1. *HELICS*
2. GridLAB-D - Enable HELICS, see instructions here
3. Python - Anaconda installation, if you don't already have Python installed. You may need to also install the following Python packages (`conda install ...`)
    - matplotlib
    - time
    - logging
4. PyPower - `pip install pypower`

This example has a very simple message topology (with only one message being sent by either federate at each time step) and uses only a single broker. Diagrams of the message and broker topology can be found below:





- **Transmission system** - The transmission system model used is the IEEE-118 bus model. To a single bus in this model the GridLAB-D distribution system is attached. All other load buses in the model use a static load shape scaled proportionately so the peak of the load shape matches meet the model-defined load value. The generators are re-dispatched every fifteen minutes by running an optimal power flow (the so-called "ACOPF" which places constraints on the voltage at the nodes in the system) and every five minutes a powerflow is run the update the state of the system. To allow for the relatively modest size of the single distribution system attached to the transmission system, the distribution system load is amplified by a factor of fifteen before being applied to the transmission system.
- **Distribution system** - A GridLAB-D model of the IEEE-123 node distribution system has been used. The model includes voltage regulators along the primary side of the system and includes secondary (or distribution) transformers with loads attached to the secondary of these transformers. The loads themselves are ZIP loads with a high impedance traction that are randomly scaled versions of the same time-varying load-shapes.

In this particular case, the Python script executing the transmission model also creates the broker; this is a choice of convenience and could have been created by any other federates. This simulation is run for 24 hours.

Running co-simulations via `helics run ...`

To run this simulation, the HELICS team has also developed a standardized means of launching co-simulations. Discussion of how to configure a JSON for use in launching a HELICS-based co-simulation is discussed in the *over here* but for all these examples, the configuration has already been done. In this case, that configuration is in the examples folder as "cosim_runner_1a.json" and looks like this:

```
{
  "broker": true,
```

(continues on next page)

(continued from previous page)

```
  "federates": [
    {
      "directory": ".",
      "exec": "python 1abc_Transmission_simulator.py -c 1a",
      "host": "localhost",
      "name": "1a_Transmission"
    },
    {
      "directory": ".",
      "exec": "gridlabd 1a_IEEE_123_feeder.glm",
      "host": "localhost",
      "name": "1a_GridLABD"
    }
  ],
  "name": "1a-T-D-Cosimulation-HELICSRunner"
}
```

Briefly, it's easy to guess what a few of these parameters do:

- "directory" is the location of the model to be run
- "exec" is the command line call (with all necessary options) to launch the co-simulation

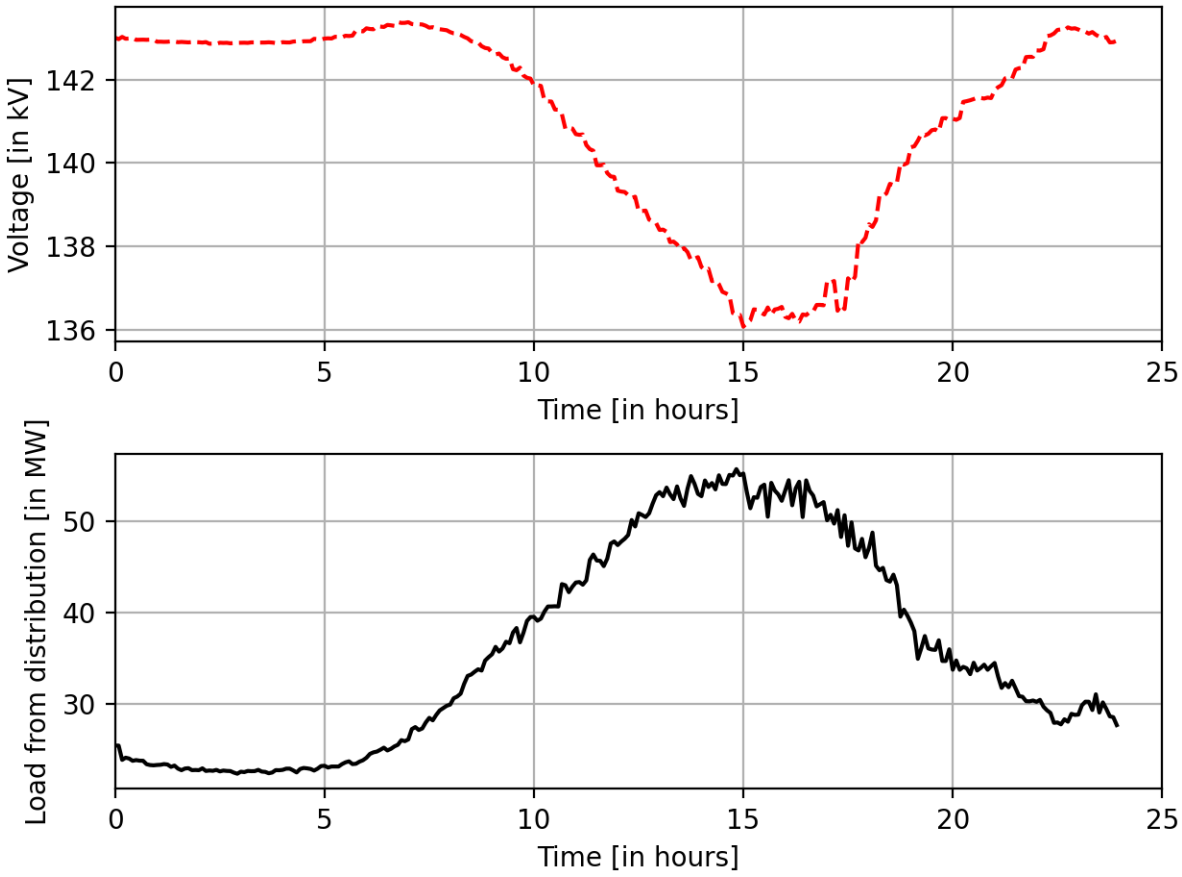
With a properly written configuration file, launching the co-simulation becomes very straightforward:

```
helics run --path=<path to HELICS runner JSON configuration file>
```

Experiment and Results

To show the difference between running these two simulators in a stand-alone analysis and as a co-simulation, modify the federate JSON configurations and use `helics run ...` in both cases to run the analysis. To run the two as a co-simulation, leave publication and subscription entries in the federate JSON configuration. To run them as stand-alone federates with no interaction, delete the publications and subscriptions from both JSON configuration files. By removing the information transfer between the two they become disconnected but are still able to be executed as if they were participating in the federation.

The figure below shows the total load on the transmission node to which the distribution system model is attached and the transmission system voltage at the same node over the course of the simulated day.



As can be seen, the load and voltage are correlated as expected but the correlation is relatively weak; big changes in load have minimal impacts on the transmission voltage. This is what we would expect for a simple transmission and distribution co-simulation.

Example 1b: GridLAB-D as a Message Federates

As previously discussed in the *federate introduction*, message federates are used to create HELICS messages that model information transfers (versus physical values) moving between federates. Measurement and control signals are typical applications for these types of federates.

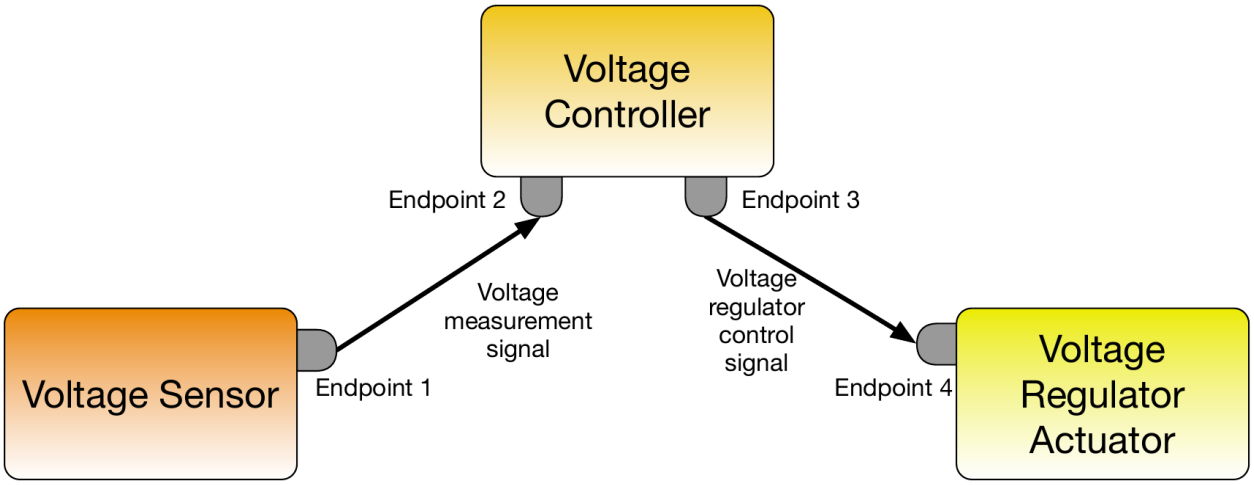
Unlike HELICS values which are persistent (meaning they are continuously available throughout the co-simulation), HELICS messages are only readable once when collected from an endpoint. Once that collection is made the message only exists within the memory of the collecting message federate. If another message federate needs the information, a new message must be created and sent to the appropriate endpoint. Filters can be created to clone messages as well if that behavior is desired.

Message Federate Endpoints

As previously discussed, message federates interact with the federation by defining an "endpoint" that acts as their address to send and receive messages. Message federates are typically sending and receiving measurements, control signals, commands, and other signal data with HELICS acting as a perfect communication system (infinite bandwidth, virtually no latency, guaranteed delivery).

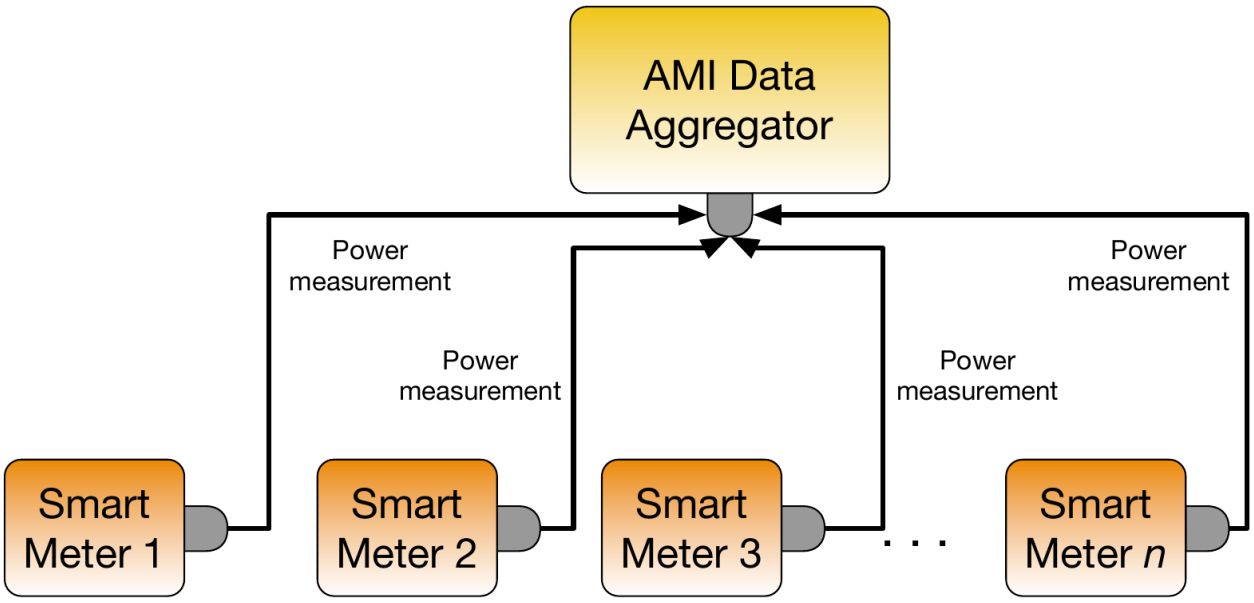
In fact, as you'll see in *a later section*, it is possible to create more realistic communication-system effects natively in HELICS (as well as use a full-blown communication simulator like ns-3 to do the same). This is relevant now, though,

because it influences how the endpoints are created and, as a consequence, how the simulator handles messages. You could, for example, have a system with three federates communicating with each other: a remote voltage sensor, a voltage controller, and a voltage regulation actuator (we'll pretend for the case of this example that the last two are physically separated though they often aren't). In this case, you could imagine that the voltage sensor only sends messages to the voltage controller and the voltage controller only sends messages to the voltage regulation actuator. That is, those two paths between the three entities are distinct, have no interaction, and have unique properties (though they may not be modeled). Given this, referring to the figure below, the voltage sensor could have one endpoint ("Endpoint 1") to send the voltage signal, the voltage regulator could receive the measurement at one endpoint ("Endpoint 2") and send the control signal on another ("Endpoint 3"), and the voltage regulation actuator would receive the control signal on its own endpoint ("Endpoint 4").



The federate code handling these messages can be relatively simple because the data coming in or going out of each endpoint is unique. The voltage controller always receives (and only receives) the voltage measurement at one endpoint and similarly only sends the control signal on another.

Consider a counter-example: automated meter-reading (AMI) using a wireless network that connects all meters in a distribution system to a data-aggregator in the substation (where, presumably, the data travels over a dedicated wired connection to a control room). All meters will have a single endpoint over which they will send their data but what about the receiver? The co-simulation could be designed with the data-aggregator having a unique endpoint for each meter but this implies come kind of dedicated communication channel for each meter; this is not the case with wireless communication. Instead, it is probably better to create a single endpoint representing the wireless connection the data aggregator has with the AMI network. In this case, messages from any of the meters in the system will be flowing through the same endpoint and to differentiate the messages from each other, the federate will have to be written to examine the metadata with the message to determine its original source.



Interactions Between Messages and Values

Though it is not possible to have a HELICS message show up at a value interface, the converse is possible; message_federates can subscribe to HELICS values. Every time a value federate publishes a new value to the federation, if a message federate has subscribed to that message HELICS will generate a new HELICS message and send it directly to the destination endpoint. These messages are queued and not overwritten (unlike in HELICS values) which means when a message federate is synchronized it may have multiple messages from the same source to manage.

This feature offers the convenience of allowing a message federate to receive messages from pure value federates that have no endpoints defined. This is particularly useful for simulators that do not support endpoints but are required to provide measurement signals controllers. Implemented in this way, though, it is not possible to later implement a full-blown communication simulator that these values-turned-messages can traverse. Such co-simulation architectures in HELICS require the existence of both a sending and receiving endpoint; this feature very explicitly bypasses the need for a sending endpoint.

Example 1b - Distribution system EV charge controller

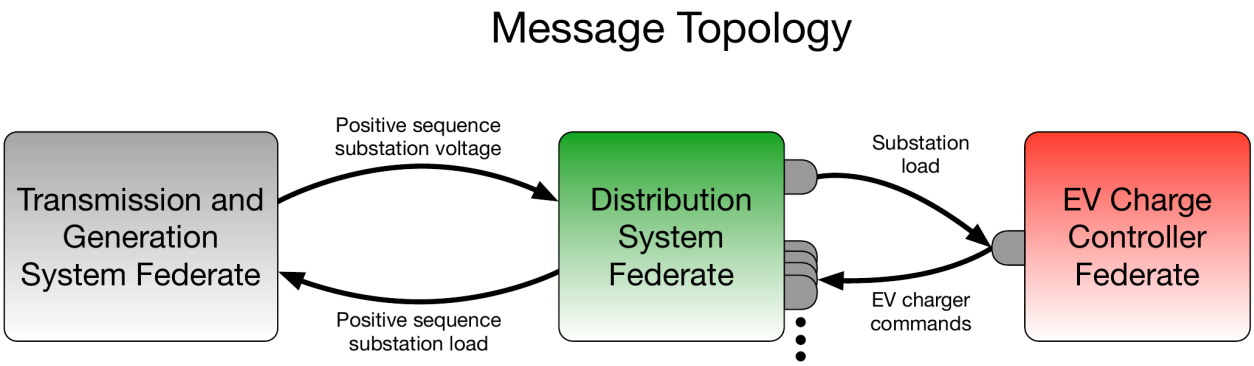
To demonstrate how a message federate interacts with the federation, let's take the previous example and add two things to it: add electric vehicle (EV) loads in the distribution system, and a centralized EV charge control manager. Models files for this example can be found here.

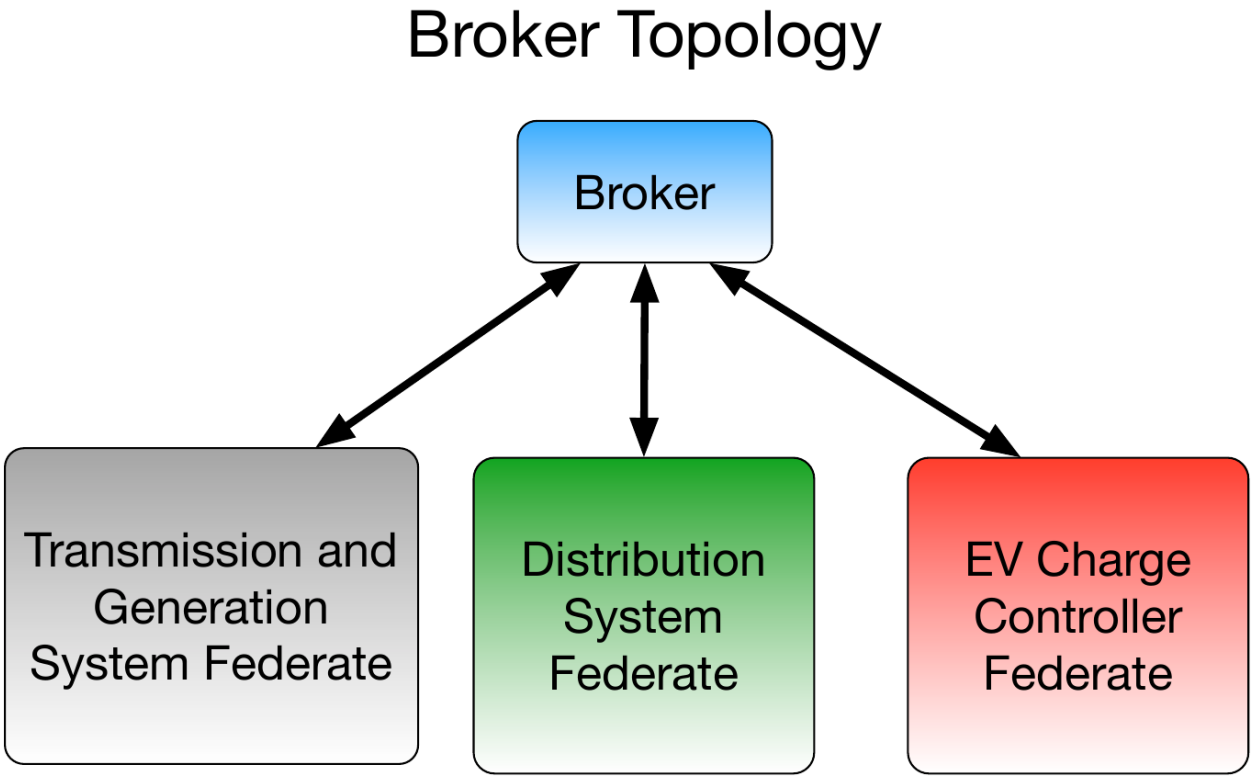
Keeping in mind that this a model for demonstration purposes (which is to say, don't take this too seriously), let's make the following assumptions and definitions to simplify the behavior of the EV charge controller:

- All EVs are very large (200kW; level 2 charging is rated up to 20kW so these are effective HVDC chargers)
- All EVs have infinite battery capacity
- All EVs will be at home all day, desiring to charge all day if they can.
- All EVs charge at the same power level.
- The EV charge controller has direct control over the charging of all EVs in the distribution system. It can tell them when to turn off and on at will.
- The EV charge controller has the responsibility to limit the total load of the distribution system to a specified level to prevent overloading on the substation transformer.

- The EV will turn off some EV charging when the total distribution load exceeds the transformer limit by a certain percentage and will turn some EVs back on when below the limit by a certain percentage.
- Nothing is fair about how the charge controller chooses which EVs to charge and which to disconnect.

The message topology (including the endpoints) and the not very interesting broker topology are shown below.





Taking these assumptions and specifications, it is not too difficult to write a simple charge controller as a Python script. And just by opening the JSON configuration file we can learn important details about how the controller works.

```
{
  "broker": true,
  "federates": [
    {
      "directory": ".",
      "exec": "python 1abc_Transmission_simulator.py -c 1b",
      "host": "localhost",
```

(continues on next page)

(continued from previous page)

```
      "name": "1b_Transmission"
    },
    {
      "directory": ".",
      "exec": "python 1bc_EV_Controller.py -c 1b",
      "host": "localhost",
      "name": "1b_Controller"
    },
    {
      "directory": ".",
      "exec": "gridlabd.sh 1b_IEEE_123_feeder.glm",
      "host": "localhost",
      "name": "1b_GridLABD"
    }
  ],
  "name": "1b-T-D-Cosimulation-HELICSRunner"
}
```

The first thing to note is that the EV controller has been written as a combination federate, having both endpoints for receiving/sending messages and subscriptions to HELICS values. The HELICS values that the controller has subscribed to give the controller access to both the total load of the feeder (`totalLoad`, presumably) as well as the charging power for each of the individual EVs being controlled (six in total).

Looking at the GridLAB-D JSON configuration file confirms this:

```
{
  "coreInit": "--federates=1",
  "coreName": "Distribution Federate",
  "coreType": "zmq",
  "name": "DistributionSim",
  "offset": 0.0,
  "period": 60,
  "timeDelta": 1.0,
  "logfile": "output.log",
  "log_level": "warning",
  "publications": [
    {
      "global": true,
      "key": "IEEE_123_feeder_0/totalLoad",
      "type": "complex",
      "unit": "VA",
      "info": {
        "object": "network_node",
        "property": "distribution_load"
      }
    }
  ],
  "subscriptions": [
    {
      "required": true,
      "key": "TransmissionSim/transmission_voltage",
      "type": "complex",
      "unit": "V",
```

(continues on next page)

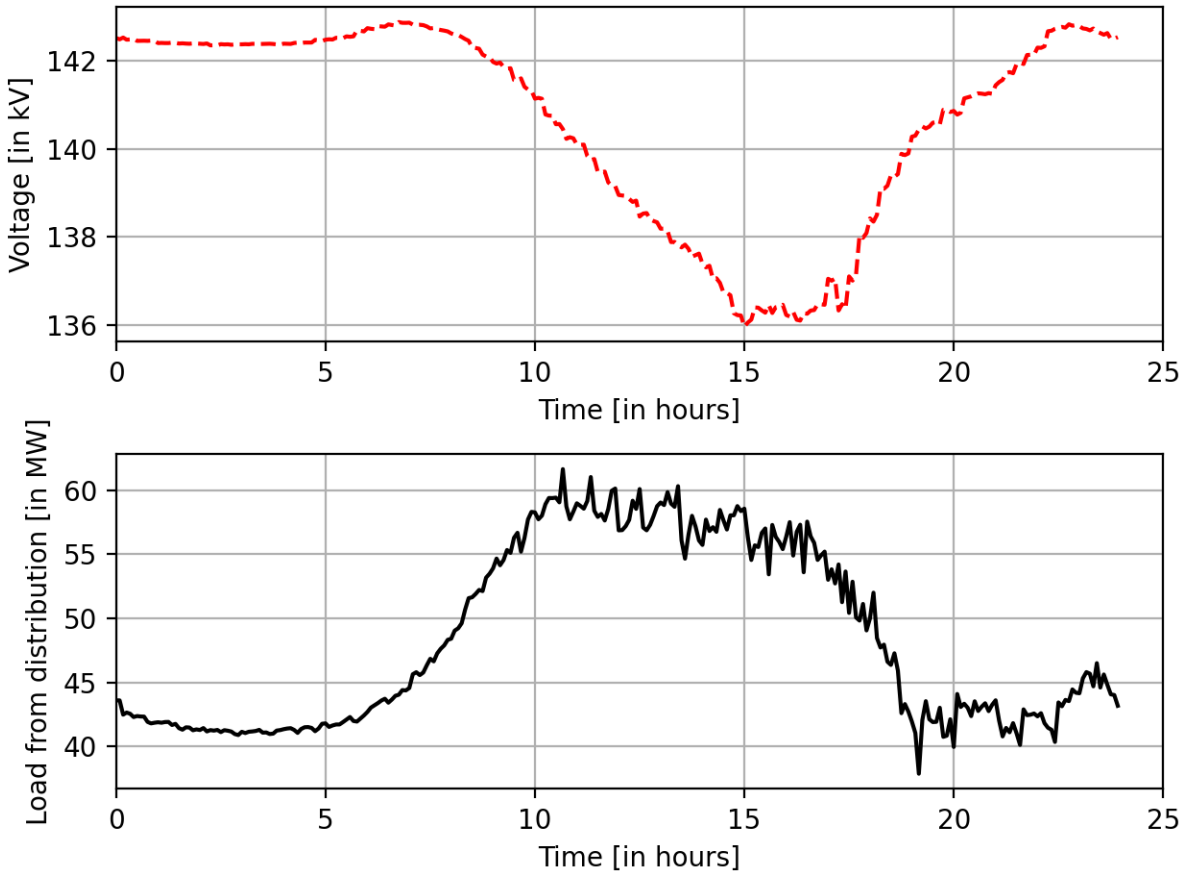
(continued from previous page)

```
      "info": {
        "object": "network_node",
        "property": "positive_sequence_voltage"
      }
    }
  ],
  "endpoints": [
    {
      "global": true,
      "key": "IEEE_123_feeder_0/EV6",
      "destination": "EV_Controller/EV6",
      "info": {
        "publication_info": {
          "object": "EV6",
          "property": "constant_power_A"
        },
        "subscription_info": {
          "object": "EV6",
          "property": "constant_power_A"
        }
      }
    }
  ]
}
```

GridLAB-D is publishing out the total load on the feeder as well as the individual EV charging loads. It also has endpoints set up for each of the EV chargers to receive messages from the controller. Based on the strings in the `info` field it appears that the received messages are used to define the EV charge power.

Running the example and looking at the results, as the total load on the feeder exceeded the pre-defined maximum loading of the feeder (red line in the graph), the EV controller disconnected an additional EV load. Conversely, as the load dipped to the lower limit (green line), the controller reconnected the EV load. Looking at a graph of the EV charge power for each EV shows the timing of the EV charging for each load.

Given the relatively dramatic changes in load, you might expect the voltage on the transmission system to be impacted.



Example 1c: Filters and their Impacts on HELICS Messages

As was introduced in the *introductory section on federates*, message federates (and combo federates) are used to send messages (control signals, measurements, anything traveling over some kind of communication network) via HELICS to other federates. Though they seem very similar, the way messages and values are handled by HELICS is very different and is motivated by the underlying reality they are being used to model.

1. **Messages are directed and unique, values are persistent.** - Because HELICS values are used to represent physical reality, they are available to any subscribing federate at any time. If the publishing federate only updates the value, say, once every minute, any subscribing federates that are granted a time during that minute window will all receive the same value regardless of when they requested it.

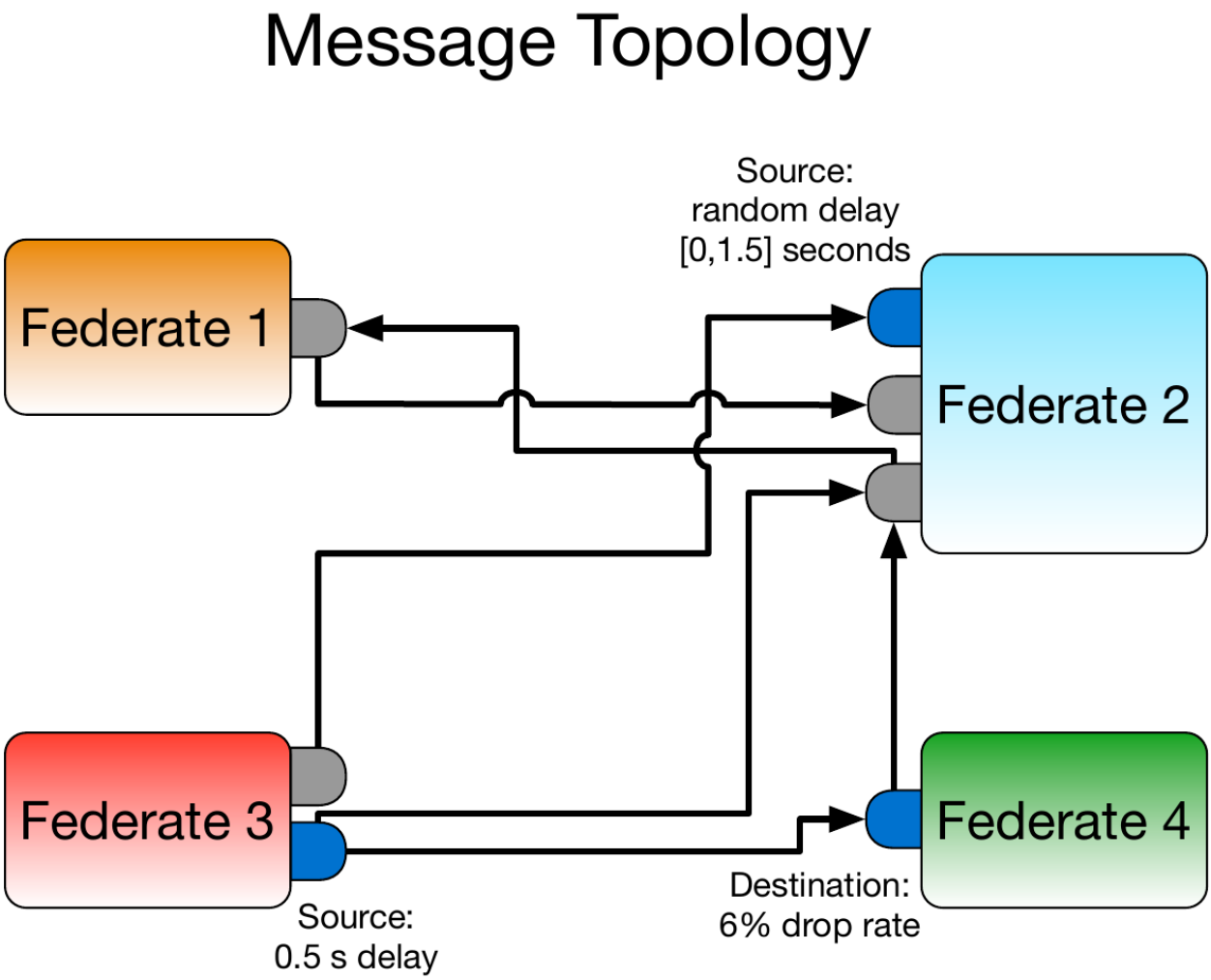
   HELICS messages, though, are much more like other kinds of real-world messages in that they are directed and unique. Messages are sent by a federate from one endpoint to another endpoint in the federation (presumably the receiving endpoint is owned by another federate but this doesn't have to be the case). Internal to HELICS, each message has a unique identifier and can be thought to travel between a generic communication system between the source and destination endpoints.

2. **Messages can be filtered, values cannot.** - By creating a generic communication system between two endpoints, HELICS has the ability to model simple communication system effects on messages traveling through said network. These effects are called "filters" and are associated with the HELICS core (which in turn manages the federate's endpoints) embedded with the federate in question. Typical filtering actions might be delaying the transmission of the message or randomly dropping a certain percentage of the received messages. Filters can also be defined to operate on messages being sent ("source filters") and/or messages being received ("destination filters").

Because HELICS values do not pass through this generic network, they cannot be operated on by filters. Since HELICS values are used to represent physics of the system not the control and coordination of it, it is appropriate that filters not be available to modify them. It is entirely possible to use HELICS value federates to send control signals to other federates; co-simulations can and have been made to work in such ways. Doing so, though, cuts out the possibility of using filters and, as we'll see, the easy integration of communication system simulators.

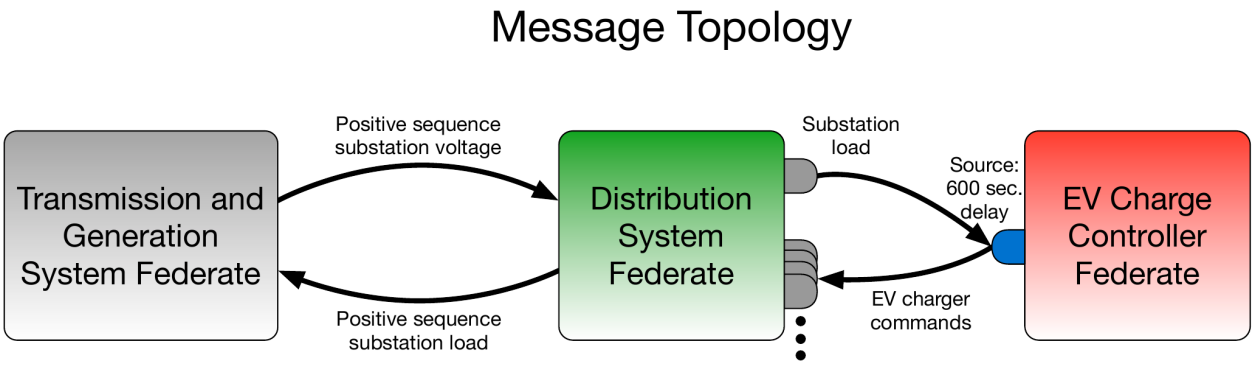
The figure below is an example of a representation of the message topology of a generic co-simulation federation composed entirely of message federates. Source and destination filters have been implemented (indicated by the blue endpoints), each showing a different built-in HELICS filter function.

- As a reminder, a single endpoint can be used to both send and receive messages (as shown by Federate 4). Both a source filter and a destination filter can be set up on a single endpoint. In fact multiple source filters can be used on the same endpoint.
- The source filter on Federate 3 delays the messages to both Federate 2 and Federate 4 by the same 0.5 seconds. Without establishing a separate endpoint devoted to each federate, there is no way to produce different delays in the messages sent along these two paths.
- Because the filter on Federate 4 is a destination filter, the message it receives from Federate 3 is affected by the filter but the message it sends to Federate 2 is not affected.
- As constructed, the source filter on Federate 2 has no impact on this co-simulation as there are no messages sent from that endpoint.
- Individual filters can be targeted to act on multiple endpoints and act as both source and destination filters.

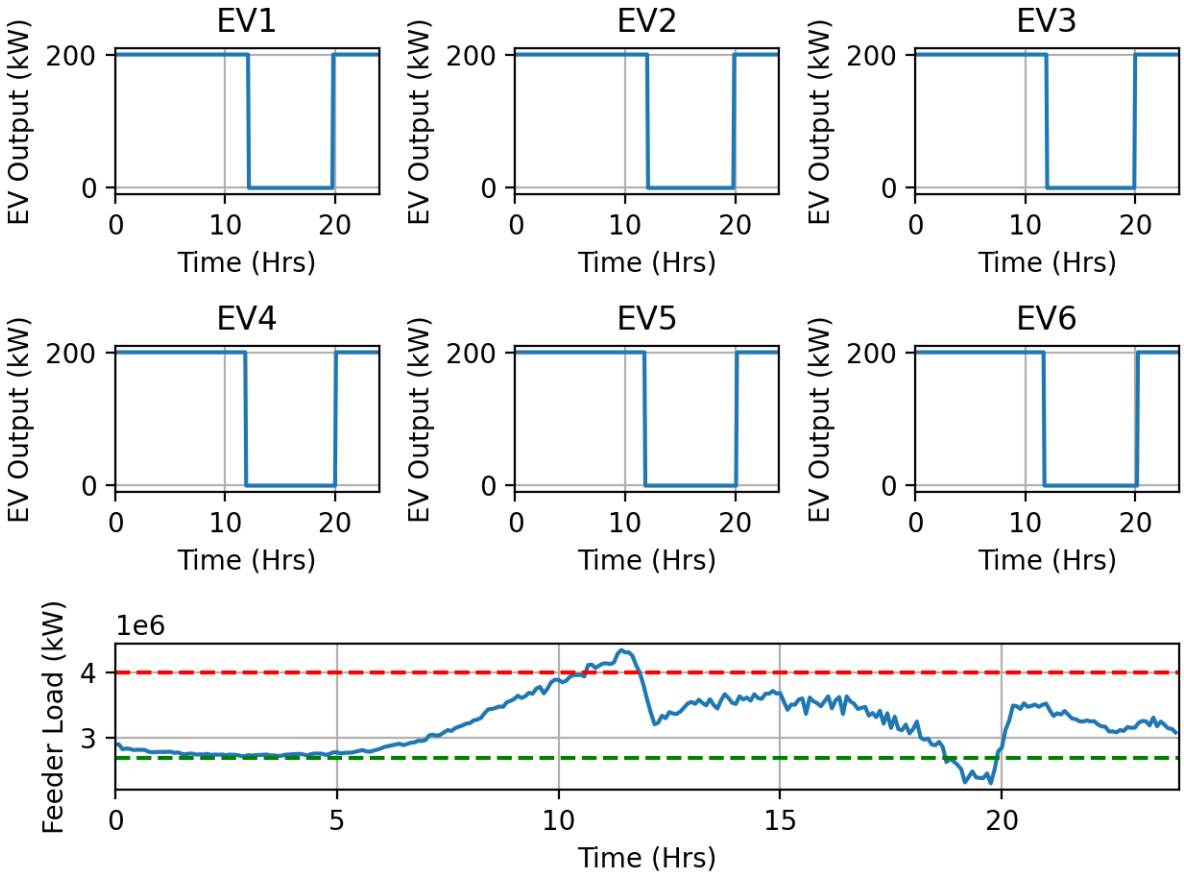


Example 1c - EV charge controller with HELICS filters

To demonstrate the effects of filters, let's take the same model we were working with in the *previous example*, and add a filter to the controller. Specifically, let's assume a very, very poor communication system and add a 3600 second delay to the control messages sent from the EV charge controller to each of the EVs. (Model files for this example can be found here.)



Let's run this co-simulation and capture the same data as last time for direct comparison: total substation load and EV charging behavior, both as a function of time.



Granted that the charge controller communication system is ridiculously poor, this example does show that communication system effects can have a significant impact on system operation. For more realistic example, the HELICS Use Case repository has an example of frequency control using real-time PMU measurements that shows the impact of imperfect communication systems.

2.5 Support

If you're having trouble with understanding what HELICS does, how to use HELICS, getting a specific feature to work, or trouble-shooting some aspect of a HELICS-based co-simulation; there are a variety of support mechanisms we offer.

- **Documentation** - Though you're already here at the documentation site, the following pages may be of particular interest:
  - *Installation Guide* - Help with getting HELICS installed with the correct options for your particular use case
  - *User Guide* - Comprehensive guide to understanding and using HELICS covering both the *fundamentals* as well as more *advanced topics*.
  - *API reference* - HELICS is written in C++ with many supported language bindings. The C++ Doxygen is the most comprehensive reference for the HELICS API
- **User Forum** - A space where users can discuss their HELICS use cases and get support on how to use HELICS.

- **Bug Reports/Support** - If HELICS doesn't appear to be working as expected, this is the place to file a bug report and get some help.
- **Examples, Use Cases, and Tutorials** - Over the years there have been a number of different public examples, use cases, and demonstrations of HELICS. These may be useful to show how particular features work or as a starting point for your use of HELICS.
  - NREL March 2020 In-house Tutorial
  - HELICS Examples Github repository
  - HELICS Fundamental Tutorials
  - Demonstration Use Cases - Fully-formed co-simulation use cases using a variety of simulators and HELICS features.
- **YouTube Channel** - HELICS developers have created a number of short lectures and tutorials about how HELICS works and how to use it.
- **Gitter** - Live chat with developers (when they're logged in) for quick fix support.
- **Virtual Office Hours** - HELICS developers have periodic open office hours where anybody can come and get hands-on help with understanding HELICS better or getting a particular feature to work.

CHAPTER

THREE

DEVELOPER GUIDE

3.1 Style Guide

The goal of the style guide is to describe in detail naming conventions for developing HELICS. Style conventions are encapsulated in the .clang_format files in the project.

We have an EditorConfig file that has basic formatting rules code editors and IDEs can use. See https://editorconfig.org/ for how to setup support in your preferred editor or IDE.

3.1.1 Naming Conventions

1. All functions should be `camelCase`

```
PublicationID registerGlobalPublication (const std::string &name, const std::string␣
↪&type, const std::string &units = "");
```

   EXCEPTION: when the functionality matches a function defined in the standard library e.g. to_string()

2. All classes should be `PascalCase`

```
class ValueFederate : public virtual Federate
{
public:
    ValueFederate (const FederateInfo &fedInfo);
}
```

3. class methods should be `camelCase`

```
Publication &registerGlobalPublication (const std::string &name, const std::string &
↪type, const std::string &units = "");
```

   Exceptions: functions that match standard library functions e.g. to_string()

4. Enumeration names should be `PascalCase`. Enumeration values should be CAPITAL_SNAKE_CASE

```
/* Type definitions */
typedef enum {
    HELICS_OK,
    HELICS_DISCARD,
    HELICS_WARNING,
    HELICS_ERROR,
} HelicsStatus;
```

5. Constants and macros should CAPITAL_SNAKE_CASE

6. Variable names: local variable names should be `camelCase` member variable names should be `mPascalCase` static const members should be `CAPITAL_SNAKE_CASE` function input names should be `camelCase` index variables can be `camelCase` or `ii,jj,kk,mm, nn` or similar if appropriate global variables should `gPascalCase`

7. All C++ functions and types should be contained in the helics namespace with subnamespaces used as appropriate

```
namespace helics
{
    ...
} // namespace helics
```

8. C interface functions should begin with helicsXXXX

```
HelicsBool helicsBrokerIsConnected (HelicsBroker broker);
```

9. C interface function should be of the format helics{Class}{Action} or helics{Action} if no class is appropriate

```
HelicsBool helicsBrokerIsConnected (HelicsBroker broker);

const char *helicsGetVersion ();
```

10. All cmake commands (those defined in cmake itself) should be lower case

```
if as opposed to IF
install vs INSTALL
```

11. Public interface functions should be documented consistent with Doxygen style comments non public ones should be documented as well with doxygen but we are a ways from that goal

```
/** get an identifier for the core
    @param core the core to query
    @return a string with the identifier of the core
*/
HELICS_EXPORT const char *helicsCoreGetIdentifier (HelicsCore core);
```

12. File names should match class names if possible

13. All user-facing options (*e.g.* `log_level`) should be expressed to the user as a single word or phrase using `nocase`, `camelCase`, and `snake_case`. Do not use more than one synonymous term for the same option; that is, do not define a single option that is expressed to the user as both `best_ice_cream_flavor` and `most_popular_ice_cream_flavor`.

Exception: when defining the command line options, the command-line parser already handles underscores and casing so there is no need to define all three cases for that parser.

3.2 Generating SWIG extension

C# A C# interface can be generated using swig and enabling HELICS_BUILD_CSHARP_INTERFACE in the CMake. The support is partial; it builds and can be run but not all the functions are completely usable and it hasn't been fully tested.

Java A JAVA interface can be generated using swig and enabling HELICS_BUILD_JAVA_INTERFACE in the CMake. This interface is tested regularly as part of the CI test system.

3.3 Run tests

TODO: Add something or remove this file

3.4 Generating Documentation

The documentation requires Pandoc to convert from Markdown to RST.

You will need the following Python packages.

```
pip install sphinx
pip install ghp-import
pip install breathe
pip install sphinx_rtd_theme
pip install sphinxcontrib-pandoc-markdown
```

You will also need doxygen.

You can then type `make doxygen html` to create the documentation locally.

If you don't have Pandoc, you can install it using `conda`.

```
conda install pandoc
```

If you are unable to install `pandoc`, you may be able to generate some of the documentation if you install the following.

```
pip install recommonmark
```

3.5 HELICS Benchmarks

The HELICS repository has a few benchmarks that are intended to test various aspects of the code and record performance over time

3.5.1 Baseline benchmarks

These benchmarks run on a single machine using Google Benchmarks and are intended to test various aspects of HELICS over a range of spaces applicable to a single machine.

ActionMessage

Micro-benchmarks to test some operations concerning the serialization of the underlying message structure in HELICS

Conversion

Micro-benchmarks to test the serialization and deserialization of common data types in HELICS

3.5.2 Simulation Benchmarks

Echo

A set of federates representing a hub and spoke model of communication for value based interfaces

Echo_c

A set of federates representing a hub and spoke model of communication for value based interfaces using the C shared library.

Echo Message

A set of federates representing a hub and spoke model of communication for message based interfaces

Filter

A variant of the Echo message test that add filters to the messages

Ring Benchmark

A ring like structure that passes a value token around a bunch of times

Ring Message Benchmark

A ring like structure that passes a message token around a bunch of times.

Timing Benchmark

Similar to echo but doesn't actually send any data just pure test of the timing messages

3.5.3 Message Benchmarks

Benchmarks testing various aspects of the messaging structure in HELICS

MessageLookup

Benchmarks sends messages to random federates, varying the total number of interfaces and federates.

MessageSend

Sending messages between 2 federates varying the message size and count per timing loop.

3.5.4 Standardized Tests

PHold

A standard PHOLD benchmark varying the number of federates.

3.5.5 Multinode Benchmarks

Some of the benchmarks above have multinode variants. These benchmarks will have a standalone binary for the federate used in the benchmark that can be run on each node. Any multinode benchmark run will require some setup to make it launch in your particular environment and knowing the basics for the job scheduler on your cluster will be very helpful.

Any sbatch files for multinode benchmark runs in the repository are setup for running in the pdebug queue on LC's Quartz cluster. They are unlikely to work as is on other clusters, however they should work as a starting point for other clusters using slurm. The minimum changes required are likely to involve setting the queue/partition correctly and ensuring the right bank/account for charging CPU time is used.

3.6 Continuous integration builds

The CI configuration is defined in the repository. This page summarizes the jobs currently enabled; the workflow and pipeline files are the authoritative source when a job matrix changes.

3.6.1 Azure Pipelines

Pushes (except `pre-commit/*`) and pull requests targeting `main` run the primary build-and-test matrix:

- Linux containers: Ubuntu 24.04 default, GCC 11, Clang 15, and Clang 18. The Ubuntu default and GCC 11 jobs enable MPI and encryption.
- macOS 14 and macOS 15.
- Windows: Visual Studio 2022 32-bit and 64-bit, plus Visual Studio 2026 64-bit.

The daily pipeline runs the full Linux test suite, a Linux build using ZeroMQ as a subproject, and a 64-bit Visual Studio 2022 build. Windows jobs use Boost 1.83.

3.6.2 CircleCI

For pull requests and branches other than `pre-commit/*`, CircleCI runs install/package checks plus GCC 14, Clang 18, and Clang 18 with C++23. Scheduled jobs on `main` also run OpenSUSE Tumbleweed, a no-ZeroMQ configuration, and an ARM64 build. Benchmarks run three times weekly.

3.6.3 GitHub Actions

GitHub Actions runs the following repository workflows:

- Static analysis (`cpplint` on pushes and pull requests to `main`, and `clang-tidy` on pull requests).
- A Clang 22 C++26 build and smoke-test job on pushes and pull requests to `main`.
- MSYS2 builds using both MinGW and MSYS Makefiles on pull requests to `main`, relevant branch pushes, and a daily schedule.
- Code coverage on `main` pushes and scheduled runs; pull-request coverage runs for branches whose names begin with `coverage_`.
- Address, memory, and thread sanitizer builds on the daily schedule.
- Scheduled benchmark and release-artifact builds, plus release packaging when a GitHub release is published.
- Docker image builds for `main`, Docker-focused branches, tags, and releases.
- CodeQL, generated SWIG-interface updates, image compression, and release-checklist automation.

3.6.4 AppVeyor and Cirrus CI

- AppVeyor performs a Cygwin 64-bit Release compilation on `main`; tests are disabled for this job.

3.6.5 Documentation

Read the Docs builds and hosts the published documentation. Codacy checks may also be reported, but they are informational and are not part of the required CI matrix.

3.7 CI/CD Infrastructure

This page describes the services that support HELICS development and releases. The active build matrix is documented in *Continuous Integration*; repository workflow files remain the source of truth for job definitions and triggers.

3.7.1 Continuous Integration

HELICS uses Azure Pipelines for its primary Linux, macOS, and Windows build matrix. CircleCI adds compiler, install/package, ARM64, and scheduled Linux jobs. GitHub Actions runs static analysis, MSYS2, C++26, coverage, sanitizer, benchmark, release, Docker, and interface-generation workflows. AppVeyor provides the Cygwin build, and Cirrus CI provides the FreeBSD build.

The Linux jobs that use container images are defined by those images and the repository scripts. Windows and macOS jobs use their hosted runner images and install their dependencies in the pipeline. Scheduled jobs run from `main`; pull-request and push triggers are defined individually by each pipeline or workflow.

3.7.2 Documentation and automation

Read the Docs builds and hosts the documentation. GitHub Actions also automates generation of SWIG interface files, release checklists, Docker images, benchmark artifacts, and release artifacts. A published GitHub release triggers the release-artifact workflow and downstream version-update notification.

3.7.3 Additional checks

Codacy and pre-commit checks may report formatting, spelling, and static-analysis findings. They are useful maintenance signals but are not part of the required CI build matrix.

3.8 Porting Guide: HELICS 2 to 3

Since HELICS 3 is a major version update, there are some breaking changes to the API for developers. This guide will try to track what breaking changes are made to the API, and what developers should use instead when updating from HELICS 2.x to 3.

3.8.1 Dependency changes

Support for some older compilers and dependencies have been removed. The new minimum version are:

- C++17 compatible-compiler (minimums: GCC 7.0, Clang 5.0, MSVC 2017 15.7, XCode 10, ICC 19)
- CMake 3.10+ (if using clang with libc++, use 3.18+)
- ZeroMQ 4.2+
- Boost 1.65.1+ (if building with Boost enabled)

3.8.2 Code changes

Changes that will require changing code are listed below based on the interface API used. A list of known PRs that made breaking changes is also provided.

PRs with breaking changes

- #1363
- #1952
- #1907

- #1856
- #1731
- #1727
- #1680
- #1679
- #1677
- #1572
- #1580

Application API (C++17)

- `Federate::error(int errorcode)` and `Federate::error(int errorcode, const std::string& message)` were removed, use `localError` instead (or `globalError` to stop the entire simulation). Changed in #1363.
- `ValueFederate::publishString` and `ValueFederate::publishDouble` have been removed, please use the Publication interface `publish` methods which take a wide variety of types
- The interface object headers are now included by default when including the corresponding federate

Command line interfaces

The numerical value corresponding with the log levels have changed. As such entering numerical values for log levels is no longer supported (it will be again someday). For now please use the text values "none(-1)", "no_print(-1)", "error(0)", "warning(1)", "summary(2)", "connections(3)", "interfaces(4)", "timing(5)", "data(6)", "debug(6)", "trace(7)". The previous values are shown in parenthesis. The new numerical values are subject to revision in a later release so are not considered stable at the moment and are not currently accepted as valid values for command line or config files.

C Shared API

- Only 1 header is now used `#include <helics/helics.h>` for all uses of the C shared library in C/C++ code – no other headers are needed, the other headers are no longer available. #1727
- Removed `helics_message` struct – call functions to set fields instead. `helicsEndpointGetMessage` and `helicsFederateGetMessage` returning this struct were removed – call functions to get field values instead. Changed in #1363.
- `helics_message_object` typedef was renamed to `HelicsMessage` in `api-data.h`; in `MessageFederate.h` and `helicsCallbacks.h` all `helics_message_object` arguments and return types are now `HelicsMessage`. Changed in #1363.
- Renamed `helicsEndpointSendMessageObject` to `helicsEndpointSendMessage`, `helicsSendMessageObjectZeroCopy` to `helicsSendMessageZeroCopy`, `helicsEndpointGetMessageObject` to `helicsEndpointGetMessage`, `helicsEndpointCreateMessageObject` to `helicsEndpointCreateMessage`, `helicsFederateGetMessageObject` to `helicsFederateGetMessage`, and `helicsFederateCreateMessageObject` to `helicsFederateCreateMessage`. Changed in #1363.
- The send data API has changed to make the usage clearer and reflect the addition of targeted endpoints. New methods are `helicsEndpointSendBytes`, `helicsEndpointSendBytesTo`, `helicsEndpointSendBytesAt`, `helicsEndpointSendBytesToAt`. This reflects usage to send a raw byte packet to the targeted destination or a user specified one, and at the current granted time or a user specified time in the simulation future. The C++98 API was changed accordingly. The order of fields has also changed for consistency #1677

- Removed `helicsEndpointClearMessages` – it did nothing, `helicsFederateClearMessages` or `helicsMessageFree` should be used instead. Changed in #1363.
- All constants such as flags and properties are now CAPITAL_SNAKE_CASE #1731
- All structures are now CamelCase, though the old form will be available in helics3 though will be deprecated at some point. #1731 #1580
- `helicsPublicationGetKey` renamed to `helicsPublicationGetName` #1856
- Recommended to change `helicsFederateFinalize` to `helicsFederateDisconnect`, the finalize method is still in place but will be deprecated in a future release. #1952.
- `helicsMessageGetFlag` renamed to `helicsMessageGetFlagOption` for better symmetry #1680
- `helics<*>PendingMessages` moved `helics<*>PendingMessageCount` #1679

C++98 API (wrapper around the C Shared API)

- Removed the `helics_message` struct, and renamed `helics_message_object` to `HelicsMessage`. Direct setting of struct fields should be done through API functions instead. This affects a few functions in the `Message` class in `Endpoint.hpp`; the explicit constructor and `release()` methods now take `HelicsMessage` arguments, and `operator helics_message_object()` becomes `operator HelicsMessage()`. Changed in #1363.

Queries

- Queries now return valid JSON except for `global_value` queries. Any code parsing the query return value will need to be adjusted. Error codes are reported back as HTML error codes and a message.

Libraries

- The C based shared library is now `libhelics.dll` or the appropriate extension based on the OS #1727 #1572
- The C++ shared library is now `libhelicscpp.[dll\so\dylib]` #1727 #1572
- The apps library is now `libhelicscpp-apps.[dll\so\dylib]` #1727 #1572

CMake

- All HELICS CMake variables now start with `HELICS_` #1907
- Projects using HELICS as a subproject or linking with the CMake targets should use `HELICS::helics` for the C API, `HELICS::helicscpp` for the C++ shared library, `HELICS::helicscpp98` for the C++98 API, and `HELICS::helicscpp-apps` for the apps library. If you are linking with the static libraries you should know enough to be able to figure out what to do, otherwise it is not recommended.

3.9 Public API

This file defines what is included in what is considered the stable user API for HELICS.

This API will be backwards code compatible through major version numbers, though functions may be marked deprecated between minor version numbers. Functions in any other header will not be considered in versioning decisions. If other headers become commonly used we will take that into consideration at a later time. Anything marked private is subject to change and most things marked protected can change as well though somewhat more consideration will be given in versioning.

The public API includes the following

- Application API headers
  - CombinationFederate.hpp

  - Publications.hpp
  - Subscriptions.hpp
  - Endpoints.hpp
  - Filters.hpp
  - Translators.hpp
  - Federate.hpp
  - helicsTypes.hpp
  - data_view.hpp
  - MessageFederate.hpp
  - MessageOperators.hpp
  - ValueConverter.hpp
  - ValueFederate.hpp
  - HelicsPrimaryTypes.hpp
  - CallbackFederate.hpp
  - queryFunctions.hpp
  - FederateInfo.hpp
  - Inputs.hpp
  - BrokerApp.hpp
  - CoreApp.hpp
  - timeOperations.hpp
  - typeOperations.hpp
  - Exceptions: Translators and the global time coordinator option are in Beta and subject to finalization in a later release (they are mostly final and any changes will be highlighted). Vector subscriptions, and vector inputs are subject to change. The queries to retrieve JSON may update the format of the returned JSON in the future. A general note on queries: the data returned via queries is subject to change, though in general queries will not be removed. As determined by the need of HELICS users and applications, the data structure may change at minor revision numbers. We hope to fully document the queries structure at which point they will be stable for at least minor releases and changes will be noted. The callback federate API is considered in Beta and may change.

- Core library headers
  - Core.hpp
  - Broker.hpp
  - core-exceptions.hpp
  - core-data.hpp
  - CoreFederateInfo.hpp
  - helicsVersion.hpp
  - federate_id.hpp
  - helics_definitions.hpp

  - NOTE: core headers in the public API are headers that need to be available for the Application API public headers. The core API can be used more directly with static linking but applications are generally recommended to use the application API or other higher level API's

- C shared library headers
  - All C library operations are merged into a single header `helics.h`
  - A `helics_api.h` header is available for generating interfaces which strips out import declarations and comments. The C shared library API is the primary driver of versioning and changes to that will be considered in all versioning decisions.
- App Library
  - Player.hpp
  - Recorder.hpp
  - Echo.hpp
  - Source.hpp
  - Tracer.hpp
  - Probe.hpp
  - Clone.hpp
  - Connector.hpp (considered in Beta and subject to change)
  - helicsApp.hpp
  - BrokerApp.hpp (aliased to application_api version)
  - CoreApp.hpp (aliased to application_api version)
- Exceptions: The vector subscription Objects, and vector Input objects are subject to change.
- C++98 Library *All headers are mostly stable. Though we reserve the ability to make changes to make them better match the main C++ API.*

In the installed folder are some additional headers from third party libraries (CLI11, utilities), we will try to make sure these are compatible in the features used in the HELICS API, though changes in other aspects of those libraries will not be considered in HELICS versioning, this caveat includes anything in the `helics/external` and `helics/utilities` directories. Only changes which impact the signatures defined above will factor into versioning decisions. You are free to use them but they are not guaranteed to be backwards compatible on version changes.

3.10 RoadMap

This document contains tentative plans for changes and improvements of note in upcoming versions of the HELICS library. All dates are approximate and subject to change, but this is a snapshot of the current planning thoughts. See the projects for additional details

3.10.1 [3.8] ~ Fall/Winter 2026

- Update IPC core
- Remove Webserver from C++, migrate it to pyHELICS with additional capabilities

3.10.2 Further in the future

- Updated MPI core
- Some sort of rollback operations
- Remote procedure call type of federate
- Plugin architecture for user defined cores
- Enable mesh networking in HELICS
- Full xSDK compatibility
- Separate Java Interface
- Observer App
- Tag based subscriptions
- Single thread cores

3.11 HELICS Type Conversions

HELICS has the ability to convert data between different types. In the C api methods are available to send and receive data as strings, integers, doubles, boolean, times, char, complex, vector of doubles, and named points.

3.11.1 Available types

The specification of publication allows setting the publication to one of following enum values:

```
    HELICS_DATA_TYPE_UNKNOWN = -1,
    /** a sequence of characters*/
    HELICS_DATA_TYPE_STRING = 0,
    /** a double precision floating point number*/
    HELICS_DATA_TYPE_DOUBLE = 1,
    /** a 64 bit integer*/
    HELICS_DATA_TYPE_INT = 2,
    /** a pair of doubles representing a complex number*/
    HELICS_DATA_TYPE_COMPLEX = 3,
    /** an array of doubles*/
    HELICS_DATA_TYPE_VECTOR = 4,
    /** a complex vector object*/
    HELICS_DATA_TYPE_COMPLEX_VECTOR = 5,
    /** a named point consisting of a string and a double*/
    HELICS_DATA_TYPE_NAMED_POINT = 6,
    /** a boolean data type*/
    HELICS_DATA_TYPE_BOOLEAN = 7,
    /** time data type*/
    HELICS_DATA_TYPE_TIME = 8,
    /** raw data type*/
    HELICS_DATA_TYPE_RAW = 25,
    /** type converts to a valid json string*/
    HELICS_DATA_TYPE_JSON = 30,
    /** the data type can change*/
    HELICS_DATA_TYPE_MULTI = 33,
    /** open type that can be anything*/
    HELICS_DATA_TYPE_ANY = 25262
```

When this data is received it can be received as any of the available types, thus there are no restrictions on which function is used based on the data that was sent. That being said not all conversions are lossless. The following image shows which conversions are lossless through round trip operations. On the sending side the same is also true in that any of the setValue methods may be used regardless of what the actual transmission data type is set to.

For the remaining conversions the loss is typically numerical or comes with conditions. For example converting a vector to a complex number you can think of a complex number as a two element vector. Thus for 1 or 2 element vectors the conversion will be lossless. For 3 or more element vectors the remaining elements are discarded. For conversion of any vector to a single value, double, or integral the value is collapsed using vector normalization. Thus for single element vectors or real valued complex numbers the result is equivalent, whereas for others there is a reduction in information, which may be desired in some cases. Bool values get reduced to `false` if the numerical value is anything but `0`, and true otherwise.

Conversion from strings to numeric values assume the string encodes a number, otherwise it results in an invalid value.

Invalid Values

Invalid values are returned if there is no value or the conversion is invalid for some reason, typically string to numeric value failures, in most cases an empty value equivalent is returned.

- INT -> -9,223,372,036,854,775,808 (`numeric_limits<std::int64_t>::min()`)
- DOUBLE -> `-1e49`
- COMPLEX -> {-1e49,0.0}
- VECTOR -> {}
- COMPLEX_VECTOR -> {}
- STRING -> “”
- NAMED_POINT -> {“”, NaN}
- BOOL -> false
- TIME -> Time::minVal() = `Time(numeric_limits<std::int64_t>::min())`

3.11.2 How types are used in HELICS

Interface specification

For Value based interfaces types are used in a number of locations. For publications the most important part is specifying the type of the publication. This determines the type of data that gets transmitted. It can be a specific known type, a custom type string that is user defined, or an `any` type.

Inputs or subscriptions may also optionally specify a type as well. The use of this is for information to other federates, and for some type compatibility checking. It defaults to an “any” type so if nothing is specified it will match to any publication type.

Data publication and extraction

As mentioned before when using any of the `publication` methods the extraction method call is not limited based on the type given in the interface specification. What the publication type does is define a conversion if necessary. The same is true on the output.

SetValueType -> PublicationType -> GetValueType

Thus there are always two conversions occurring in the data translation pipeline. The first from the setValue to the type specified in the publication registration. The second is from the publish transmission type to the value requested in a respective getValue method.

3.11.3 Data Representation

`int` and `double` are base level types in C++. `complex` is two doubles, `vector` and `complex_vector` are variable length arrays of doubles. `named_point` is `string` and double. String is a variable length sequence of 8 bit values. `char` is a string of length 1, `boolean` is a single char either `0` or `1`. Time is a representation of internal HELICS time as a 64 bit integer (most commonly number of nanoseconds). The JSON type is a string compatible with JSON with two fields "type" and "value". The custom type is a variable length sequence of bytes which HELICS will not attempt to convert, it transmits a sequence of bytes and it would be up to the user to define any conversions or endianness concerns.

3.11.4 Data conversions

There are defined conversions from all known available types to all others.

Conversion from Double

- INT -> trunc(val)
- DOUBLE -> val
- COMPLEX -> val+0j
- VECTOR -> [val]
- COMPLEX_VECTOR -> [val+0j]
- NAMED_POINT -> {"value", val}
- STRING -> string representation such that all required bits are included
- BOOL -> (val!=0)?"1":"0"

Conversion from INT

- INT -> val
- DOUBLE -> val[1]
- COMPLEX -> val+0j
- VECTOR -> [val]
- COMPLEX_VECTOR -> [val+0j]
- NAMED_POINT -> {"value", val}[2]
- STRING -> std::to_string(val)
- BOOL -> (val!=0)?"1":"0"

Conversion from String

- INT -> `getIntFromString(val)`
- DOUBLE -> `getDoubleFromString(val)`
- COMPLEX -> `getComplexFromString(val)`
- VECTOR -> `helicsGetVector(val)`
- COMPLEX_VECTOR -> `helicsGetComplexVector(val)`

[1] conversion to double is lossless only if the value actually fits in a double mantissa value.

[2] for a named point conversion, if the value doesn't fit in double the string translation is placed in the string field and a NaN value in the value segment to ensure no data loss.

- NAMED_POINT -> {val, NaN}
- STRING -> val
- BOOL -> (helicsBoolValue(val))?"1":"0"

helicsGetComplexVector

This method will read a vector of numbers from a string in either JSON format, or `c[X1,X2,...XN]` where XN is a complex number of the format R+Ij. It can also interpret `v[X1,X2, ..., XN]` in which case the values are assumed to be alternating real and imaginary values. If the string is a single value either real or complex it is placed in a vector of length 1.

helicsGetVector

This function is similar to `helicsGetComplexVector` with distinction that string like `v[X1,X2, ..., XN]` are all assumed separate values, and complex vectors are generated as alternating real and imaginary values.

getIntFromString

Converts a string into 64 bit integer. If the string has properties of a double or vector it will truncate the values from `getDoubleFromString`.

getDoubleFromString

Converts a string into a double value, a complex, or a vector of real or complex numbers. If the vector is more than a single element the output is the vector norm of the vector. If the string is not convertible the invalid_double is returned (-1e49).

helicsGetComplexFromString

Similar to getDoubleFromString in conversion of vectors. It will convert most representations of complex number patterns using a trailing `i` or `j` for the imaginary component and assumes the imaginary component is last. The real component can be omitted if not present, for example `4.7+2.7j` or `99.453i`

helicsBoolValue

"0", "00","0000", "\0","false","f","F","FALSE","N","no","n","OFF","off","","disable","disabled" all return false, everything else returns true.

Conversion from vector double

- INT -> trunc(vectorNorm(val))[3]
- DOUBLE -> vectorNorm(val)
- COMPLEX -> val[0]+val[1]j
- VECTOR -> val
- COMPLEX_VECTOR -> [val[0],val[1],...,val[N]]
- STRING -> vectorString(val)[4]
- NAMED_POINT -> {vectorString(val), NaN}[5]

[3] vectorNorm is the sqrt of the inner product of the vector.

[4] vectorString is comma-separated string of the numerical values enclosed in `[]`, for example `[45.7,22.7,17.8]`. This is a JSON compatible string format.

[5] if the vector is a single element the NAMED_POINT translation is equivalent to a double translation.

- BOOL -> (vectorNorm(val)!=0)?"1":"0"

Conversion from Complex Vector

- INT -> trunc(vectorNorm(val))
- DOUBLE -> vectorNorm(val)
- COMPLEX -> val[0]
- VECTOR -> [abs(val[0]),abs(val[1]),...abs(val[N])][6]
- COMPLEX_VECTOR -> val
- NAMED_POINT -> {vectorString(val), NaN}
- STRING -> complexVectorString(val)
- BOOL -> (vectorNorm(val)!=0)?"1":"0"

See *Conversion from vector double* for definitions of vectorNorm.

Conversion from Complex

- INT -> trunc((val.imag() == 0)?val.real(): std::abs(val))
- DOUBLE -> val.imag() == 0)?val.real(): std::abs(val)
- COMPLEX -> val
- VECTOR -> [val.real,val.imag]
- COMPLEX_VECTOR -> [val]
- NAMED_POINT -> {helicsComplexString(val), NaN}
- STRING -> helicsComplexString(val)
- BOOL -> (std::abs(val)!=0)?"1":"0"

If the imaginary value == 0, the value is treated the same as a double.

Conversion from a Named Point

If the value of the named point is `NaN` then treat the name part the same as a string, otherwise use the numerical value as a double and convert appropriately. The exception is a string which has a dedicated operation to generate a JSON string with two fields {"name" and "value"}.

Conversion from Bool

- INT -> (val)?1:0
- DOUBLE -> (val)?1.0:0.0
- COMPLEX -> (val)?1.0:0.0 + 0.0j
- VECTOR -> [(val)?1.0:0.0]
- COMPLEX_VECTOR -> [(val)?1.0:0.0 +0.0j]
- NAMED_POINT -> {"value",(val)?1.0:0.0}
- STRING -> val?"1":"0";
- BOOL -> val?"1":"0";

[6] if the imaginary part of these values is 0, then the real part is used; otherwise it uses the absolute value.

Conversion from Time

Time is transmitted as a 64 bit integer so conversion rules of an integer apply with the note that a double as a time is assumed as seconds and the integer represents nanoseconds so a double (as long as not too big) will be transmitted without loss as long as the precision is 9 decimal digits or less.

3.11.5 Unit conversions

HELICS also handles unit conversions if units are specified on the publication and subscription and can be understood by the units library. This applies primarily for pub/sub of numerical types. HELICS uses Units as the units library.

CHAPTER

FOUR

REFERENCES

This section is most useful for those that are acting as integrators, getting a new simulation tool or custom code working as a HELICS federate. It provides references to the existing functionality HELICS provides as well as links to tools that are currently integrated with HELICS, both external simulation tools and internal support tools.

4.1 HELICS Publications

4.1.1 Citing HELICS

T. Hardy, B. Palmintier, P. Top, D. Krishnamurthy and J. Fuller, "HELICS: A Co-Simulation Framework for Scalable Multi-Domain Modeling and Analysis," in IEEE Access, doi: 10.1109/ACCESS.2024.3363615, available at https://ieeexplore.ieee.org/document/10424422

4.1.2 Co-Simulation Architecture and Theory with HELICS

B. Palmintier, D. Krishnamurthy, P. Top, S. Smith, J. Daily and J. Fuller, "Design of the HELICS high-performance transmission-distribution-communication-market co-simulation framework", Proc. Workshop Model. Simul. Cyber-Physical Energy Syst. (MSCPES), pp. 1-6, Apr. 2017, available at: https://ieeexplore.ieee.org/document/8064542/.

Juan Gilberto Rueda Vasquez. COMPARISON BETWEEN TWO CO-SIMULATION FRAMEWORKS, MOSAIK AND HELICS. Rel. Lorenzo Bottaccioli, Edoardo Patti, Claudia De Vizia. Politecnico di Torino, Corso di laurea magistrale in Ict For Smart Societies (Ict Per La Società Del Futuro), 2022, available at https://webthesis.biblio.polito.it/25624/

P. Palensky, P. Mancarella, T. Hardy and M. Cvetkovic, "Cosimulating Integrated Energy Systems With Heterogeneous Digital Twins: Matching a Connected World," in IEEE Power and Energy Magazine, vol. 22, no. 1, pp. 52-60, Jan.-Feb. 2024, doi: 10.1109/MPE.2023.3324886, available at https://ieeexplore.ieee.org/document/10398554.

Barbierato, L., Rando Mazzarino, P., Montarolo, M. et al. A comparison study of co-simulation frameworks for multi-energy systems: the scalability problem. Energy Inform 5 (Suppl 4), 53 (2022). doi: 10.1186/s42162-022-00231-6, available at https://energyinformatics.springeropen.com/articles/10.1186/s42162-022-00231-6.

A. Latif, B. Palmintier, F. Nejabatkhah and P. Jacques, "Multi-federate co-convergence with HELICS," 2024 IEEE Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT), Washington, DC, USA, 2024, pp. 1-5, doi: 10.1109/ISGT59692.2024.10454238, available at https://ieeexplore.ieee.org/document/10454238

4.1.3 HELICS Application papers

The following are a list of some of the published work using HELICS. If you have published work using HELICS you would like to see on this list, email support@helics.org with the citation and we'll review it.

J. R. Theisen, A. Bose, M. Mukherjee, D. Burgess, K. Wilhelm and M. Diedesch, "Community-Based Transactive Coordination Mechanism for Enabling Grid-Edge Systems," 2024 IEEE Texas Power and Energy Conference (TPEC),

College Station, TX, USA, 2024, pp. 1-6, doi: 10.1109/TPEC60005.2024.10472237, available at https://ieeexplore.ieee.org/document/10472237.

A. K. Bharati, V. Ajjarapu, W. Du and Y. Liu, "Role of Distributed Inverter-Based-Resources in Bulk Grid Primary Frequency Response Through HELICS Based SMTD Co-Simulation," in IEEE Systems Journal, vol. 17, no. 1, pp. 1071-1082, March 2023, doi: 10.1109/JSYST.2022.3218117 available at https://ieeexplore.ieee.org/document/9966492.

Y. Liu, R. Huang, W. Du, A. Singhal and Z. Huang, "Highly-Scalable Transmission and Distribution Dynamic Co-Simulation With 10,000+ Grid-Following and Grid-Forming Inverters," in IEEE Transactions on Power Delivery, vol. 39, no. 1, pp. 578-590, Feb. 2024, doi: 10.1109/TPWRD.2023.3302303, available at https://ieeexplore.ieee.org/abstract/document/10210125.

Panossian, N.V.; Laarabi, H.; Moffat, K.; Chang, H.; Palmintier, B.; Meintz, A.; Lipman, T.E.; Waraich, R.A. Architecture for Co-Simulation of Transportation and Distribution Systems with Electric Vehicle Charging at Scale in the San Francisco Bay Area. Energies 2023, 16, 2189. doi: 10.3390/en16052189, available at https://www.mdpi.com/1996-1073/16/5/2189.

T. Phillips, L. D. Marinovici, C. Rieger and A. Orrell, "Scalable Resilience Analysis Through Power Systems Co-Simulation," in IEEE Access, vol. 11, pp. 18205-18214, 2023, doi: 10.1109/ACCESS.2023.3246486, available at https://ieeexplore.ieee.org/document/10050105.

William Lardier, "ASGARDS-H: Enabling Advanced Smart Grid cyber-physical Attacks, Risk and Data Studies with HELICS". Masters thesis at Concordia University, 2020, available at https://spectrum.library.concordia.ca/id/eprint/987690/.

Y. Liu, R. Huang, W. Du, A. Singhal and Z. Huang, "High-Performance Transmission and Distribution Co-simulation with 10,000+ Inverter-Based Resources," 2022 IEEE Industry Applications Society Annual Meeting (IAS), Detroit, MI, USA, 2022, pp. 1-5, doi: 10.1109/IAS54023.2022.9940149, available at https://ieeexplore.ieee.org/document/9940149.

Narayanan, Anantha, and Hardy, Trevor. "Synthetic Data Generation for Machine Learning Model Training for Energy Theft Scenarios Using Cosimulation". United Kingdom: N. p., 2022. Web. doi:10.1049/gtd2.12619, available at https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/gtd2.12619.

Jing Wang, Prateek Munankarmi, Jeff Maguire, Chengnan Shi, Wangda Zuo, David Roberts, Xin Jin. "Carbon emission responsive building control: A case study with an all-electric residential community in a cold climate". Applied Energy, Volume 314,2022, 118910, doi: 10.1016/j.apenergy.2022.118910, available at https://www.sciencedirect.com/science/article/pii/S0306261922003336.

A. Singhal, T. L. Vu and W. Du, "Consensus Control for Coordinating Grid-Forming and Grid-Following Inverters in Microgrids," in IEEE Transactions on Smart Grid, vol. 13, no. 5, pp. 4123-4133, Sept. 2022, doi: 10.1109/TSG.2022.3158254, available at https://ieeexplore.ieee.org/document/9732444.

N. Gray, R. Sadnan, A. Bose and A. Dubey, "Effects of Communication Network Topology on Distributed Optimal Power Flow for Radial Distribution Networks," 2021 North American Power Symposium (NAPS), College Station, TX, USA, 2021, pp. 1-6, doi: 10.1109/NAPS52732.2021.9654692, available at https://ieeexplore.ieee.org/document/9654692.

A. O. Abbas and B. H. Chowdhury, "A Stochastic Optimization Framework for Realizing Combined Value Streams From Customer-Side Resources," in IEEE Transactions on Smart Grid, vol. 13, no. 2, pp. 1139-1150, March 2022, doi: 10.1109/TSG.2021.3135155, available at https://ieeexplore.ieee.org/document/9650528.

J. Wang, J. Simpson, R. Yang, B. Palmintier, S. Tiwari and Y. Zhang, "Performance Evaluation of an Advanced Distributed Energy Resource Management Algorithm," 2021 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm), Aachen, Germany, 2021, pp. 378-384, doi: 10.1109/SmartGridComm51999.2021.9632298, available at https://ieeexplore.ieee.org/document/9632298.

H. M. Mustafa et al., "Cyber-Power Co-Simulation for End-to-End Synchrophasor Network Analysis and Applications," 2021 IEEE International Conference on Communications, Control, and Computing Technologies for Smart

Grids (SmartGridComm), Aachen, Germany, 2021, pp. 164-169, doi: 10.1109/SmartGridComm51999.2021.9632286, available at https://ieeexplore.ieee.org/document/9632286.

Francis K. Tuffner, John Undrill, Don Scoffield, Joseph H. Eto, Dmitry Kosterev, and Ryan D. Quint, 2021. "Distribution-Level Impacts of Plug-in Electric Vehicle Charging on the Transmission System during Fault Conditions". United States. doi: 10.2172/1832905, avialble at https://www.osti.gov/servlets/purl/1832905.

M. Mukherjee, T. Hardy, J. Fuller and A. Bose, "Transactive Implementation of Decentralized Electricity Market for Grid-Edge Systems," 2021 IEEE Power & Energy Society General Meeting (PESGM), Washington, DC, USA, 2021, pp. 1-5, doi: 10.1109/PESGM46819.2021.9638098, available at https://ieeexplore.ieee.org/document/9638098.

Monish Mukherjee, Trevor Hardy, Jason C. Fuller, Anjan Bose. "Implementing Multi-Settlement Decentralized Electricity Market Design for Transactive Communities with Imperfect Communication. Applied Energy, Volume 306, Part A, 2022, 117979, doi: 10.1016/j.apenergy.2021.117979. https://www.sciencedirect.com/science/article/pii/S0306261921012824.

Michael Blonsky, Jeff Maguire, Killian McKenna, Dylan Cutler, Sivasathya Pradha Balamurugan, Xin Jin. "OCHRE: The Object-oriented, Controllable, High-resolution Residential Energy Model for Dynamic Integration Studies". Applied Energy, Volume 290, 2021,116732, doi: 10.1016/j.apenergy.2021.116732, available at https://www.sciencedirect.com/science/article/pii/S0306261921002464.

Prateek Munankarmi, Jeff Maguire, Sivasathya Pradha Balamurugan, Michael Blonsky, David Roberts, Xin Jin. "Community-scale interaction of energy efficiency and demand flexibility in residential buildings". Applied Energy, Volume 298, 2021, 117149, doi: 10.1016/j.apenergy.2021.117149, available at https://www.sciencedirect.com/science/article/pii/S0306261921005754.

N. Duan, N. Yee, B. Salazar, J. -Y. You, E. Stewart and E. Cortez, "Cybersecurity Analysis of Distribution Grid Operation with Distributed Energy Resources via Co-Simulation," 2020 IEEE Power & Energy Society General Meeting (PESGM), Montreal, QC, Canada, 2020, pp. 1-5, doi: 10.1109/PESGM41954.2020.9281757, available at https://ieeexplore.ieee.org/document/9281757.

W. Wang, X. Fang and A. Florita, "Impact of DER Communication Delay in AGC: Cyber-Physical Dynamic Co-simulation," 2021 IEEE 48th Photovoltaic Specialists Conference (PVSC), Fort Lauderdale, FL, USA, 2021, pp. 2616-2620, doi: 10.1109/PVSC43889.2021.9518779, available at https://ieeexplore.ieee.org/document/9518779.

N. Panossian, T. Elgindy, B. Palmintier and D. Wallison, "Synthetic, Realistic Transmission and Distribution Co-Simulation for Voltage Control Benchmarking," 2021 IEEE Texas Power and Energy Conference (TPEC), College Station, TX, USA, 2021, pp. 1-5, doi: 10.1109/TPEC51183.2021.9384935, available at https://ieeexplore.ieee.org/document/9384935

A. K. Bharati and V. Ajjarapu, "A Scalable Multi-Timescale T&D Co-Simulation Framework using HELICS," 2021 IEEE Texas Power and Energy Conference (TPEC), College Station, TX, USA, 2021, pp. 1-6, doi: 10.1109/TPEC51183.2021.9384985, available at https://ieeexplore.ieee.org/document/9384985.

N. Duan, C. Huang, C. -C. Sun and L. Min, "Smart Meters Enabling Voltage Monitoring and Control: The Last-Mile Voltage Stability Issue," in IEEE Transactions on Industrial Informatics, vol. 18, no. 1, pp. 677-687, Jan. 2022, doi: 10.1109/TII.2021.3062628, available at https://ieeexplore.ieee.org/document/9366429.

Dylan Cutler, Ted Kwasnik, Sivasathya Balamurugan, Tarek Elgindy, Siddharth Swaminathan, Jeff Maguire, Dane Christensen. "Co-Simulation of Transactive Energy Markets: A Framework for Market Testing and Evaluation". International Journal of Electrical Power & Energy Systems,Volume 128, 2021, 106664, doi: 10.1016/j.ijepes.2020.106664, available at https://www.sciencedirect.com/science/article/pii/S014206152034209.

J. Zhang et al., "Development of HELICS-based High-Performance Cyber-Physical Co-simulation Framework for Distributed Energy Resources Applications," 2020 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm), Tempe, AZ, USA, 2020, pp. 1-5, doi: 10.1109/SmartGridComm47815.2020.9302977, available at https://ieeexplore.ieee.org/abstract/document/9302977.

Bhattarai B.P., L.D. Marinovici, M. Touhiduzzaman, F.K. Tuffner, K.P. Schneider, J. Xie, and P. Thekkumparambath Mana, et al. "Studying impacts of communication system performance on dynamic stability of networked microgrid".

IET Smart Grid 3, no. 5:667 - 676, 2020, doi:10.1049/iet-stg.2019.0303, available at https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-stg.2019.0303.

4.2 API Reference

4.2.1 C API Reference

Table of Contents

1. *Enums*
2. *General function*
3. *Creation functions*
4. *Broker methods*
5. *Core methods*
6. *App methods*
7. *FederateInfo methods*
8. *Federate methods*
9. *ValueFederate*
10. *Publication methods*
11. *Input methods*
12. *MessageFederate methods*
13. *Endpoint methods*
14. *Message object methods*
15. *Data buffer methods*
16. *FilterFederate methods*
17. *Filter methods*
18. *Translator methods*
19. *Query methods*

Enums

enumerator **HELICS_ITERATION_REQUEST_NO_ITERATION**

no iteration is requested

enumerator **HELICS_ITERATION_REQUEST_FORCE_ITERATION**

force iteration return when able

enumerator **HELICS_ITERATION_REQUEST_ITERATE_IF_NEEDED**

only return an iteration if necessary

enumerator **HELICS_ITERATION_REQUEST_HALT_OPERATIONS**

halt the federate

enumerator **HELICS_ITERATION_REQUEST_ERROR**

indicate there was an error

enumerator **HELICS_ITERATION_RESULT_NEXT_STEP**

the iterations have progressed to the next time

enumerator **HELICS_ITERATION_RESULT_ERROR**

there was an error

enumerator **HELICS_ITERATION_RESULT_HALTED**

the federation has halted

enumerator **HELICS_ITERATION_RESULT_ITERATING**

the federate is iterating at current time

enumerator **HELICS_STATE_UNKNOWN**

enumerator **HELICS_STATE_STARTUP**

used when no information is available about the federate state when created the federate is in startup state

enumerator **HELICS_STATE_INITIALIZATION**

entered after the enterInitializingMode call has returned

enumerator **HELICS_STATE_EXECUTION**

entered after the enterExectuationState call has returned

enumerator **HELICS_STATE_FINALIZE**

the federate has finished executing normally final values may be retrieved

enumerator **HELICS_STATE_ERROR**

error state no core communication is possible but values can be retrieved

enumerator **HELICS_STATE_PENDING_INIT**

indicator that the federate is pending entry to initialization state

enumerator **HELICS_STATE_PENDING_EXEC**

state pending EnterExecution State

enumerator **HELICS_STATE_PENDING_TIME**

state that the federate is pending a timeRequest

enumerator **HELICS_STATE_PENDING_ITERATIVE_TIME**

state that the federate is pending an iterative time request

enumerator **HELICS_STATE_PENDING_FINALIZE**

state that the federate is pending a finalize request

enumerator **HELICS_STATE_FINISHED**

state that the federate is finished simulating but still connected

enumerator **HELICS_CORE_TYPE_DEFAULT**

a default core type that will default to something available

enumerator **HELICS_CORE_TYPE_ZMQ**

use the Zero MQ networking protocol

enumerator **HELICS_CORE_TYPE_MPI**

use MPI for operation on a parallel cluster

enumerator **HELICS_CORE_TYPE_TEST**

use the Test core if all federates are in the same process

enumerator **HELICS_CORE_TYPE_INTERPROCESS**

interprocess uses memory mapped files to transfer data (for use when all federates are on the same machine

Warning

doxygenenumvalue: Cannot find enumvalue "HELICS_CORE_TYPE_" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

enumerator **HELICS_CORE_TYPE_IPC**

interprocess uses memory mapped files to transfer data (for use when all federates are on the same machine ipc is the same as /ref HELICS_CORE_TYPE_interprocess

enumerator **HELICS_CORE_TYPE_TCP**

use a generic TCP protocol message stream to send messages

enumerator **HELICS_CORE_TYPE_UDP**

use UDP packets to send the data

enumerator **HELICS_CORE_TYPE_ZMQ_SS**

single socket version of ZMQ core usually for high fed count on the same system

enumerator **HELICS_CORE_TYPE_NNG**

for using the nanomsg communications

enumerator **HELICS_CORE_TYPE_TCP_SS**

a single socket version of the TCP core for more easily handling firewalls

enumerator **HELICS_CORE_TYPE_HTTP**

a core type using http for communication

enumerator **HELICS_CORE_TYPE_WEBSOCKET**

a core using websockets for communication

enumerator **HELICS_CORE_TYPE_INPROC**

an in process core type for handling communications in shared memory it is pretty similar to the test core but stripped from the "test" components

enumerator **HELICS_CORE_TYPE_NULL**

an explicit core type that is recognized but explicitly doesn't exist, for testing and a few other assorted reasons

enumerator **HELICS_CORE_TYPE_EMPTY**

an explicit core type exists but does nothing but return empty values or sink calls

enumerator **HELICS_CORE_TYPE_EXTRACT**

core type specification to allow extraction from later arguments or files

enumerator **HELICS_DATA_TYPE_UNKNOWN**

enumerator **HELICS_DATA_TYPE_STRING**

a sequence of characters

enumerator **HELICS_DATA_TYPE_DOUBLE**

a double precision floating point number

enumerator **HELICS_DATA_TYPE_INT**

a 64 bit integer

enumerator **HELICS_DATA_TYPE_COMPLEX**

a pair of doubles representing a complex number

enumerator **HELICS_DATA_TYPE_VECTOR**

an array of doubles

enumerator **HELICS_DATA_TYPE_COMPLEX_VECTOR**

a complex vector object

enumerator **HELICS_DATA_TYPE_NAMED_POINT**

a named point consisting of a string and a double

enumerator **HELICS_DATA_TYPE_BOOLEAN**

a boolean data type

enumerator **HELICS_DATA_TYPE_TIME**

time data type

enumerator **HELICS_DATA_TYPE_CHAR**

data type for a single character

enumerator **HELICS_DATA_TYPE_RAW**

raw data type

enumerator **HELICS_DATA_TYPE_JSON**

type converts to a valid json string

enumerator **HELICS_DATA_TYPE_MULTI**

the data type can change

enumerator **HELICS_DATA_TYPE_ANY**

open type that can be anything

enumerator **HELICS_FLAG_OBSERVER**

flag indicating that a federate is observe only

enumerator **HELICS_FLAG_UNINTERRUPTIBLE**

flag indicating that a federate can only return requested times

enumerator **HELICS_FLAG_INTERRUPTIBLE**

flag indicating that a federate can be interrupted

enumerator **HELICS_FLAG_SOURCE_ONLY**

flag indicating that a federate/interface is a signal generator only

enumerator **HELICS_FLAG_ONLY_TRANSMIT_ON_CHANGE**

flag indicating a federate/interface should only transmit values if they have changed(binary equivalence)

enumerator **HELICS_FLAG_ONLY_UPDATE_ON_CHANGE**

flag indicating a federate/interface should only trigger an update if a value has changed (binary equivalence)

enumerator **HELICS_FLAG_WAIT_FOR_CURRENT_TIME_UPDATE**

flag indicating a federate should only grant time if all other federates have already passed the requested time

enumerator **HELICS_FLAG_RESTRICTIVE_TIME_POLICY**

flag indicating a federate should operate on a restrictive time policy, which disallows some 2nd order time evaluation and can be useful for certain types of dependency cycles and update patterns, but generally shouldn't be used as it can lead to some very slow update conditions

enumerator **HELICS_FLAG_ROLLBACK**

flag indicating that a federate has rollback capability

enumerator **HELICS_FLAG_FORWARD_COMPUTE**

flag indicating that a federate performs forward computation and does internal rollback

enumerator **HELICS_FLAG_REALTIME**

flag indicating that a federate needs to run in real time

enumerator **HELICS_FLAG_SINGLE_THREAD_FEDERATE**

flag indicating that the federate will only interact on a single thread

enumerator **HELICS_FLAG_MULTI_THREAD_CORE**

flag indicating use of a thread safe core

enumerator **HELICS_FLAG_SINGLE_THREAD_CORE**

flag indicating use of a single threaded core

enumerator **HELICS_FLAG_REENTRANT**

flag indicating the federate can be reentrant

enumerator **HELICS_FLAG_IGNORE_TIME_MISMATCH_WARNINGS**

used to not display warnings on mismatched requested times

enumerator **HELICS_FLAG_STRICT_CONFIG_CHECKING**

specify that checking on configuration files should be strict and throw and error on any invalid values

enumerator **HELICS_FLAG_USE_JSON_SERIALIZATION**

specify that the federate should use json serialization for all data types

enumerator **HELICS_FLAG_EVENT_TRIGGERED**

specify that the federate is event triggered-meaning (all/most) events are triggered by incoming events

enumerator **HELICS_FLAG_LOCAL_PROFILING_CAPTURE**

specify that that federate should capture the profiling data to the local federate logging system

enumerator **HELICS_FLAG_CALLBACK_FEDERATE**

specify that the federate is a callback based federate using callbacks for execution

enumerator **HELICS_FLAG_AUTOMATED_TIME_REQUEST**

specify that a federate should automatically call timeRequest on completion of current request

enumerator **HELICS_FLAG_DELAY_INIT_ENTRY**

used to delay a core from entering initialization mode even if it would otherwise be ready

enumerator **HELICS_FLAG_ENABLE_INIT_ENTRY**

used to clear the HELICS_DELAY_INIT_ENTRY flag in cores

enumerator **HELICS_FLAG_IGNORE**

ignored flag used to test some code paths

enumerator **HELICS_FLAG_SLOW_RESPONDING**

flag specifying that a federate, core, or broker may be slow to respond to pings If the federate goes offline there is no good way to detect it so use with caution

enumerator **HELICS_FLAG_DEBUGGING**

flag specifying the federate/core/broker is operating in a user debug mode so deadlock timers and timeout are disabled this flag is a combination of slow_responding and disabling of some timeouts

enumerator **HELICS_FLAG_TERMINATE_ON_ERROR**

specify that a federate error should terminate the federation

enumerator **HELICS_FLAG_FORCE_LOGGING_FLUSH**

specify that the log files should be flushed on every log message

enumerator **HELICS_FLAG_DUMPLOG**

specify that a full log should be dumped into a file

enumerator **HELICS_FLAG_PROFILING**

specify that helics should capture profiling data

enumerator **HELICS_FLAG_PROFILING_MARKER**

flag trigger for generating a profiling marker

enumerator **HELICS_FLAG_ALLOW_REMOTE_CONTROL**

specify that the federate/core/broker should allow some remote control operations such as finalize

enumerator **HELICS_FLAG_DISABLE_REMOTE_CONTROL**

specify that the federate/core/broker should *NOT* allow some remote control operations such as finalize

enumerator **HELICS_LOG_LEVEL_DUMPLOG**

log level for dumping log messages

enumerator **HELICS_LOG_LEVEL_NO_PRINT**

don't print anything except a few catastrophic errors

enumerator **HELICS_LOG_LEVEL_ERROR**

only print error level indicators

enumerator **HELICS_LOG_LEVEL_PROFILING**

profiling log level

enumerator **HELICS_LOG_LEVEL_WARNING**

only print warnings and errors

enumerator **HELICS_LOG_LEVEL_SUMMARY**
warning errors and summary level information

enumerator **HELICS_LOG_LEVEL_CONNECTIONS**
summary+ notices about federate and broker connections +messages about network connections

enumerator **HELICS_LOG_LEVEL_INTERFACES**
connections+ interface definitions

enumerator **HELICS_LOG_LEVEL_TIMING**
interfaces + timing message

enumerator **HELICS_LOG_LEVEL_DATA**
timing+ data transfer notices

enumerator **HELICS_LOG_LEVEL_DEBUG**
data+ additional debug message

enumerator **HELICS_LOG_LEVEL_TRACE**
all internal messages

enumerator **HELICS_ERROR_FATAL**
global fatal error for federation

enumerator **HELICS_ERROR_EXTERNAL_TYPE**
an unknown non-helics error was produced

enumerator **HELICS_ERROR_OTHER**
the function produced a helics error of some other type

enumerator **HELICS_USER_EXCEPTION**
user code generated exception

enumerator **HELICS_ERROR_INSUFFICIENT_SPACE**
insufficient space is available to store requested data

enumerator **HELICS_ERROR_EXECUTION_FAILURE**
the function execution has failed

enumerator **HELICS_ERROR_INVALID_FUNCTION_CALL**
the call made was invalid in the present state of the calling object

enumerator **HELICS_ERROR_INVALID_STATE_TRANSITION**
error issued when an invalid state transition occurred

enumerator **HELICS_ERROR_SYSTEM_FAILURE**

the federate has terminated unexpectedly and the call cannot be completed

enumerator **HELICS_ERROR_DISCARD**

the input was discarded and not used for some reason

enumerator **HELICS_ERROR_INVALID_ARGUMENT**

the parameter passed was invalid and unable to be used

enumerator **HELICS_ERROR_INVALID_OBJECT**

indicator that the object used was not a valid object

enumerator **HELICS_ERROR_CONNECTION_FAILURE**

the operation to connect has failed

enumerator **HELICS_ERROR_REGISTRATION_FAILURE**

registration has failed

enumerator **HELICS_ERROR_USER_ABORT**

user system abort to match typical SIGINT value

enumerator **HELICS_ERROR_TERMINATED**

force termination to match typical SIGKILL value

enumerator **HELICS_PROPERTY_TIME_DELTA**

the property controlling the minimum time delta for a federate

enumerator **HELICS_PROPERTY_TIME_PERIOD**

the property controlling the period for a federate

enumerator **HELICS_PROPERTY_TIME_OFFSET**

the property controlling time offset for the period of federate

enumerator **HELICS_PROPERTY_TIME_RT_LAG**

the property controlling real time lag for a federate the max time a federate can lag real time

enumerator **HELICS_PROPERTY_TIME_RT_LEAD**

the property controlling real time lead for a federate the max time a federate can be ahead of real time

enumerator **HELICS_PROPERTY_TIME_RT_TOLERANCE**

the property controlling real time tolerance for a federate sets both rt_lag and rt_lead

enumerator **HELICS_PROPERTY_TIME_INPUT_DELAY**

the property controlling input delay for a federate

enumerator **HELICS_PROPERTY_TIME_OUTPUT_DELAY**
the property controlling output delay for a federate

enumerator **HELICS_PROPERTY_TIME_STOPTIME**
specify the maximum time of a federate

enumerator **HELICS_PROPERTY_TIME_GRANT_TIMEOUT**
the property specifying a timeout to trigger actions if the time for granting exceeds a certain threshold

enumerator **HELICS_PROPERTY_INT_CURRENT_ITERATION**
read only property getting the current iteration count

enumerator **HELICS_PROPERTY_INT_MAX_ITERATIONS**
integer property controlling the maximum number of iterations in a federate

enumerator **HELICS_PROPERTY_INT_LOG_LEVEL**
integer property controlling the log level in a federate see HelicsLogLevels

enumerator **HELICS_PROPERTY_INT_FILE_LOG_LEVEL**
integer property controlling the log level for file logging in a federate see HelicsLogLevels

enumerator **HELICS_PROPERTY_INT_CONSOLE_LOG_LEVEL**
integer property controlling the log level for console logging in a federate see HelicsLogLevels

enumerator **HELICS_PROPERTY_INT_LOG_BUFFER**
integer property controlling the size of the log buffer

enumerator **HELICS_PROPERTY_INT_VALUE_BUFFER_WARNING**
integer property specifying queued value buffer warning threshold in MiB

enumerator **HELICS_PROPERTY_INT_INDEX_GROUP**
integer property specifying an index group

enumerator **HELICS_MULTI_INPUT_NO_OP**
time and priority order the inputs from the core library

enumerator **HELICS_MULTI_INPUT_VECTORIZE_OPERATION**
vectorize the inputs either double vector or string vector

enumerator **HELICS_MULTI_INPUT_AND_OPERATION**
all inputs are assumed to be boolean and all must be true to return true

enumerator **HELICS_MULTI_INPUT_OR_OPERATION**
all inputs are assumed to be boolean and at least one must be true to return true

enumerator **HELICS_MULTI_INPUT_SUM_OPERATION**

sum all the inputs

enumerator **HELICS_MULTI_INPUT_DIFF_OPERATION**

do a difference operation on the inputs, first-sum(rest) for double input, vector diff for vector input

enumerator **HELICS_MULTI_INPUT_MAX_OPERATION**

find the max of the inputs

enumerator **HELICS_MULTI_INPUT_MIN_OPERATION**

find the min of the inputs

enumerator **HELICS_MULTI_INPUT_AVERAGE_OPERATION**

take the average of the inputs

enumerator **HELICS_HANDLE_OPTION_CONNECTION_REQUIRED**

specify that a connection is required for an interface and will generate an error if not available

enumerator **HELICS_HANDLE_OPTION_CONNECTION_OPTIONAL**

specify that a connection is NOT required for an interface and will only be made if available no warning will be issues if not available

enumerator **HELICS_HANDLE_OPTION_SINGLE_CONNECTION_ONLY**

specify that only a single connection is allowed for an interface

enumerator **HELICS_HANDLE_OPTION_MULTIPLE_CONNECTIONS_ALLOWED**

specify that multiple connections are allowed for an interface

enumerator **HELICS_HANDLE_OPTION_BUFFER_DATA**

specify that the last data should be buffered and sent on subscriptions after init

enumerator **HELICS_HANDLE_OPTION_RECONNECTABLE**

specify that the handle can be reconnected for reentrant federates

enumerator **HELICS_HANDLE_OPTION_STRICT_TYPE_CHECKING**

specify that the types should be checked strictly for pub/sub and filters

enumerator **HELICS_HANDLE_OPTION_RECEIVE_ONLY**

specify that the handle is receive only

enumerator **HELICS_HANDLE_OPTION_SOURCE_ONLY**

specify that the handle is source only

enumerator **HELICS_HANDLE_OPTION_IGNORE_UNIT_MISMATCH**

specify that the mismatching units should be ignored

enumerator **HELICS_HANDLE_OPTION_ONLY_TRANSMIT_ON_CHANGE**

specify that an interface will only transmit on change(only applicable to publications)

enumerator **HELICS_HANDLE_OPTION_ONLY_UPDATE_ON_CHANGE**

specify that an interface will only update if the value has actually changed

enumerator **HELICS_HANDLE_OPTION_IGNORE_INTERRUPTS**

specify that an interface does not participate in determining time interrupts

enumerator **HELICS_HANDLE_OPTION_MULTI_INPUT_HANDLING_METHOD**

specify the multi-input processing method for inputs

enumerator **HELICS_HANDLE_OPTION_INPUT_PRIORITY_LOCATION**

specify the source index with the highest priority

enumerator **HELICS_HANDLE_OPTION_CLEAR_PRIORITY_LIST**

specify that the priority list should be cleared or question if it is cleared

enumerator **HELICS_HANDLE_OPTION_CONNECTIONS**

specify the required number of connections or get the actual number of connections

enumerator **HELICS_HANDLE_OPTION_TIME_RESTRICTED**

specify that the interface only sends or receives data at specified intervals

enumerator **HELICS_FILTER_TYPE_CUSTOM**

a custom filter type that executes a user defined callback

enumerator **HELICS_FILTER_TYPE_DELAY**

a filter type that executes a fixed delay on a message

enumerator **HELICS_FILTER_TYPE_RANDOM_DELAY**

a filter type that executes a random delay on the messages

enumerator **HELICS_FILTER_TYPE_RANDOM_DROP**

a filter type that randomly drops messages

enumerator **HELICS_FILTER_TYPE_REROUTE**

a filter type that reroutes a message to a different destination than originally specified

enumerator **HELICS_FILTER_TYPE_CLONE**

a filter type that duplicates a message and sends the copy to a different destination

enumerator **HELICS_FILTER_TYPE_FIREWALL**

a customizable filter type that can perform different actions on a message based on firewall like rules

enumerator **HELICS_TRANSLATOR_TYPE_CUSTOM**

a custom filter type that executes a user defined callback

enumerator **HELICS_TRANSLATOR_TYPE_JSON**

a translator type that converts to and from JSON

enumerator **HELICS_TRANSLATOR_TYPE_BINARY**

a translator type that just encodes the message again in binary

enumerator **HELICS_SEQUENCING_MODE_FAST**

sequencing mode to operate on priority channels

enumerator **HELICS_SEQUENCING_MODE_ORDERED**

sequencing mode to operate on the normal channels

enumerator **HELICS_SEQUENCING_MODE_DEFAULT**

select the default channel

General

const char ***helicsGetVersion**(void)

Get a version string for HELICS.

const char ***helicsGetBuildFlags**(void)

Get the build flags used to compile HELICS.

const char ***helicsGetCompilerVersion**(void)

Get the compiler version used to compile HELICS.

const char ***helicsGetSystemInfo**(void)

Get a json formatted system information string, containing version info. The string contains fields with system information like cpu, core count, operating system, and memory, as well as information about the HELICS build. Used for debugging reports and gathering other information.

HelicsError **helicsErrorInitialize**(void)

Return an initialized error object.

void **helicsErrorClear**(HelicsError *err)

Clear an error object.

clear an error object

void **helicsLoadSignalHandler**()

Load a signal handler that handles Ctrl-C and shuts down all HELICS brokers, cores, and federates then exits the process.

void **helicsLoadThreadedSignalHandler**()

Load a signal handler that handles Ctrl-C and shuts down all HELICS brokers, cores, and federates then exits the process. This operation will execute in a newly created and detached thread returning control back to the calling program before completing operations.

void **helicsClearSignalHandler**()

Clear HELICS based signal handlers.

void **helicsLoadSignalHandlerCallback**(HelicsBool (*handler)(int), HelicsBool useSeparateThread)

Load a custom signal handler to execute prior to the abort signal handler.

This function is not 100% reliable it will most likely work but uses some functions and techniques that are not 100% guaranteed to work in a signal handler and in worst case it could deadlock. That is somewhat unlikely given usage patterns but it is possible. The callback has signature HelicsBool(*handler)(int) and it will take the SIG_INT as an argument and return a boolean. If the boolean return value is HELICS_TRUE (or the callback is null) the default signal handler is run after the callback finishes; if it is HELICS_FALSE the default callback is not run and the default signal handler is executed. If the second argument is set to HELICS_TRUE the default signal handler will execute in a separate thread(this may be a bad idea).

void **helicsLoadSignalHandlerCallbackNoExit**(HelicsBool (*handler)(int), HelicsBool useSeparateThread)

Load a custom signal handler to execute prior to the abort signal handler. The signal handler does not call exit.

This function is not 100% reliable. It will most likely work but uses some functions and techniques that are not 100% guaranteed to work in a signal handler and in worst case it could deadlock. That is somewhat unlikely given usage patterns but it is possible. The callback has signature HelicsBool(*handler)(int) and it will take the SIG_INT as an argument and return a boolean. If the boolean return value is HELICS_TRUE (or the callback is null) the no exit signal handler is run after the callback finishes; if it is HELICS_FALSE the default callback is not run and the default signal handler is executed. If the second argument is set to HELICS_TRUE the default signal handler will execute in a separate thread (this may be a bad idea).

void **helicsAbort**(int errorCode, const char *errorString)

Execute a global abort by sending an error code to all cores, brokers, and federates that were created through the current library instance.

HelicsBool **helicsIsCoreTypeAvailable**(const char *type)

Returns true if core/broker type specified is available in current compilation.

Options include “zmq”, “udp”, “ipc”, “interprocess”, “tcp”, “default”, “mpi”.

Parameters

type – A string representing a core type.

HelicsFederate **helicsGetFederateByName**(const char *fedName, HelicsError *err)

Get an existing federate object from a core by name.

The federate must have been created by one of the other functions and at least one of the objects referencing the created federate must still be active in the process.

Parameters

- **fedName** – The name of the federate to retrieve.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

NULL if no fed is available by that name otherwise a HelicsFederate with that name.

int **helicsGetPropertyIndex**(const char *val)

Get a property index for use in /ref helicsFederateInfoSetFlagOption, /ref helicsFederateInfoSetTimeProperty, or /ref helicsFederateInfoSetIntegerProperty

Parameters

val – A string with the property name.

Returns

An int with the property code, (-1) if val is NULL, or HELICS_INVALID_OPTION_INDEX if not a valid property.

int **helicsGetFlagIndex**(const char *val)

Get a property index for use in /ref helicsFederateInfoSetFlagOption, /ref helicsFederateSetFlagOption,

Parameters

val – A string with the option name.

Returns

An int with the flag index, (-1) if val is NULL, or HELICS_INVALID_OPTION_INDEX if not a valid flag.

int **helicsGetOptionIndex**(const char *val)

Get an option index for use in /ref helicsPublicationSetOption, /ref helicsInputSetOption, /ref helicsEndpointSetOption, /ref helicsFilterSetOption, and the corresponding get functions.

Parameters

val – A string with the option name.

Returns

An int with the option index, (-1) if val is NULL, or HELICS_INVALID_OPTION_INDEX if not a valid option.

int **helicsGetOptionValue**(const char *val)

Get an option value for use in /ref helicsPublicationSetOption, /ref helicsInputSetOption, /ref helicsEndpointSetOption, /ref helicsFilterSetOption.

Parameters

val – A string representing the value.

Returns

An int with the option value, (-1) if val is NULL, or HELICS_INVALID_OPTION_INDEX if not a valid value.

int **helicsGetDataType**(const char *val)

Get the data type for use in /ref helicsFederateRegisterPublication, /ref helicsFederateRegisterInput, /ref helicsFilterSetOption.

Parameters

val – A string representing a data type.

Returns

An int with the data type or HELICS_DATA_TYPE_UNKNOWN(-1) if not a valid value.

void **helicsCloseLibrary**(void)

Call when done using the helics library. This function will ensure the threads are closed properly. If possible this should be the last call before exiting.

void **helicsCleanupLibrary**(void)

Function to do some housekeeping work.

This runs some cleanup routines and tries to close out any residual thread that haven't been shutdown yet.

void **helicsCallbackFederateNextTimeCallback**(HelicsFederate fed, HelicsTime (*timeUpdate)(HelicsTime time, void *userdata), void *userdata, HelicsError *err)

Set callback for the next time update.

This callback will be triggered to compute the next time update for a callback federate.

Parameters

- **fed** – The federate to set the callback for.

- **timeUpdate** – A callback with signature HelicsTime(HelicsTime time, void *userdata);
- **userdata** – A pointer to user data that is passed to the function when executing.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsCallbackFederateNextTimeIterativeCallback**(HelicsFederate fed, HelicsTime (*timeUpdate)(HelicsTime time, HelicsIterationResult result, HelicsIterationRequest *iteration, void *userdata), void *userdata, HelicsError *err)

Set callback for the next time update with iteration capability.

This callback will be triggered to compute the next time update for a callback federate.

Parameters

- **fed** – The federate to set the callback for.
- **timeUpdate** – A callback with signature HelicsTime(HelicsTime time, HelicsIterationResult result, HelicsIterationRequest* iteration, ;
- **userdata** – A pointer to user data that is passed to the function when executing.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsCallbackFederateInitializeCallback**(HelicsFederate fed, HelicsIterationRequest (*initialize)(void *userdata), void *userdata, HelicsError *err)

Set callback for initialization.

This callback will be executed when computing whether to iterate in initialization mode.

Parameters

- **fed** – The federate to set the callback for.
- **initialize** – A callback with signature HelicsIterationRequest(void *userdata);
- **userdata** – A pointer to user data that is passed to the function when executing.
- **err** – **[inout]** A pointer to an error object for catching errors.

HelicsDataBuffer **helicsWrapDataInBuffer**(void *data, int dataSize, int dataCapacity)

wrap user data in a buffer object

Creation

HelicsCore **helicsCreateCore**(const char *type, const char *name, const char *initString, HelicsError *err)

Create a core object.

If the core is invalid, err will contain the corresponding error message and the returned object will be NULL.

Parameters

- **type** – The type of the core to create.
- **name** – The name of the core. It can be a nullptr or empty string to have a name automatically assigned.
- **initString** – An initialization string to send to the core. The format is similar to command line arguments. Typical options include a broker name, the broker address, the number of federates, etc. Can also be a file (toml, ini, json) or json object containing the core configuration.

- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns
A HelicsCore object.

HelicsCore **helicsCreateCoreFromArgs**(const char *type, const char *name, int argc, const char *const *argv, HelicsError *err)

Create a core object by passing command line arguments.

Parameters

- **type** – The type of the core to create.
- **name** – The name of the core. It can be a nullptr or empty string to have a name automatically assigned.
- **argc** – The number of arguments.
- **argv** – The list of string values from a command line.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns
A HelicsCore object.

HelicsBroker **helicsCreateBroker**(const char *type, const char *name, const char *initString, HelicsError *err)

Create a broker object.

It will be NULL if there was an error indicated in the err object.

Parameters

- **type** – The type of the broker to create.
- **name** – The name of the broker. It can be a nullptr or empty string to have a name automatically assigned.
- **initString** – An initialization string to send to the core-the format is similar to command line arguments. Typical options include a broker address such as —broker=”XSSAF” if this is a subbroker, or the number of federates, or it can also be a json or toml file with broker configuration. or the address.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns
A HelicsBroker object.

HelicsBroker **helicsCreateBrokerFromArgs**(const char *type, const char *name, int argc, const char *const *argv, HelicsError *err)

Create a core object by passing command line arguments.

Parameters

- **type** – The type of the core to create.
- **name** – The name of the core. It can be a nullptr or empty string to have a name automatically assigned.
- **argc** – The number of arguments.
- **argv** – The list of string values from a command line.

- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns
A HelicsCore object.

HelicsFederate **helicsCreateValueFederate**(const char *fedName, HelicsFederateInfo fedInfo, HelicsError *err)

Create a value federate from a federate info object.

HelicsFederate objects can be used in all functions that take a HelicsFederate or HelicsFederate object as an argument.

Parameters

- **fedName** – The name of the federate to create, can NULL or an empty string to use the default name from fedInfo or an assigned name.
- **fedInfo** – The federate info object that contains details on the federate.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns
An opaque value federate object.

HelicsFederate **helicsCreateValueFederateFromConfig**(const char *configFile, HelicsError *err)

Create a value federate from a JSON file, JSON string, or TOML file.

HelicsFederate objects can be used in all functions that take a HelicsFederate or HelicsFederate object as an argument.

Parameters

- **configFile** – A JSON file or a JSON string or TOML file that contains setup and configuration information.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns
An opaque value federate object.

HelicsFederate **helicsCreateMessageFederate**(const char *fedName, HelicsFederateInfo fedInfo, HelicsError *err)

Create a message federate from a federate info object.

helics_message_federate objects can be used in all functions that take a helics_message_federate or HelicsFederate object as an argument.

Parameters

- **fedName** – The name of the federate to create.
- **fedInfo** – The federate info object that contains details on the federate.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns
An opaque message federate object.

HelicsFederate **helicsCreateMessageFederateFromConfig**(const char *configFile, HelicsError *err)

Create a message federate from a JSON file or JSON string or TOML file.

helics_message_federate objects can be used in all functions that take a helics_message_federate or HelicsFederate object as an argument.

Parameters

- **configFile** – A Config(JSON,TOML) file or a JSON string that contains setup and configuration information.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

An opaque message federate object.

HelicsFederate **helicsCreateCombinationFederate**(const char *fedName, HelicsFederateInfo fedInfo, HelicsError *err)

Create a combination federate from a federate info object.

Combination federates are both value federates and message federates, objects can be used in all functions that take a HelicsFederate, helics_message_federate or HelicsFederate object as an argument

Parameters

- **fedName** – A string with the name of the federate, can be NULL or an empty string to pull the default name from fedInfo.
- **fedInfo** – The federate info object that contains details on the federate.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

An opaque value federate object nullptr if the object creation failed.

HelicsFederate **helicsCreateCombinationFederateFromConfig**(const char *configFile, HelicsError *err)

Create a combination federate from a JSON file or JSON string or TOML file.

Combination federates are both value federates and message federates, objects can be used in all functions that take a HelicsFederate, helics_message_federate or HelicsFederate object as an argument

Parameters

- **configFile** – A JSON file or a JSON string,TOML file, or command line arguments that contains setup and configuration information.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

An opaque combination federate object.

HelicsFederate **helicsCreateCallbackFederate**(const char *fedName, HelicsFederateInfo fedInfo, HelicsError *err)

Create a callback federate from a federate info object.

Callback federates are combination federates that run a series of callback for execution in a completely automated fashion.

Parameters

- **fedName** – A string with the name of the federate, can be NULL or an empty string to pull the default name from fedInfo.
- **fedInfo** – The federate info object that contains details on the federate.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

An opaque value federate object; nullptr if the object creation failed.

HelicsFederate **helicsCreateCallbackFederateFromConfig**(const char *configFile, HelicsError *err)

Create a callback federate from a JSON file or JSON string or TOML file.

Callback federates are combination federates that run a series of callbacks for execution in a completely automated fashion. The enterInitializingMode call transfers complete control of the federate to the Core and no further user interaction is expected.

Parameters

- **configFile** – A JSON file or a JSON string or TOML file that contains setup and configuration information.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

An opaque combination federate object.

HelicsFederateInfo **helicsCreateFederateInfo**(void)

Create a federate info object for specifying federate information when constructing a federate.

Returns

A HelicsFederateInfo object which is a reference to the created object.

HelicsQuery **helicsCreateQuery**(const char *target, const char *query)

Create a query object.

A query object consists of a target and query string.

Parameters

- **target** – The name of the target to query.
- **query** – The query to make of the target.

HelicsApp **helicsCreateApp**(const char *appName, const char *appType, const char *configFile, HelicsFederateInfo fedInfo, HelicsError *err)

Create a HelicsApp object.

Create a HelicsApp object.

Parameters

- **appName** – A string with the name of the app, can be NULL or an empty string to pull the default name from fedInfo or the config file.
- **appType** – The type of app to create.
- **configFile** – Configuration file or string to pass into the app, can be NULL or empty.
- **fedInfo** – The federate information to pass into the app, can be NULL.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

An opaque value app object, or nullptr if the object creation failed.

HelicsDataBuffer **helicsCreateDataBuffer**(int32_t initialCapacity)

create a helics managed data buffer with initial capacity

Broker

HelicsBroker **helicsBrokerClone**(HelicsBroker broker, HelicsError *err)

Create a new reference to an existing broker.

This will create a new broker object that references the existing broker it must be freed as well.

Parameters

- **broker** – An existing HelicsBroker.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

A new reference to the same broker.

HelicsBool **helicsBrokerIsValid**(HelicsBroker broker)

Check if a broker object is a valid object.

Parameters

broker – The HelicsBroker object to test.

HelicsBool **helicsBrokerIsConnected**(HelicsBroker broker)

Check if a broker is connected.

A connected broker implies it is attached to cores or cores could reach out to communicate.

Returns

HELICS_FALSE if not connected.

void **helicsBrokerDataLink**(HelicsBroker broker, const char *source, const char *target, HelicsError *err)

Link a named publication and named input using a broker.

Parameters

- **broker** – The broker to generate the connection from.
- **source** – The name of the publication (cannot be NULL).
- **target** – The name of the target to send the publication data (cannot be NULL).
- **err** – **[inout]** A HelicsError object, can be NULL if the errors are to be ignored.

void **helicsBrokerAddSourceFilterToEndpoint**(HelicsBroker broker, const char *filter, const char *endpoint, HelicsError *err)

Link a named filter to a source endpoint.

Parameters

- **broker** – The broker to generate the connection from.
- **filter** – The name of the filter (cannot be NULL).
- **endpoint** – The name of the endpoint to filter the data from (cannot be NULL).
- **err** – **[inout]** A HelicsError object, can be NULL if the errors are to be ignored.

void **helicsBrokerAddDestinationFilterToEndpoint**(HelicsBroker broker, const char *filter, const char *endpoint, HelicsError *err)

Link a named filter to a destination endpoint.

Parameters

- **broker** – The broker to generate the connection from.
- **filter** – The name of the filter (cannot be NULL).
- **endpoint** – The name of the endpoint to filter the data going to (cannot be NULL).

- **err** – **[inout]** A HelicsError object, can be NULL if the errors are to be ignored.

void **helicsBrokerMakeConnections**(HelicsBroker broker, const char *file, HelicsError *err)

Load a file containing connection information.

Parameters

- **broker** – The broker to generate the connections from.
- **file** – A JSON or TOML file containing connection information.
- **err** – **[inout]** A HelicsError object, can be NULL if the errors are to be ignored.

HelicsBool **helicsBrokerWaitForDisconnect**(HelicsBroker broker, int msToWait, HelicsError *err)

Wait for the broker to disconnect.

Parameters

- **broker** – The broker to wait for.
- **msToWait** – The time out in millisecond (<0 for infinite timeout).
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

HELICS_TRUE if the disconnect was successful, HELICS_FALSE if there was a timeout.

const char ***helicsBrokerGetIdentifier**(HelicsBroker broker)

Get an identifier for the broker.

Parameters

broker – The broker to query.

Returns

A string containing the identifier for the broker.

const char ***helicsBrokerGetAddress**(HelicsBroker broker)

Get the network address associated with a broker.

Parameters

broker – The broker to query.

Returns

A string with the network address of the broker.

void **helicsBrokerDisconnect**(HelicsBroker broker, HelicsError *err)

Disconnect a broker.

Parameters

- **broker** – The broker to disconnect.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsBrokerDestroy**(HelicsBroker broker)

Disconnect and free a broker.

void **helicsBrokerFree**(HelicsBroker broker)

Release the memory associated with a broker.

void **helicsBrokerSetGlobal**(HelicsBroker broker, const char *valueName, const char *value, HelicsError *err)

Set a federation global value.

This overwrites any previous value for this name.

Parameters

- **broker** – The broker to set the global through.
- **valueName** – The name of the global to set.
- **value** – The value of the global.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsBrokerAddAlias**(HelicsBroker broker, const char *interfaceName, const char *alias, HelicsError *err)

Create an alias for an interface.

Parameters

- **broker** – The broker to use to set the alias.
- **interfaceName** – The current name of an interface.
- **alias** – The additional name to use for the given interface.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsBrokerSendCommand**(HelicsBroker broker, const char *target, const char *command, HelicsError *err)

Send a command to another helics object through a broker using asynchronous(fast) messages.

Parameters

- **broker** – The broker to send the command through.
- **target** – The name of the object to send the command to.
- **command** – The command to send.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsBrokerSendOrderedCommand**(HelicsBroker broker, const char *target, const char *command, HelicsError *err)

Send a command to another helics object through a broker using ordered sequencing.

Parameters

- **broker** – The broker to send the command through.
- **target** – The name of the object to send the command to.
- **command** – The command to send.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsBrokerSetLogFile**(HelicsBroker broker, const char *logFileName, HelicsError *err)

Set the log file on a broker.

Parameters

- **broker** – The broker to set the log file for.
- **logFileName** – The name of the file to log to.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsBrokerSetTimeBarrier**(HelicsBroker broker, HelicsTime barrierTime, HelicsError *err)

Set a broker time barrier.

Parameters

- **broker** – The broker to set the time barrier for.

- **barrierTime** – The time to set the barrier at.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsBrokerClearTimeBarrier**(HelicsBroker broker)

Clear any time barrier on a broker.

Parameters

broker – The broker to clear the barriers on.

void **helicsBrokerGlobalError**(HelicsBroker broker, int errorCode, const char *errorString, HelicsError *err)

Generate a global error through a broker. This will terminate the federation.

Parameters

- **broker** – The broker to generate the global error on.
- **errorCode** – The error code to associate with the global error.
- **errorString** – An error message to associate with the global error.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsBrokerSetLoggingCallback**(HelicsBroker broker, void (*logger)(int loglevel, const char *identifier, const char *message, void *userData), void *userdata, HelicsError *err)

Set the logging callback to a broker.

Add a logging callback function to a broker. The logging callback will be called when a message flows into a broker from the core or from a broker.

Parameters

- **broker** – The broker object in which to set the callback.
- **logger** – A callback with signature void(int, const char *, const char *, void *); the function arguments are loglevel, an identifier, a message string, and a pointer to user data.
- **userdata** – A pointer to user data that is passed to the function when executing.
- **err** – **[inout]** A pointer to an error object for catching errors.

Core

HelicsCore **helicsCoreClone**(HelicsCore core, HelicsError *err)

Create a new reference to an existing core.

This will create a new broker object that references the existing broker. The new broker object must be freed as well.

Parameters

- **core** – An existing HelicsCore.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

A new reference to the same broker.

HelicsBool **helicsCoreIsValid**(HelicsCore core)

Check if a core object is a valid object.

Parameters

core – The HelicsCore object to test.

HelicsBool **helicsCoreWaitForDisconnect**(HelicsCore core, int msToWait, HelicsError *err)

Wait for the core to disconnect.

Parameters

- **core** – The core to wait for.
- **msToWait** – The time out in millisecond (<0 for infinite timeout).
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

HELICS_TRUE if the disconnect was successful, HELICS_FALSE if there was a timeout.

HelicsBool **helicsCoreIsConnected**(HelicsCore core)

Check if a core is connected.

A connected core implies it is attached to federates or federates could be attached to it

Returns

HELICS_FALSE if not connected, HELICS_TRUE if it is connected.

void **helicsCoreDataLink**(HelicsCore core, const char *source, const char *target, HelicsError *err)

Link a named publication and named input using a core.

Parameters

- **core** – The core to generate the connection from.
- **source** – The name of the publication (cannot be NULL).
- **target** – The name of the target to send the publication data (cannot be NULL).
- **err** – **[inout]** A HelicsError object, can be NULL if the errors are to be ignored.

void **helicsCoreAddSourceFilterToEndpoint**(HelicsCore core, const char *filter, const char *endpoint, HelicsError *err)

Link a named filter to a source endpoint.

Parameters

- **core** – The core to generate the connection from.
- **filter** – The name of the filter (cannot be NULL).
- **endpoint** – The name of the endpoint to filter the data from (cannot be NULL).
- **err** – **[inout]** A HelicsError object, can be NULL if the errors are to be ignored.

void **helicsCoreAddDestinationFilterToEndpoint**(HelicsCore core, const char *filter, const char *endpoint, HelicsError *err)

Link a named filter to a destination endpoint.

Parameters

- **core** – The core to generate the connection from.
- **filter** – The name of the filter (cannot be NULL).
- **endpoint** – The name of the endpoint to filter the data going to (cannot be NULL).
- **err** – **[inout]** A HelicsError object, can be NULL if the errors are to be ignored.

void **helicsCoreMakeConnections**(HelicsCore core, const char *file, HelicsError *err)

Load a file containing connection information.

Parameters

- **core** – The core to generate the connections from.
- **file** – A JSON or TOML file containing connection information.
- **err** – **[inout]** A HelicsError object, can be NULL if the errors are to be ignored.

const char ***helicsCoreGetIdentifier**(HelicsCore core)

Get an identifier for the core.

Parameters

core – The core to query.

Returns

A string with the identifier of the core.

const char ***helicsCoreGetAddress**(HelicsCore core)

Get the network address associated with a core.

Parameters

core – The core to query.

Returns

A string with the network address of the broker.

void **helicsCoreSetReadyToInit**(HelicsCore core, HelicsError *err)

Set the core to ready for init.

This function is used for cores that have filters but no federates so there needs to be a direct signal to the core to trigger the federation initialization.

Parameters

- **core** – The core object to enable init values for.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

HelicsBool **helicsCoreConnect**(HelicsCore core, HelicsError *err)

Connect a core to the federate based on current configuration.

Parameters

- **core** – The core to connect.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

HELICS_FALSE if not connected, HELICS_TRUE if it is connected.

void **helicsCoreDisconnect**(HelicsCore core, HelicsError *err)

Disconnect a core from the federation.

Parameters

- **core** – The core to query.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsCoreDestroy**(HelicsCore core)

Disconnect and free a core.

void **helicsCoreFree**(HelicsCore core)

Release the memory associated with a core.

void **helicsCoreSetGlobal**(HelicsCore core, const char *valueName, const char *value, HelicsError *err)

Set a global value in a core.

This overwrites any previous value for this name.

Parameters

- **core** – The core to set the global through.
- **valueName** – The name of the global to set.
- **value** – The value of the global.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsCoreAddAlias**(HelicsCore core, const char *interfaceName, const char *alias, HelicsError *err)

Create an alias for an interface.

Parameters

- **core** – The core to use to set the alias.
- **interfaceName** – The current name of an interface.
- **alias** – The additional name to use for the given interface.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsCoreSendCommand**(HelicsCore core, const char *target, const char *command, HelicsError *err)

Send a command to another helics object though a core using asynchronous(fast) operations.

Parameters

- **core** – The core to send the command through.
- **target** – The name of the object to send the command to.
- **command** – The command to send.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsCoreSendOrderedCommand**(HelicsCore core, const char *target, const char *command, HelicsError *err)

Send a command to another helics object though a core using ordered operations.

Parameters

- **core** – The core to send the command through.
- **target** – The name of the object to send the command to.
- **command** – The command to send.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsCoreSetLogFile**(HelicsCore core, const char *logFileName, HelicsError *err)

Set the log file on a core.

Parameters

- **core** – The core to set the log file for.
- **logFileName** – The name of the file to log to.

- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsCoreGlobalError**(HelicsCore core, int errorCode, const char *errorString, HelicsError *err)

Generate a global error through a broker. This will terminate the federation.

Parameters

- **core** – The core to generate the global error.
- **errorCode** – The error code to associate with the global error.
- **errorString** – An error message to associate with the global error.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsCoreSetLoggingCallback**(HelicsCore core, void (*logger)(int loglevel, const char *identifier, const char *message, void *userData), void *userdata, HelicsError *err)

Set the logging callback for a core.

Add a logging callback function to a core. The logging callback will be called when a message flows into a core from the core or from a broker.

Parameters

- **core** – The core object in which to set the callback.
- **logger** – A callback with signature void(int, const char *, const char *, void *); The function arguments are loglevel, an identifier, a message string, and a pointer to user data.
- **userdata** – A pointer to user data that is passed to the function when executing.
- **err** – **[inout]** A pointer to an error object for catching errors.

HelicsFilter **helicsCoreRegisterFilter**(HelicsCore core, HelicsFilterTypes type, const char *name, HelicsError *err)

Create a source Filter on the specified core.

Filters can be created through a federate or a core, linking through a federate allows a few extra features of name matching to function on the federate interface but otherwise equivalent behavior.

Parameters

- **core** – The core to register through.
- **type** – The type of filter to create /ref HelicsFilterTypes.
- **name** – The name of the filter (can be NULL).
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns

A HelicsFilter object.

HelicsFilter **helicsCoreRegisterCloningFilter**(HelicsCore core, const char *name, HelicsError *err)

Create a cloning Filter on the specified core.

Cloning filters copy a message and send it to multiple locations, source and destination can be added through other functions.

Parameters

- **core** – The core to register through.
- **name** – The name of the filter (can be NULL).

- **err** – **[inout]** A pointer to an error object for catching errors.

Returns
A HelicsFilter object.

HelicsTranslator **helicsCoreRegisterTranslator**(HelicsCore core, HelicsTranslatorTypes type, const char *name, HelicsError *err)

Create a Translator on the specified core.

Translators can be created through a federate or a core. Linking through a federate allows a few extra features of name matching to function on the federate interface but otherwise have equivalent behavior.

Parameters

- **core** – The core to register through.
- **type** – The type of translator to create /ref HelicsTranslatorTypes.
- **name** – The name of the translator (can be NULL).
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns
A HelicsTranslator object.

App

HelicsBool **helicsAppEnabled**()

HELICS_TRUE if the app library is enabled

HelicsFederate **helicsAppGetFederate**(HelicsApp app, HelicsError *err)

Run the HelicsApp.

Execute the HelicsApp to completion

Parameters

- **app** – The app to execute.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns
An opaque value federate object that can be used in any of the federate methods, not recommended to use this object to advance time, the app will not likely function normally; other query, information calls, or modification calls on the federate are fine.

void **helicsAppLoadFile**(HelicsApp app, const char *configFile, HelicsError *err)

Load a file to an App.

Loads a configuration file for an app.

Parameters

- **app** – The app to load a file.
- **configFile** – Configuration file or string to pass into the app, can be NULL or empty.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsAppInitialize**(HelicsApp app, HelicsError *err)

Initialize the HelicsApp federate.

Generate all the interfaces and load data for the application.

Parameters

- **app** – The app to initialize.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsAppRun**(HelicsApp app, HelicsError *err)

Run the HelicsApp.

Execute the app to completion.

Parameters

- **app** – The app to execute.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsAppRunTo**(HelicsApp app, HelicsTime stopTime, HelicsError *err)

Run a HelicsApp to a specified stop time.

It is possible to call this method repeatedly with different times.

Parameters

- **app** – The app to run.
- **stopTime** – The desired stop time.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsAppFinalize**(HelicsApp app, HelicsError *err)

Finalize the HelicsApp.

Parameters

- **app** – The app to finalize.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsAppFree**(HelicsApp app)

Free the HelicsApp object.

Parameters

app – The app to free.

void **helicsAppDestroy**(HelicsApp app)

Disconnect and free a HelicsApp.

Parameters

app – The app to destroy.

HelicsBool **helicsAppIsActive**(HelicsApp app)

Check if the HelicsApp is active and ready to run.

Parameters

app – The app to check.

Returns

True if the app is active, otherwise false.

FederateInfo

HelicsFederateInfo **helicsFederateInfoClone**(HelicsFederateInfo fedInfo, HelicsError *err)

Create a federate info object from an existing one and clone the information.

Parameters

- **fedInfo** – A federateInfo object to duplicate.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns
A HelicsFederateInfo object which is a reference to the created object.

void **helicsFederateInfoLoadFromArgs**(HelicsFederateInfo fedInfo, int argc, const char *const *argv, HelicsError *err)

Load federate info from command line arguments.

Parameters

- **fedInfo** – A federateInfo object.
- **argc** – The number of command line arguments.
- **argv** – An array of strings from the command line.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateInfoLoadFromString**(HelicsFederateInfo fedInfo, const char *args, HelicsError *err)

Load federate info from command line arguments contained in a string.

Parameters

- **fedInfo** – A federateInfo object.
- **args** – Command line arguments specified in a string.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateInfoFree**(HelicsFederateInfo fedInfo)

Delete the memory associated with a federate info object.

void **helicsFederateInfoSetCoreName**(HelicsFederateInfo fedInfo, const char *corename, HelicsError *err)

Set the name of the core to link to for a federate.

Parameters

- **fedInfo** – The federate info object to alter.
- **corename** – The identifier for a core to link to.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateInfoSetCoreInitString**(HelicsFederateInfo fedInfo, const char *coreInit, HelicsError *err)

Set the initialization string for the core usually in the form of command line arguments.

Parameters

- **fedInfo** – The federate info object to alter.
- **coreInit** – A string containing command line arguments to be passed to the core.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateInfoSetBrokerInitString**(HelicsFederateInfo fedInfo, const char *brokerInit, HelicsError *err)

Set the initialization string that a core will pass to a generated broker usually in the form of command line arguments.

Parameters

- **fedInfo** – The federate info object to alter.
- **brokerInit** – A string with command line arguments for a generated broker.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateInfoSetCoreType**(HelicsFederateInfo fedInfo, int coretype, HelicsError *err)

Set the core type by integer code.

Valid values available by definitions in api-data.h.

Parameters

- **fedInfo** – The federate info object to alter.
- **coretype** – An numerical code for a core type see /ref helics_CoreType.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateInfoSetCoreTypeFromString**(HelicsFederateInfo fedInfo, const char *coretype, HelicsError *err)

Set the core type from a string.

Parameters

- **fedInfo** – The federate info object to alter.
- **coretype** – A string naming a core type.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateInfoSetBroker**(HelicsFederateInfo fedInfo, const char *broker, HelicsError *err)

Set the name or connection information for a broker.

This is only used if the core is automatically created, the broker information will be transferred to the core for connection.

Parameters

- **fedInfo** – The federate info object to alter.
- **broker** – A string which defines the connection information for a broker either a name or an address.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateInfoSetBrokerKey**(HelicsFederateInfo fedInfo, const char *brokerkey, HelicsError *err)

Set the key for a broker connection.

This is only used if the core is automatically created, the broker information will be transferred to the core for connection.

Parameters

- **fedInfo** – The federate info object to alter.
- **brokerkey** – A string containing a key for the broker to connect.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateInfoSetBrokerPort**(HelicsFederateInfo fedInfo, int brokerPort, HelicsError *err)

Set the port to use for the broker.

This is only used if the core is automatically created, the broker information will be transferred to the core for connection. This will only be useful for network broker connections.

Parameters

- **fedInfo** – The federate info object to alter.
- **brokerPort** – The integer port number to use for connection with a broker.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateInfoSetLocalPort**(HelicsFederateInfo fedInfo, const char *localPort, HelicsError *err)

Set the local port to use.

This is only used if the core is automatically created, the port information will be transferred to the core for connection.

Parameters

- **fedInfo** – The federate info object to alter.
- **localPort** – A string with the port information to use as the local server port can be a number or "auto" or "os_local".
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateInfoSetFlagOption**(HelicsFederateInfo fedInfo, int flag, HelicsBool value, HelicsError *err)

Set a flag in the info structure.

Valid flags are available /ref helics_federate_flags.

Parameters

- **fedInfo** – The federate info object to alter.
- **flag** – A numerical index for a flag.
- **value** – The desired value of the flag HELICS_TRUE or HELICS_FALSE.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateInfoSetSeparator**(HelicsFederateInfo fedInfo, char separator, HelicsError *err)

Set the separator character in the info structure.

The separator character is the separation character for local publications/endpoints in creating their global name. For example if the separator character is '/' then a local endpoint would have a globally reachable name of fedName/localName.

Parameters

- **fedInfo** – The federate info object to alter.
- **separator** – The character to use as a separator.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateInfoSetTimeProperty**(HelicsFederateInfo fedInfo, int timeProperty, HelicsTime propertyValue, HelicsError *err)

Set the output delay for a federate.

Parameters

- **fedInfo** – The federate info object to alter.
- **timeProperty** – An integer representation of the time based property to set see /ref helics_properties.
- **propertyValue** – The value of the property to set the timeProperty to.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateInfoSetIntegerProperty**(HelicsFederateInfo fedInfo, int intProperty, int propertyValue, HelicsError *err)

Set an integer property for a federate.

Set known properties.

Parameters

- **fedInfo** – The federateInfo object to alter.
- **intProperty** – An int identifying the property.
- **propertyValue** – The value to set the property to.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Federate

void **helicsFederateDestroy**(HelicsFederate fed)

Disconnect and free a federate.

HelicsFederate **helicsFederateClone**(HelicsFederate fed, HelicsError *err)

Create a new reference to an existing federate.

This will create a new HelicsFederate object that references the existing federate. The new object must be freed as well.

Parameters

- **fed** – An existing HelicsFederate.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

A new reference to the same federate.

void **helicsFederateProtect**(const char *fedName, HelicsError *err)

Protect a federate from finalizing and closing if all references go out of scope

this function allows a federate to be retrieved on demand, it must be explicitly close later otherwise it will be destroyed when the library is closed

Parameters

- **fedName** – The name of an existing HelicsFederate.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function, in particular if no federate with the given name exists

void **helicsFederateUnProtect**(const char *fedName, HelicsError *err)

remove the protection of an existing federate

this function allows a federate to be retrieved on demand, it must be explicitly close later otherwise it will be destroyed when the library is closed

Parameters

- **fedName** – the name of an existing federate that should not be protected
- **err** – **[inout]** An error object that will contain an error code and string if the federate was not found.

HelicsBool **helicsFederateIsProtected**(const char *fedName, HelicsError *err)

checks if an existing federate is protected

Parameters

- **fedName** – the name of an existing federate to check the protection status
- **err** – **[inout]** An error object that will contain an error code and string if the federate was not found.

HelicsBool **helicsFederateIsValid**(HelicsFederate fed)

Check if a federate_object is valid.

Returns

HELICS_TRUE if the federate is a valid active federate, HELICS_FALSE otherwise

void **helicsFederateRegisterInterfaces**(HelicsFederate fed, const char *file, HelicsError *err)

Load interfaces from a file.

Parameters

- **fed** – The federate to which to load interfaces.
- **file** – The name of a file to load the interfaces from either JSON, or TOML.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateGlobalError**(HelicsFederate fed, int errorCode, const char *errorString, HelicsError *err)

Generate a global error from a federate.

A global error halts the co-simulation completely.

Parameters

- **fed** – The federate to create an error in.
- **errorCode** – The integer code for the error.
- **errorString** – A string describing the error.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateLocalError**(HelicsFederate fed, int errorCode, const char *errorString, HelicsError *err)

Generate a local error in a federate.

This will propagate through the co-simulation but not necessarily halt the co-simulation, it has a similar effect to finalize but does allow some interaction with a core for a brief time.

Parameters

- **fed** – The federate to create an error in.
- **errorCode** – The integer code for the error.
- **errorString** – A string describing the error.

- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateFinalize**(HelicsFederate fed, HelicsError *err)

Disconnect/finalize the federate. This function halts all communication in the federate and disconnects it from the core.

void **helicsFederateFinalizeAsync**(HelicsFederate fed, HelicsError *err)

Disconnect/finalize the federate in an async call.

void **helicsFederateFinalizeComplete**(HelicsFederate fed, HelicsError *err)

Complete the asynchronous disconnect/finalize call.

void **helicsFederateDisconnect**(HelicsFederate fed, HelicsError *err)

Disconnect/finalize the federate. This function halts all communication in the federate and disconnects it from the core. This call is identical to helicsFederateFinalize.

void **helicsFederateDisconnectAsync**(HelicsFederate fed, HelicsError *err)

Disconnect/finalize the federate in an async call. This call is identical to helicsFederateFinalizeAsync.

void **helicsFederateDisconnectComplete**(HelicsFederate fed, HelicsError *err)

Complete the asynchronous disconnect/finalize call. This call is identical to helicsFederateFinalizeComplete

void **helicsFederateFree**(HelicsFederate fed)

Release the memory associated with a federate.

void **helicsFederateEnterInitializingMode**(HelicsFederate fed, HelicsError *err)

Enter the initialization state of a federate.

The initialization state allows initial values to be set and received if the iteration is requested on entry to the execution state. This is a blocking call and will block until the core allows it to proceed.

Parameters

- **fed** – The federate to operate on.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateEnterInitializingModeAsync**(HelicsFederate fed, HelicsError *err)

Non blocking alternative to helicsFederateEnterInitializingMode.

The function helicsFederateEnterInitializationModeComplete must be called to finish the operation.

Parameters

- **fed** – The federate to operate on.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateEnterInitializingModeComplete**(HelicsFederate fed, HelicsError *err)

Complete the entry to initialize mode that was initiated with /ref heliceEnterInitializingModeAsync.

Parameters

- **fed** – The federate desiring to complete the initialization step.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateEnterInitializingModeIterative**(HelicsFederate fed, HelicsError *err)

Trigger a blocking call and return to created state after all federates have either triggered an iteration or are waiting to enter initializing mode.

This call will return the federate to the created state to allow additional setup to occur with federates either iterating in the mode or waiting.

Parameters

- **fed** – The federate to operate on.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateEnterInitializingModeIterativeAsync**(HelicsFederate fed, HelicsError *err)

Non blocking alternative to helicsFederateEnterInitializingModeIterative.

The function helicsFederateEnterInitializationModeIterativeComplete must be called to finish the operation.

Parameters

- **fed** – The federate to operate on.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateEnterInitializingModeIterativeComplete**(HelicsFederate fed, HelicsError *err)

Complete the call to enter initializing mode Iterative that was initiated with /ref heliceEnterInitializingModeIterativeAsync. The federate will be in created or error mode on return

Parameters

- **fed** – The federate used in the corresponding async call
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

HelicsBool **helicsFederateIsAsyncOperationCompleted**(HelicsFederate fed, HelicsError *err)

Check if the current Asynchronous operation has completed.

Parameters

- **fed** – The federate to operate on.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

HELICS_FALSE if not completed, HELICS_TRUE if completed.

void **helicsFederateEnterExecutingMode**(HelicsFederate fed, HelicsError *err)

Request that the federate enter the Execution mode.

This call is blocking until granted entry by the core object. On return from this call the federate will be at time 0. For an asynchronous alternative call see /ref helicsFederateEnterExecutingModeAsync.

Parameters

- **fed** – A federate to change modes.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateEnterExecutingModeAsync**(HelicsFederate fed, HelicsError *err)

Request that the federate enter the Execution mode.

This call is non-blocking and will return immediately. Call /ref helicsFederateEnterExecutingModeComplete to finish the call sequence.

Parameters

- **fed** – The federate object to complete the call.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateEnterExecutingModeComplete**(HelicsFederate fed, HelicsError *err)

Complete the call to /ref helicsFederateEnterExecutingModeAsync.

Parameters

- **fed** – The federate object to complete the call.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

HelicsIterationResult **helicsFederateEnterExecutingModeIterative**(HelicsFederate fed,
HelicsIterationRequest iterate,
HelicsError *err)

Request an iterative time.

This call allows for finer grain control of the iterative process than /ref helicsFederateRequestTime. It takes a time and iteration request, and returns a time and iteration status.

Parameters

- **fed** – The federate to make the request of.
- **iterate** – The requested iteration mode.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

An iteration structure with field containing the time and iteration status.

void **helicsFederateEnterExecutingModeIterativeAsync**(HelicsFederate fed, HelicsIterationRequest iterate,
HelicsError *err)

Request an iterative entry to the execution mode.

This call allows for finer grain control of the iterative process than /ref helicsFederateRequestTime. It takes a time and iteration request, and returns a time and iteration status

Parameters

- **fed** – The federate to make the request of.
- **iterate** – The requested iteration mode.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

HelicsIterationResult **helicsFederateEnterExecutingModeIterativeComplete**(HelicsFederate fed,
HelicsError *err)

Complete the asynchronous iterative call into ExecutionMode.

Parameters

- **fed** – The federate to make the request of.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

An iteration object containing the iteration time and iteration_status.

HelicsFederateState **helicsFederateGetState**(HelicsFederate fed, HelicsError *err)

Get the current state of a federate.

Parameters

- **fed** – The federate to query.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function. The err object will be removed in a future release as it is not necessary for use the function will not error, invalid federate return HELICS_STATE_UNKOWN

Returns

State the resulting state if the federate is invalid will return HELICS_STATE_UNKNOWN

HelicsCore **helicsFederateGetCore**(HelicsFederate fed, HelicsError *err)

Get the core object associated with a federate.

Parameters

- **fed** – A federate object.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

A core object, nullptr if invalid.

HelicsTime **helicsFederateRequestTime**(HelicsFederate fed, HelicsTime requestTime, HelicsError *err)

Request the next time for federate execution.

Parameters

- **fed** – The federate to make the request of.
- **requestTime** – The next requested time.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

The time granted to the federate, will return HELICS_TIME_MAXTIME if the simulation has terminated or is invalid.

HelicsTime **helicsFederateRequestTimeAdvance**(HelicsFederate fed, HelicsTime timeDelta, HelicsError *err)

Request the next time for federate execution.

Parameters

- **fed** – The federate to make the request of.
- **timeDelta** – The requested amount of time to advance.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

The time granted to the federate, will return HELICS_TIME_MAXTIME if the simulation has terminated or is invalid

HelicsTime **helicsFederateRequestNextStep**(HelicsFederate fed, HelicsError *err)

Request the next time step for federate execution.

Feds should have setup the period or minDelta for this to work well but it will request the next time step which is the current time plus the minimum time step.

Parameters

- **fed** – The federate to make the request of.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

The time granted to the federate, will return HELICS_TIME_MAXTIME if the simulation has terminated or is invalid

HelicsTime **helicsFederateRequestTimeIterative**(HelicsFederate fed, HelicsTime requestTime, HelicsIterationRequest iterate, HelicsIterationResult *outIteration, HelicsError *err)

Request an iterative time.

This call allows for finer grain control of the iterative process than /ref helicsFederateRequestTime. It takes a time and and iteration request, and returns a time and iteration status.

Parameters

- **fed** – The federate to make the request of.
- **requestTime** – The next desired time.
- **iterate** – The requested iteration mode.
- **outIteration** – **[out]** The iteration specification of the result.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

The granted time, will return HELICS_TIME_MAXTIME if the simulation has terminated along with the appropriate iteration result.

void **helicsFederateRequestTimeAsync**(HelicsFederate fed, HelicsTime requestTime, HelicsError *err)

Request the next time for federate execution in an asynchronous call.

Call /ref helicsFederateRequestTimeComplete to finish the call.

Parameters

- **fed** – The federate to make the request of.
- **requestTime** – The next requested time.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

HelicsTime **helicsFederateRequestTimeComplete**(HelicsFederate fed, HelicsError *err)

Complete an asynchronous requestTime call.

Parameters

- **fed** – The federate to make the request of.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

The time granted to the federate, will return HELICS_TIME_MAXTIME if the simulation has terminated.

void **helicsFederateRequestTimeIterativeAsync**(HelicsFederate fed, HelicsTime requestTime, HelicsIterationRequest iterate, HelicsError *err)

Request an iterative time through an asynchronous call.

This call allows for finer grain control of the iterative process than /ref helicsFederateRequestTime. It takes a time and iteration request, and returns a time and iteration status. Call /ref helicsFederateRequestTimeIterativeComplete to finish the process.

Parameters

- **fed** – The federate to make the request of.
- **requestTime** – The next desired time.
- **iterate** – The requested iteration mode.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

HelicsTime **helicsFederateRequestTimeIterativeComplete**(HelicsFederate fed, HelicsIterationResult *outIterate, HelicsError *err)

Complete an iterative time request asynchronous call.

Parameters

- **fed** – The federate to make the request of.
- **outIterate** – **[out]** The iteration specification of the result.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

The granted time, will return HELICS_TIME_MAXTIME if the simulation has terminated.

void **helicsFederateProcessCommunications**(HelicsFederate fed, HelicsTime period, HelicsError *err)

Tell helics to process internal communications for a period of time.

Parameters

- **fed** – The federate to tell to process.
- **period** – The length of time to process communications and then return control.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

const char ***helicsFederateGetName**(HelicsFederate fed)

Get the name of the federate.

Parameters

fed – The federate object to query.

Returns

A pointer to a string with the name.

void **helicsFederateSetTimeProperty**(HelicsFederate fed, int timeProperty, HelicsTime time, HelicsError *err)

Set a time based property for a federate.

Parameters

- **fed** – The federate object to set the property for.
- **timeProperty** – A integer code for a time property.
- **time** – The requested value of the property.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateSetFlagOption**(HelicsFederate fed, int flag, HelicsBool flagValue, HelicsError *err)

Set a flag for the federate.

Parameters

- **fed** – The federate to alter a flag for.
- **flag** – The flag to change.
- **flagValue** – The new value of the flag. 0 for false, !=0 for true.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateSetSeparator**(HelicsFederate fed, char separator, HelicsError *err)

Set the separator character in a federate.

The separator character is the separation character for local publications/endpoints in creating their global name. For example if the separator character is '/' then a local endpoint would have a globally reachable name of fedName/localName.

Parameters

- **fed** – The federate info object to alter.
- **separator** – The character to use as a separator.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsFederateSetIntegerProperty**(HelicsFederate fed, int intProperty, int propertyVal, HelicsError *err)

Set an integer based property of a federate.

Parameters

- **fed** – The federate to change the property for.
- **intProperty** – The property to set.
- **propertyVal** – The value of the property.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

HelicsTime **helicsFederateGetTimeProperty**(HelicsFederate fed, int timeProperty, HelicsError *err)

Get the current value of a time based property in a federate.

Parameters

- **fed** – The federate query.
- **timeProperty** – The property to query.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

HelicsBool **helicsFederateGetFlagOption**(HelicsFederate fed, int flag, HelicsError *err)

Get a flag value for a federate.

Parameters

- **fed** – The federate to get the flag for.
- **flag** – The flag to query.
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns
The value of the flag.

int **helicsFederateGetIntegerProperty**(HelicsFederate fed, int intProperty, HelicsError *err)

Get the current value of an integer property (such as a logging level).

Parameters

- **fed** – The federate to get the flag for.
- **intProperty** – A code for the property to set /ref helics_handle_options.
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns
The value of the property.

HelicsTime **helicsFederateGetCurrentTime**(HelicsFederate fed, HelicsError *err)

Get the current time of the federate.

Parameters

- **fed** – The federate object to query.
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns
The current time of the federate.

void **helicsFederateAddAlias**(HelicsFederate fed, const char *interfaceName, const char *alias, HelicsError *err)

Create an alias for an interface.

Parameters

- **fed** – The federate to use to set the alias.
- **interfaceName** – The current name of an interface.
- **alias** – The additional name to use for the given interface.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsFederateSetGlobal**(HelicsFederate fed, const char *valueName, const char *value, HelicsError *err)

Set a federation global value through a federate.

This overwrites any previous value for this name.

Parameters

- **fed** – The federate to set the global through.
- **valueName** – The name of the global to set.
- **value** – The value of the global.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsFederateSetTag**(HelicsFederate fed, const char *tagName, const char *value, HelicsError *err)

Set a federate tag value.

This overwrites any previous value for this tag.

Parameters

- **fed** – The federate to set the tag for.
- **tagName** – The name of the tag to set.
- **value** – The value of the tag.

- **err** – **[inout]** A pointer to an error object for catching errors.

const char ***helicsFederateGetTag**(HelicsFederate fed, const char *tagName, HelicsError *err)

Get a federate tag value.

Parameters

- **fed** – The federate to get the tag for.
- **tagName** – The name of the tag to query.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsFederateAddDependency**(HelicsFederate fed, const char *fedName, HelicsError *err)

Add a time dependency for a federate. The federate will depend on the given named federate for time synchronization.

Parameters

- **fed** – The federate to add the dependency for.
- **fedName** – The name of the federate to depend on.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsFederateSetLogFile**(HelicsFederate fed, const char *logFile, HelicsError *err)

Set the logging file for a federate (actually on the core associated with a federate).

Parameters

- **fed** – The federate to set the log file for.
- **logFile** – The name of the log file.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsFederateLogErrorMessage**(HelicsFederate fed, const char *logmessage, HelicsError *err)

Log an error message through a federate.

Parameters

- **fed** – The federate to log the error message through.
- **logmessage** – The message to put in the log.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsFederateLogWarningMessage**(HelicsFederate fed, const char *logmessage, HelicsError *err)

Log a warning message through a federate.

Parameters

- **fed** – The federate to log the warning message through.
- **logmessage** – The message to put in the log.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsFederateLogInfoMessage**(HelicsFederate fed, const char *logmessage, HelicsError *err)

Log an info message through a federate.

Parameters

- **fed** – The federate to log the info message through.
- **logmessage** – The message to put in the log.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsFederateLogDebugMessage**(HelicsFederate fed, const char *logmessage, HelicsError *err)

Log a debug message through a federate.

Parameters

- **fed** – The federate to log the debug message through.
- **logmessage** – The message to put in the log.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsFederateLogLevelMessage**(HelicsFederate fed, int loglevel, const char *logmessage, HelicsError *err)

Log a message through a federate.

Parameters

- **fed** – The federate to log the message through.
- **loglevel** – The level of the message to log see /ref helics_log_levels.
- **logmessage** – The message to put in the log.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsFederateSendCommand**(HelicsFederate fed, const char *target, const char *command, HelicsError *err)

Send a command to another helics object through a federate.

Parameters

- **fed** – The federate to send the command through.
- **target** – The name of the object to send the command to.
- **command** – The command to send.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

const char ***helicsFederateGetCommand**(HelicsFederate fed, HelicsError *err)

Get a command sent to the federate.

Parameters

- **fed** – The federate to get the command for.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

A string with the command for the federate, if the string is empty no command is available.

const char ***helicsFederateGetCommandSource**(HelicsFederate fed, HelicsError *err)

Get the source of the most recently retrieved command sent to the federate.

Parameters

- **fed** – The federate to get the command for.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

A string with the command for the federate, if the string is empty no command is available.

const char *helicsFederateWaitCommand(HelicsFederate fed, HelicsError *err)

Get a command sent to the federate. Blocks until a command is received.

Parameters

- **fed** – The federate to get the command for.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

A string with the command for the federate, if the string is empty no command is available.

void helicsFederateSetLoggingCallback(HelicsFederate fed, void (*logger)(int loglevel, const char *identifier, const char *message, void *userData), void *userdata, HelicsError *err)

Set the logging callback for a federate.

Add a logging callback function to a federate. The logging callback will be called when a message flows into a federate from the core or from a federate.

Parameters

- **fed** – The federate object in which to create a subscription must have been created with helicsCreateValueFederate or helicsCreateCombinationFederate.
- **logger** – A callback with signature void(int, const char *, const char *, void *); The function arguments are loglevel, an identifier string, a message string, and a pointer to user data.
- **userdata** – A pointer to user data that is passed to the function when executing.
- **err** – **[inout]** A pointer to an error object for catching errors.

void helicsFederateSetQueryCallback(HelicsFederate fed, void (*queryAnswer)(const char *query, int querySize, HelicsQueryBuffer buffer, void *userdata), void *userdata, HelicsError *err)

Set callback for queries executed against a federate.

There are many queries that HELICS understands directly, but it is occasionally useful to have a federate be able to respond to specific queries with answers specific to a federate.

Parameters

- **fed** – The federate to set the callback for.
- **queryAnswer** – A callback with signature const char *(const char *query, int querySize, HelicsQueryBuffer buffer, void *userdata); The function arguments include the query string requesting an answer along with its size; the string is not guaranteed to be null terminated. HelicsQueryBuffer is the buffer intended to filled out by the userCallback. The buffer can be empty if the query is not recognized and HELICS will generate the appropriate response. The buffer is used to ensure memory ownership separation between user code and HELICS code. The HelicsQueryBufferFill method can be used to load a string into the buffer.
- **userdata** – A pointer to user data that is passed to the function when executing.
- **err** – **[inout]** A pointer to an error object for catching errors.

void helicsFederateSetTimeRequestEntryCallback(HelicsFederate fed, void (*requestTime)(HelicsTime currentTime, HelicsTime requestTime, HelicsBool iterating, void *userdata), void *userdata, HelicsError *err)

Set callback for the time request.

This callback will be executed when a valid time request is made. It is intended for the possibility of embedded data grabbers in a callback to simplify user code.

Parameters

- **fed** – The federate to set the callback for.
- **requestTime** – A callback with signature void(HelicsTime currentTime, HelicsTime requestTime, bool iterating, void *userdata); The function arguments are the current time value, the requested time value, a bool indicating that the time is iterating, and pointer to the userdata.
- **userdata** – A pointer to user data that is passed to the function when executing.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsFederateSetTimeUpdateCallback**(HelicsFederate fed, void (*timeUpdate)(HelicsTime newTime, HelicsBool iterating, void *userdata), void *userdata, HelicsError *err)

Set callback for the time update.

This callback will be executed every time the simulation time is updated starting on entry to executing mode.

Parameters

- **fed** – The federate to set the callback for.
- **timeUpdate** – A callback with signature void(HelicsTime newTime, bool iterating, void *userdata); The function arguments are the new time value, a bool indicating that the time is iterating, and pointer to the userdata.
- **userdata** – A pointer to user data that is passed to the function when executing.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsFederateSetStateChangeCallback**(HelicsFederate fed, void (*stateChange)(HelicsFederateState newState, HelicsFederateState oldState, void *userdata), void *userdata, HelicsError *err)

Set callback for the federate mode change.

This callback will be executed every time the operating mode of the federate changes.

Parameters

- **fed** – The federate to set the callback for.
- **stateChange** – A callback with signature void(HelicsFederateState newState, HelicsFederateState oldState, void *userdata); The function arguments are the new state, the old state, and pointer to the userdata.
- **userdata** – A pointer to user data that is passed to the function when executing.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsFederateSetTimeRequestReturnCallback**(HelicsFederate fed, void (*requestTimeReturn)(HelicsTime newTime, HelicsBool iterating, void *userdata), void *userdata, HelicsError *err)

Set callback for the time request return.

This callback will be executed after all other callbacks for a time request return. This callback will be the last thing executed before returning control to the user program. The difference between this and the TimeUpdate

callback is the order of execution. The timeUpdate callback is executed prior to individual interface callbacks, this callback is executed after all others.

Parameters

- **fed** – The federate to set the callback for.
- **requestTimeReturn** – A callback with signature void(HelicsTime newTime, bool iterating, void *userdata); The function arguments are the new time value, a bool indicating that the time is iterating, and pointer to the userdata.
- **userdata** – A pointer to user data that is passed to the function when executing.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsFederateInitializingEntryCallback**(HelicsFederate fed, void (*initializingEntry)(HelicsBool iterating, void *userdata), void *userdata, HelicsError *err)

Set callback for the entry to initializingMode.

This callback will be executed when the initializingMode is entered

Parameters

- **fed** – The federate to set the callback for.
- **initializingEntry** – A callback with signature void(HelicsBool iterating, void *userdata); the bool parameter is set to true if the entry is iterative, therefore the first time this is called the bool is false and all subsequent times it is true.
- **userdata** – A pointer to user data that is passed to the function when executing.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsFederateExecutingEntryCallback**(HelicsFederate fed, void (*executingEntry)(void *userdata), void *userdata, HelicsError *err)

Set callback for the entry to ExecutingMode.

This callback will be executed once on first entry to executingMode

Parameters

- **fed** – The federate to set the callback for.
- **executingEntry** – A callback with signature void(void *userdata);
- **userdata** – A pointer to user data that is passed to the function when executing.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsFederateCosimulationTerminationCallback**(HelicsFederate fed, void (*cosimTermination)(void *userdata), void *userdata, HelicsError *err)

Set callback for cosimulation termination.

This callback will be executed once when the time advancement of the federate/co-simulation has terminated. This may be called as part of the finalize operation, or when a maxTime signal is returned from requestTime or when an error is encountered.

Parameters

- **fed** – The federate to set the callback for.
- **cosimTermination** – A callback with signature void(void *userdata);
- **userdata** – A pointer to user data that is passed to the function when executing.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsFederateErrorHandlerCallback**(HelicsFederate fed, void (*errorHandler)(int errorCode, const char *errorString, void *userdata), void *userdata, HelicsError *err)

Set callback for error handling.

This callback will be called when a federate error is encountered.

Parameters

- **fed** – The federate to set the callback for.
- **errorHandler** – A callback with signature void(int errorCode, const char *errorString, void *userdata);
- **userdata** – A pointer to user data that is passed to the function when executing.
- **err** – **[inout]** A pointer to an error object for catching errors.

HelicsTranslator **helicsFederateRegisterTranslator**(HelicsFederate fed, HelicsTranslatorTypes type, const char *name, HelicsError *err)

Create a Translator on the specified federate.

Translators can be created through a federate or a core. Linking through a federate allows a few extra features of name matching to function on the federate interface but otherwise have equivalent behavior.

Parameters

- **fed** – The federate to register through.
- **type** – The type of translator to create /ref HelicsTranslatorTypes.
- **name** – The name of the translator (can be NULL).
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns

A HelicsTranslator object.

HelicsTranslator **helicsFederateRegisterGlobalTranslator**(HelicsFederate fed, HelicsTranslatorTypes type, const char *name, HelicsError *err)

Create a global translator through a federate.

Translators can be created through a federate or a core. Linking through a federate allows a few extra features of name matching to function on the federate interface but otherwise have equivalent behavior.

Parameters

- **fed** – The federate to register through.
- **type** – The type of translator to create /ref HelicsTranslatorTypes.
- **name** – The name of the translator (can be NULL).
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns

A HelicsTranslator object.

int **helicsFederateGetTranslatorCount**(HelicsFederate fed)

Get the number of translators registered through a federate.

Parameters

fed – The federate object to use to get the translator.

Returns

A count of the number of translators registered through a federate.

HelicsTranslator **helicsFederateGetTranslator**(HelicsFederate fed, const char *name, HelicsError *err)

Get a translator by its name, typically already created via registerInterfaces file or something of that nature.

Parameters

- **fed** – The federate object to use to get the translator.
- **name** – The name of the translator.
- **err** – **[inout]** The error object to complete if there is an error.

Returns

A HelicsTranslator object. If no translator with the specified name exists, the object will not be valid and err will contain an error code.

HelicsTranslator **helicsFederateGetTranslatorByIndex**(HelicsFederate fed, int index, HelicsError *err)

Get a translator by its index, typically already created via registerInterfaces file or something of that nature.

Parameters

- **fed** – The federate object in which to create a publication.
- **index** – The index of the translator to get.
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns

A HelicsTranslator, which will be NULL if an invalid index is given.

ValueFederate

HelicsInput **helicsFederateRegisterSubscription**(HelicsFederate fed, const char *key, const char *units, HelicsError *err)

input/publication registration Create an input and add a publication target.

this method is a wrapper method to create and unnamed input and add a publication target to it

Parameters

- **fed** – The federate object in which to create an input, must have been created with /ref helicsCreateValueFederate or /ref helicsCreateCombinationFederate.
- **key** – The identifier matching a publication to add as an input target.
- **units** – A string listing the units of the input (may be NULL).
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns

An object containing the input.

HelicsPublication **helicsFederateRegisterPublication**(HelicsFederate fed, const char *key, HelicsDataTypes type, const char *units, HelicsError *err)

Register a publication with a known type.

The publication becomes part of the federate and is destroyed when the federate is freed so there are no separate free functions for inputs and publications.

Parameters

- **fed** – The federate object in which to create a publication.
- **key** – The identifier for the publication the global publication key will be prepended with the federate name (may be NULL).
- **type** – A code identifying the type of the input see /ref HelicsDataTypes for available options.

- **units** – A string listing the units of the publication (may be NULL).
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns
An object containing the publication.

HelicsPublication **helicsFederateRegisterTypePublication**(HelicsFederate fed, const char *key, const char *type, const char *units, HelicsError *err)

Register a publication with a defined type.

The publication becomes part of the federate and is destroyed when the federate is freed so there are no separate free functions for inputs and publications.

Parameters

- **fed** – The federate object in which to create a publication.
- **key** – The identifier for the publication (may be NULL).
- **type** – A string labeling the type of the publication (may be NULL).
- **units** – A string listing the units of the publication (may be NULL).
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns
An object containing the publication.

HelicsPublication **helicsFederateRegisterGlobalPublication**(HelicsFederate fed, const char *key, HelicsDataTypes type, const char *units, HelicsError *err)

Register a global named publication with an arbitrary type.

The publication becomes part of the federate and is destroyed when the federate is freed so there are no separate free functions for inputs and publications.

Parameters

- **fed** – The federate object in which to create a publication.
- **key** – The identifier for the publication (may be NULL).
- **type** – A code identifying the type of the input see /ref HelicsDataTypes for available options.
- **units** – A string listing the units of the publication (may be NULL).
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns
An object containing the publication.

HelicsPublication **helicsFederateRegisterGlobalTypePublication**(HelicsFederate fed, const char *key, const char *type, const char *units, HelicsError *err)

Register a global publication with a defined type.

The publication becomes part of the federate and is destroyed when the federate is freed so there are no separate free functions for inputs and publications.

Parameters

- **fed** – The federate object in which to create a publication.
- **key** – The identifier for the publication (may be NULL).

- **type** – A string describing the expected type of the publication (may be NULL).
- **units** – A string listing the units of the publication (may be NULL).
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns
An object containing the publication.

HelicsInput **helicsFederateRegisterInput**(HelicsFederate fed, const char *key, HelicsDataTypes type, const char *units, HelicsError *err)

Register a named input.

The input becomes part of the federate and is destroyed when the federate is freed so there are no separate free functions for inputs, and publications.

Parameters

- **fed** – The federate object in which to create an input.
- **key** – The identifier for the publication the global input key will be prepended with the federate name (may be NULL).
- **type** – A code identifying the type of the input see /ref HelicsDataTypes for available options.
- **units** – A string listing the units of the input (may be NULL).
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns
An object containing the input.

HelicsInput **helicsFederateRegisterTypeInput**(HelicsFederate fed, const char *key, const char *type, const char *units, HelicsError *err)

Register an input with a defined type.

The input becomes part of the federate and is destroyed when the federate is freed so there are no separate free functions for inputs, and publications.

Parameters

- **fed** – The federate object in which to create an input.
- **key** – The identifier for the input (may be NULL).
- **type** – A string describing the expected type of the input (may be NULL).
- **units** – A string listing the units of the input maybe NULL.
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns
An object containing the input.

HelicsInput **helicsFederateRegisterGlobalInput**(HelicsFederate fed, const char *key, HelicsDataTypes type, const char *units, HelicsError *err)

Register a global named input.

The input becomes part of the federate and is destroyed when the federate is freed so there are no separate free functions for inputs and publications.

Parameters

- **fed** – The federate object in which to create an input.
- **key** – The identifier for the input (may be NULL).

- **type** – A code identifying the type of the input see /ref HelicsDataTypes for available options.
- **units** – A string listing the units of the input (may be NULL).
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns
An object containing the input.

HelicsInput **helicsFederateRegisterGlobalTypeInput**(HelicsFederate fed, const char *key, const char *type, const char *units, HelicsError *err)

Register an input with an arbitrary type.

The input becomes part of the federate and is destroyed when the federate is freed so there are no separate free functions for interfaces.

Parameters

- **fed** – The federate object in which to create an input.
- **key** – The identifier for the input (may be NULL).
- **type** – A string defining the type of the input (may be NULL).
- **units** – A string listing the units of the input (may be NULL).
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns
An object containing the input.

HelicsPublication **helicsFederateGetPublication**(HelicsFederate fed, const char *key, HelicsError *err)

Get a publication object from a key.

Parameters

- **fed** – The value federate object to use to get the publication.
- **key** – The name of the publication.
- **err** – **[inout]** The error object to complete if there is an error.

Returns
A HelicsPublication object, the object will not be valid and err will contain an error code if no publication with the specified key exists.

HelicsPublication **helicsFederateGetPublicationByIndex**(HelicsFederate fed, int index, HelicsError *err)

Get a publication by its index, typically already created via registerInterfaces file or something of that nature.

Parameters

- **fed** – The federate object in which to create a publication.
- **index** – The index of the publication to get.
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns
A HelicsPublication.

HelicsInput **helicsFederateGetInput**(HelicsFederate fed, const char *key, HelicsError *err)

Get an input object from a key.

Parameters

- **fed** – The value federate object to use to get the publication.
- **key** – The name of the input.

- **err** – **[inout]** The error object to complete if there is an error.

Returns

A HelicsInput object, the object will not be valid and err will contain an error code if no input with the specified key exists.

HelicsInput **helicsFederateGetInputByIndex**(HelicsFederate fed, int index, HelicsError *err)

Get an input by its index, typically already created via registerInterfaces file or something of that nature.

Parameters

- **fed** – The federate object in which to create a publication.
- **index** – The index of the publication to get.
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns

A HelicsInput, which will be NULL if an invalid index.

HelicsInput **helicsFederateGetInputByTarget**(HelicsFederate fed, const char *target, HelicsError *err)

Get an input object from a target.

Parameters

- **fed** – The value federate object to use to get the input.
- **target** – The name of the publication that an input is targeting.
- **err** – **[inout]** The error object to complete if there is an error.

Returns

A HelicsInput object, the object will not be valid and err will contain an error code if no input with the specified key exists.

void **helicsFederateClearUpdates**(HelicsFederate fed)

Clear all the update flags from a federates inputs.

Parameters

fed – The value federate object for which to clear update flags.

void **helicsFederateRegisterFromPublicationJSON**(HelicsFederate fed, const char *json, HelicsError *err)

Register the publications via JSON publication string.

This would be the same JSON that would be used to publish data.

Parameters

- **fed** – The value federate object to use to register the publications.
- **json** – The JSON publication string.
- **err** – **[inout]** The error object to complete if there is an error.

void **helicsFederatePublishJSON**(HelicsFederate fed, const char *json, HelicsError *err)

Publish data contained in a JSON file or string.

Parameters

- **fed** – The value federate object through which to publish the data.
- **json** – The publication file name or literal JSON data string.
- **err** – **[inout]** The error object to complete if there is an error.

int **helicsFederateGetPublicationCount**(HelicsFederate fed)

Get the number of publications in a federate.

Returns

(-1) if fed was not a valid federate otherwise returns the number of publications.

int **helicsFederateGetInputCount**(HelicsFederate fed)

Get the number of inputs in a federate.

Returns

(-1) if fed was not a valid federate otherwise returns the number of inputs.

Publication

Warning

doxygenfunction: Cannot find function "helicsPublicationIsValid" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsPublicationPublishBytes" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsPublicationPublishString" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsPublicationPublishInteger" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsPublicationPublishBoolean" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsPublicationPublishDouble" in doxygen xml output for

project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsPublicationPublishTime” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsPublicationPublishChar” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsPublicationPublishComplex” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsPublicationPublishVector” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsPublicationPublishComplexVector” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsPublicationPublishNamedPoint” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsPublicationPublishDataBuffer” in doxygen xml output for

project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsPublicationAddTarget” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsPublicationGetType” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsPublicationGetName” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsPublicationGetUnits” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsPublicationGetInfo” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsPublicationSetInfo” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsPublicationGetTag” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsPublicationSetTag” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsPublicationGetOption” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsPublicationSetOption” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsPublicationSetMinimumChange” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Input

Warning

doxygenfunction: Cannot find function “helicsInputIsValid” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputAddTarget” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetByteCount” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetBytes” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetDataBuffer” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetStringSize” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetString” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetInteger” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetBoolean” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetDouble” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetTime” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetChar” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetComplexObject” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetComplex” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetVectorSize” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetVector” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetComplexVector” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetNamedPoint” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputSetDefaultBytes” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsInputSetDefaultString" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsInputSetDefaultInteger" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsInputSetDefaultBoolean" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsInputSetDefaultTime" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsInputSetDefaultChar" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsInputSetDefaultDouble" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsInputSetDefaultComplex" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsInputSetDefaultVector" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsInputSetDefaultComplexVector" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsInputSetDefaultNamedPoint" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsInputGetType" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsInputGetPublicationType" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsInputGetPublicationDataType" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsInputGetName" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetTarget” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetUnits” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetInjectionUnits” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetExtractionUnits” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetInfo” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputSetInfo” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetTag” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputSetTag” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputGetOption” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputSetOption” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsInputSetMinimumChange” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

HelicsBool **helicsInputIsUpdated**(HelicsInput ipt)

Check if a particular input was updated.

Returns

HELICS_TRUE if it has been updated since the last value retrieval.

HelicsTime **helicsInputLastUpdateTime**(HelicsInput ipt)

Get the last time a input was updated.

void **helicsInputClearUpdate**(HelicsInput ipt)

Clear the updated flag from an input.

MessageFederate

HelicsEndpoint **helicsFederateRegisterEndpoint**(HelicsFederate fed, const char *name, const char *type, HelicsError *err)

Create an endpoint.

The endpoint becomes part of the federate and is destroyed when the federate is freed so there are no separate free functions for endpoints.

Parameters

- **fed** – The federate object in which to create an endpoint must have been created with helicsCreateMessageFederate or helicsCreateCombinationFederate.
- **name** – The identifier for the endpoint. This will be prepended with the federate name for the global identifier.
- **type** – A string describing the expected type of the publication (may be NULL).
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns

An object containing the endpoint, or nullptr on failure.

HelicsEndpoint **helicsFederateRegisterGlobalEndpoint**(HelicsFederate fed, const char *name, const char *type, HelicsError *err)

Create an endpoint.

The endpoint becomes part of the federate and is destroyed when the federate is freed so there are no separate free functions for endpoints.

Parameters

- **fed** – The federate object in which to create an endpoint must have been created with helicsCreateMessageFederate or helicsCreateCombinationFederate.
- **name** – The identifier for the endpoint, the given name is the global identifier.
- **type** – A string describing the expected type of the publication (may be NULL).
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns

An object containing the endpoint, or nullptr on failure.

HelicsEndpoint **helicsFederateRegisterTargetedEndpoint**(HelicsFederate fed, const char *name, const char *type, HelicsError *err)

Create a targeted endpoint. Targeted endpoints have specific destinations predefined and do not allow sending messages to other endpoints

The endpoint becomes part of the federate and is destroyed when the federate is freed so there are no separate free functions for endpoints.

Parameters

- **fed** – The federate object in which to create an endpoint must have been created with helicsCreateMessageFederate or helicsCreateCombinationFederate.
- **name** – The identifier for the endpoint. This will be prepended with the federate name for the global identifier.
- **type** – A string describing the expected type of the publication (may be NULL).
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns

An object containing the endpoint, or nullptr on failure.

HelicsEndpoint **helicsFederateRegisterGlobalTargetedEndpoint**(HelicsFederate fed, const char *name, const char *type, HelicsError *err)

Create a global targeted endpoint, Targeted endpoints have specific destinations predefined and do not allow sending messages to other endpoints

The endpoint becomes part of the federate and is destroyed when the federate is freed so there are no separate free functions for endpoints.

Parameters

- **fed** – The federate object in which to create an endpoint must have been created with helicsCreateMessageFederate or helicsCreateCombinationFederate.
- **name** – The identifier for the endpoint, the given name is the global identifier.
- **type** – A string describing the expected type of the publication (may be NULL).
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns

An object containing the endpoint, or nullptr on failure.

HelicsEndpoint **helicsFederateGetEndpoint**(HelicsFederate fed, const char *name, HelicsError *err)

Get an endpoint object from a name.

The object will not be valid and err will contain an error code if no endpoint with the specified name exists.

Parameters

- **fed** – The message federate object to use to get the endpoint.
- **name** – The name of the endpoint.
- **err** – **[inout]** The error object to complete if there is an error.

Returns

A HelicsEndpoint object.

HelicsEndpoint **helicsFederateGetEndpointByIndex**(HelicsFederate fed, int index, HelicsError *err)

Get an endpoint by its index, typically already created via registerInterfaces file or something of that nature.

The HelicsEndpoint returned will be NULL if given an invalid index.

Parameters

- **fed** – The federate object in which to create a publication.
- **index** – The index of the publication to get.
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns

A HelicsEndpoint.

HelicsBool **helicsFederateHasMessage**(HelicsFederate fed)

Check if the federate has any outstanding messages.

Parameters

fed – The federate to check.

Returns

HELICS_TRUE if the federate has a message waiting, HELICS_FALSE otherwise.

int **helicsFederatePendingMessageCount**(HelicsFederate fed)

Returns the number of pending receives for the specified destination endpoint.

Parameters

fed – The federate to get the number of waiting messages from.

HelicsMessage **helicsFederateGetMessage**(HelicsFederate fed)

Receive a communication message for any endpoint in the federate.

The return order will be in order of endpoint creation. So all messages that are available for the first endpoint, then all for the second, and so on. Within a single endpoint, the messages are ordered by time, then source_id, then order of arrival.

Returns

A HelicsMessage which references the data in the message.

HelicsMessage **helicsFederateCreateMessage**(HelicsFederate fed, HelicsError *err)

Create a new empty message object.

The message is empty and isValid will return false since there is no data associated with the message yet.

Parameters

- **fed** – the federate object to associate the message with
- **err** – **[inout]** An error object to fill out in case of an error.

Returns

A HelicsMessage containing the message data.

void **helicsFederateClearMessages**(HelicsFederate fed)

Clear all stored messages from a federate.

This clears messages retrieved through helicsEndpointGetMessage or helicsFederateGetMessage

Parameters

fed – The federate to clear the message for.

int **helicsFederateGetEndpointCount**(HelicsFederate fed)

Get the number of endpoints in a federate.

Parameters

fed – The message federate to query.

Returns

(-1) if fed was not a valid federate, otherwise returns the number of endpoints.

Endpoint

HelicsBool **helicsEndpointIsValid**(HelicsEndpoint endpoint)

Check if an endpoint is valid.

Parameters

endpoint – The endpoint object to check.

Returns

HELICS_TRUE if the Endpoint object represents a valid endpoint.

void **helicsEndpointSetDefaultDestination**(HelicsEndpoint endpoint, const char *dst, HelicsError *err)

Set the default destination for an endpoint if no other endpoint is given.

Parameters

- **endpoint** – The endpoint to set the destination for.
- **dst** – A string naming the desired default endpoint.
- **err** – **[inout]** A pointer to an error object for catching errors.

const char ***helicsEndpointGetDefaultDestination**(HelicsEndpoint endpoint)

Get the default destination for an endpoint.

Parameters

endpoint – The endpoint to set the destination for.

Returns

A string with the default destination.

void **helicsEndpointSendString**(HelicsEndpoint endpoint, const char *message, HelicsError *err)

Send a message to the targeted destination.

Parameters

- **endpoint** – The endpoint to send the data from.
- **message** – The string to send.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsEndpointSendStringTo**(HelicsEndpoint endpoint, const char *message, const char *dst, HelicsError *err)

Send a message to the specified destination.

Parameters

- **endpoint** – The endpoint to send the data from.
- **message** – The string to send.

- **dst** – The target destination. Use nullptr to send to the default destination.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsEndpointSendStringToAt**(HelicsEndpoint endpoint, const char *message, const char *dst, HelicsTime time, HelicsError *err)

Send a message to the specified destination at a specific time.

Parameters

- **endpoint** – The endpoint to send the data from.
- **message** – The data to send.
- **dst** – The target destination. Use nullptr to send to the default destination.
- **time** – The time the message should be sent.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsEndpointSendStringAt**(HelicsEndpoint endpoint, const char *message, HelicsTime time, HelicsError *err)

Send a message at a specific time to the targeted destinations.

Parameters

- **endpoint** – The endpoint to send the data from.
- **message** – The data to send.
- **time** – The time the message should be sent.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsEndpointSendBytes**(HelicsEndpoint endpoint, const void *data, int inputDataLength, HelicsError *err)

Send a message to the targeted destination.

Parameters

- **endpoint** – The endpoint to send the data from.
- **data** – The data to send.
- **inputDataLength** – The length of the data to send.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsEndpointSendBytesTo**(HelicsEndpoint endpoint, const void *data, int inputDataLength, const char *dst, HelicsError *err)

Send a message to the specified destination.

Parameters

- **endpoint** – The endpoint to send the data from.
- **data** – The data to send.
- **inputDataLength** – The length of the data to send.
- **dst** – The target destination. Use nullptr to send to the default destination.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsEndpointSendBytesToAt**(HelicsEndpoint endpoint, const void *data, int inputDataLength, const char *dst, HelicsTime time, HelicsError *err)

Send a message to the specified destination at a specific time.

Parameters

- **endpoint** – The endpoint to send the data from.
- **data** – The data to send.
- **inputDataLength** – The length of the data to send.
- **dst** – The target destination. Use nullptr to send to the default destination.
- **time** – The time the message should be sent.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsEndpointSendBytesAt**(HelicsEndpoint endpoint, const void *data, int inputDataLength, HelicsTime time, HelicsError *err)

Send a message at a specific time to the targeted destinations

Parameters

- **endpoint** – The endpoint to send the data from.
- **data** – The data to send.
- **inputDataLength** – The length of the data to send.
- **time** – The time the message should be sent.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsEndpointSendMessage**(HelicsEndpoint endpoint, HelicsMessage message, HelicsError *err)

Send a message object from a specific endpoint.

Parameters

- **endpoint** – The endpoint to send the data from.
- **message** – The actual message to send which will be copied.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsEndpointSendMessageZeroCopy**(HelicsEndpoint endpoint, HelicsMessage message, HelicsError *err)

Send a message object from a specific endpoint, the message will not be copied and the message object will no longer be valid after the call.

Parameters

- **endpoint** – The endpoint to send the data from.
- **message** – The actual message to send which will be copied.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsEndpointSubscribe**(HelicsEndpoint endpoint, const char *key, HelicsError *err)

Subscribe an endpoint to a publication.

Parameters

- **endpoint** – The endpoint to use.
- **key** – The name of the publication.
- **err** – **[inout]** A pointer to an error object for catching errors.

HelicsBool **helicsEndpointHasMessage**(HelicsEndpoint endpoint)

Check if a given endpoint has any unread messages.

Parameters

endpoint – The endpoint to check.

Returns
HELICS_TRUE if the endpoint has a message, HELICS_FALSE otherwise.

int **helicsEndpointPendingMessageCount**(HelicsEndpoint endpoint)

Returns the number of pending receives for all endpoints of a particular federate.

Parameters
endpoint – The endpoint to query.

HelicsMessage **helicsEndpointGetMessage**(HelicsEndpoint endpoint)

Receive a packet from a particular endpoint.

Parameters
endpoint – **[in]** The identifier for the endpoint.

Returns
A message object.

HelicsMessage **helicsEndpointCreateMessage**(HelicsEndpoint endpoint, HelicsError *err)

Create a new empty message object.

The message is empty and isValid will return false since there is no data associated with the message yet.

Parameters

- **endpoint** – The endpoint object to associate the message with.
- **err** – **[inout]** An error object to fill out in case of an error.

Returns
A new HelicsMessage.

void **helicsEndpointClearMessages**(HelicsEndpoint endpoint)

Clear all stored messages stored from an endpoint.

This clears messages retrieved through helicsEndpointGetMessage or helicsEndpointCreateMessage

Parameters
endpoint – The endpoint to clear the message for.

const char ***helicsEndpointGetType**(HelicsEndpoint endpoint)

Get the type specified for an endpoint.

Parameters
endpoint – The endpoint object in question.

Returns
The defined type of the endpoint.

const char ***helicsEndpointGetName**(HelicsEndpoint endpoint)

Get the name of an endpoint.

Parameters
endpoint – The endpoint object in question.

Returns
The name of the endpoint.

const char ***helicsEndpointGetInfo**(HelicsEndpoint end)

Get the local information field of an endpoint.

Parameters
end – The endpoint to query.

Returns
A string with the info field string.

void **helicsEndpointSetInfo**(HelicsEndpoint endpoint, const char *info, HelicsError *err)

Set the data in the interface information field for an endpoint.

Parameters

- **endpoint** – The endpoint to set the information for
- **info** – The string to store in the field
- **err** – **[inout]** An error object to fill out in case of an error.

const char ***helicsEndpointGetTag**(HelicsEndpoint endpoint, const char *tagname)

Get the data in a specified tag of an endpoint

Parameters

- **endpoint** – The endpoint to query.
- **tagname** – The name of the tag to query.

Returns

A string with the tag data.

void **helicsEndpointSetTag**(HelicsEndpoint endpoint, const char *tagname, const char *tagvalue, HelicsError *err)

Set the data in a specific tag for an endpoint.

Parameters

- **endpoint** – The endpoint to query.
- **tagname** – The string to set.
- **tagvalue** – The string value to associate with a tag.
- **err** – **[inout]** An error object to fill out in case of an error.

void **helicsEndpointSetOption**(HelicsEndpoint endpoint, int option, int value, HelicsError *err)

Set a handle option on an endpoint.

Parameters

- **endpoint** – The endpoint to modify.
- **option** – Integer code for the option to set /ref helics_handle_options.
- **value** – The value to set the option to.
- **err** – **[inout]** An error object to fill out in case of an error.

int **helicsEndpointGetOption**(HelicsEndpoint endpoint, int option)

Set a handle option on an endpoint.

Parameters

- **endpoint** – The endpoint to modify.
- **option** – Integer code for the option to set /ref helics_handle_options.

Returns

the value of the option, for boolean options will be 0 or 1

void **helicsEndpointAddSourceTarget**(HelicsEndpoint endpoint, const char *targetEndpoint, HelicsError *err)

add a source target to an endpoint, Specifying an endpoint to receive undirected messages from

Parameters

- **endpoint** – The endpoint to modify.
- **targetEndpoint** – the endpoint to get messages from

- **err** – **[inout]** An error object to fill out in case of an error.

void **helicsEndpointAddDestinationTarget**(HelicsEndpoint endpoint, const char *targetEndpoint, HelicsError *err)

add a destination target to an endpoint, Specifying an endpoint to send undirected messages to

Parameters

- **endpoint** – The endpoint to modify.
- **targetEndpoint** – the name of the endpoint to send messages to
- **err** – **[inout]** An error object to fill out in case of an error.

void **helicsEndpointRemoveTarget**(HelicsEndpoint endpoint, const char *targetEndpoint, HelicsError *err)

remove an endpoint from being targeted

Parameters

- **endpoint** – The endpoint to modify.
- **targetEndpoint** – the name of the endpoint to send messages to
- **err** – **[inout]** An error object to fill out in case of an error.

void **helicsEndpointAddSourceFilter**(HelicsEndpoint endpoint, const char *filterName, HelicsError *err)

add a source Filter to an endpoint

Parameters

- **endpoint** – The endpoint to modify.
- **filterName** – the name of the filter to add
- **err** – **[inout]** An error object to fill out in case of an error.

void **helicsEndpointAddDestinationFilter**(HelicsEndpoint endpoint, const char *filterName, HelicsError *err)

add a destination filter to an endpoint

Parameters

- **endpoint** – The endpoint to modify.
- **filterName** – The name of the filter to add.
- **err** – **[inout]** An error object to fill out in case of an error.

Message

Warning

doxygenfunction: Cannot find function “helicsMessageGetSource” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageGetDestination” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsMessageGetOriginalSource" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsMessageGetOriginalDestination" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsMessageGetTime" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsMessageGetString" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsMessageGetMessageID" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsMessageGetFlagOption" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsMessageGetByteCount" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageGetBytes” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageDataBuffer” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageIsValid” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageSetSource” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageSetDestination” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageSetOriginalSource” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageSetOriginalDestination” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageSetTime” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageResize” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageReserve” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageSetMessageID” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageClearFlags” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageSetFlagOption” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageSetString” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageSetData” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageSetDataBuffer” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageAppendData” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageCopy” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageClone” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageFree” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsMessageClear” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Data

HelicsBool **helicsDataBufferIsValid**(HelicsDataBuffer data)

check whether a buffer is valid

void **helicsDataBufferFree**(HelicsDataBuffer data)

free a DataBuffer

int32_t **helicsDataBufferSize**(HelicsDataBuffer data)

get the data buffer size

int32_t **helicsDataBufferCapacity**(HelicsDataBuffer data)

get the data buffer capacity

HelicsBool **helicsDataBufferReserve**(HelicsDataBuffer data, int32_t newCapacity)

increase the capacity a data buffer can hold without reallocating memory

Returns

HELICS_TRUE if the reservation was successful HELICS_FALSE otherwise

HelicsDataBuffer **helicsDataBufferClone**(HelicsDataBuffer data)

create a new data buffer and copy an existing buffer

Returns
a data buffer object with a copy of the data

int32_t **helicsDataBufferFillFromInteger**(HelicsDataBuffer data, int64_t value)

convert an integer to serialized bytes in a buffer

Parameters

- **data** – the HelicsDataBuffer to fill
- **value** – the value to serialize into the dataBuffer

Returns
the buffer size after filling

int32_t **helicsDataBufferFillFromDouble**(HelicsDataBuffer data, double value)

convert a double to serialized bytesin a buffer

convert a double to serialized bytes

Parameters

- **data** – the HelicsDataBuffer to fill
- **value** – the value to serialize into the dataBuffer

Returns
the buffer size after filling

int32_t **helicsDataBufferFillFromString**(HelicsDataBuffer data, const char *value)

convert a string to serialized bytesin a buffer

convert a string to serialized bytes

Parameters

- **data** – the HelicsDataBuffer to fill
- **value** – the value to serialize into the dataBuffer

Returns
the buffer size after filling

int32_t **helicsDataBufferFillFromRawString**(HelicsDataBuffer data, const char *str, int stringSize)

convert a raw string (may contain nulls) to serialized bytesin a buffer

convert a string to serialized bytes

Parameters

- **data** – the HelicsDataBuffer to fill
- **str** – the value to serialize into the dataBuffer
- **stringSize** – the size of the string value to serialize into the dataBuffer

Returns
the buffer size after filling

int32_t **helicsDataBufferFillFromBoolean**(HelicsDataBuffer data, HelicsBool value)

convert a bool to serialized bytesin a buffer

Parameters

- **data** – the HelicsDataBuffer to fill

- **value** – the value to serialize into the dataBuffer

Returns
the buffer size after filling

int32_t **helicsDataBufferFillFromChar**(HelicsDataBuffer data, char value)

convert a char to serialized bytesin a buffer

convert a char to serialized bytes

Parameters

- **data** – the HelicsDataBuffer to fill
- **value** – the value to serialize into the dataBuffer

Returns
the buffer size after filling

int32_t **helicsDataBufferFillFromTime**(HelicsDataBuffer data, HelicsTime value)

convert a time to serialized bytesin a buffer

convert a HelicsTime to serialized bytes

Parameters

- **data** – the HelicsDataBuffer to fill
- **value** – the value to serialize into the dataBuffer

Returns
the buffer size after filling

int32_t **helicsDataBufferFillFromComplex**(HelicsDataBuffer data, double real, double imag)

convert a complex pair to serialized bytesin a buffer

Parameters

- **data** – the HelicsDataBuffer to fill
- **real** – the real part of the imaginary number to serialize into the dataBuffer
- **imag** – the imaginary part of the imaginary number to serialize into the dataBuffer

Returns
the buffer size after filling

int32_t **helicsDataBufferFillFromComplexObject**(HelicsDataBuffer data, HelicsComplex value)

convert a complex object to serialized bytesin a buffer

Parameters

- **data** – the HelicsDataBuffer to fill
- **value** – the value to serialize into the dataBuffer

Returns
the buffer size after filling

int32_t **helicsDataBufferFillFromVector**(HelicsDataBuffer data, const double *value, int dataSize)

convert a real vector to serialized bytesin a buffer

Parameters

- **data** – the HelicsDataBuffer to fill
- **value** – the value to serialize into the dataBuffer

Returns
the buffer size after filling

int32_t **helicsDataBufferFillFromNamedPoint**(HelicsDataBuffer data, const char *name, double value)

convert a named point to serialized bytesin a buffer

Parameters

- **data** – the HelicsDataBuffer to fill
- **value** – the value to serialize into the dataBuffer

Returns
the buffer size after filling

int32_t **helicsDataBufferFillFromComplexVector**(HelicsDataBuffer data, const double *value, int dataSize)

convert a complex vector to serialized bytesin a buffer

Parameters

- **data** – the HelicsDataBuffer to fill
- **value** – the value to serialize into the dataBuffer

Returns
the buffer size after filling

int **helicsDataBufferType**(HelicsDataBuffer data)

extract the data type from the data buffer, if the type isn't recognized UNKNOWN is returned

int64_t **helicsDataBufferToInteger**(HelicsDataBuffer data)

convert a data buffer to an int

double **helicsDataBufferToDouble**(HelicsDataBuffer data)

convert a data buffer to a double

HelicsBool **helicsDataBufferToBoolean**(HelicsDataBuffer data)

convert a data buffer to a boolean

char **helicsDataBufferToChar**(HelicsDataBuffer data)

convert a data buffer to a char

int **helicsDataBufferStringSize**(HelicsDataBuffer data)

get the size of memory required to retrieve a string from a data buffer this includes space for a null terminator

void **helicsDataBufferToString**(HelicsDataBuffer data, char *outputString, int maxStringLen, int *actualLength)

void **helicsDataBufferToRawString**(HelicsDataBuffer data, char *outputString, int maxStringLen, int *actualLength)

HelicsTime **helicsDataBufferToTime**(HelicsDataBuffer data)

convert a data buffer to a time

HelicsComplex **helicsDataBufferToComplexObject**(HelicsDataBuffer data)

convert a data buffer to a complex object

void **helicsDataBufferToComplex**(HelicsDataBuffer data, double *real, double *imag)

convert a data buffer to complex values

int **helicsDataBufferVectorSize**(HelicsDataBuffer data)

get the number of elements that would be required if a vector were retrieved

void **helicsDataBufferToVector**(HelicsDataBuffer data, double values[], int maxlen, int *actualSize)

convert a data buffer to double vector values

void **helicsDataBufferToComplexVector**(HelicsDataBuffer data, double values[], int maxlen, int *actualSize)

convert a data buffer to complex double vector values

Parameters

- **data** – the buffer containing data
- **values** – the storage for the converted data
- **maxlen** – the number of complex values that the values vector can hold
- **actualSize** – the number of complex values copied to values array

void **helicsDataBufferToNamedPoint**(HelicsDataBuffer data, char *outputString, int maxStringLength, int *actualLength, double *val)

HelicsBool **helicsDataBufferConvertToType**(HelicsDataBuffer data, int newDataType)

convert the data in a data buffer to a different type representation

Parameters

- **data** – the buffer to convert
- **newDataType** – the type that it is desired for the buffer to be converted to

Returns

true if the conversion was successful

FilterFederate

HelicsFilter **helicsFederateRegisterFilter**(HelicsFederate fed, HelicsFilterTypes type, const char *name, HelicsError *err)

Create a source Filter on the specified federate.

Filters can be created through a federate or a core, linking through a federate allows a few extra features of name matching to function on the federate interface but otherwise equivalent behavior

Parameters

- **fed** – The federate to register through.
- **type** – The type of filter to create /ref HelicsFilterTypes.
- **name** – The name of the filter (can be NULL).
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns

A HelicsFilter object.

HelicsFilter **helicsFederateRegisterGlobalFilter**(HelicsFederate fed, HelicsFilterTypes type, const char *name, HelicsError *err)

Create a global source filter through a federate.

Filters can be created through a federate or a core, linking through a federate allows a few extra features of name matching to function on the federate interface but otherwise equivalent behavior.

Parameters

- **fed** – The federate to register through.
- **type** – The type of filter to create /ref HelicsFilterTypes.
- **name** – The name of the filter (can be NULL).

- **err** – **[inout]** A pointer to an error object for catching errors.

Returns
A HelicsFilter object.

HelicsFilter **helicsFederateRegisterCloningFilter**(HelicsFederate fed, const char *name, HelicsError *err)

Create a cloning Filter on the specified federate.

Cloning filters copy a message and send it to multiple locations, source and destination can be added through other functions.

Parameters

- **fed** – The federate to register through.
- **name** – The name of the filter (can be NULL).
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns
A HelicsFilter object.

HelicsFilter **helicsFederateRegisterGlobalCloningFilter**(HelicsFederate fed, const char *name, HelicsError *err)

Create a global cloning Filter on the specified federate.

Cloning filters copy a message and send it to multiple locations, source and destination can be added through other functions.

Parameters

- **fed** – The federate to register through.
- **name** – The name of the filter (can be NULL).
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns
A HelicsFilter object.

int **helicsFederateGetFilterCount**(HelicsFederate fed)

Get the number of filters registered through a federate.

Parameters
fed – The federate object to use to get the filter.

Returns
A count of the number of filters registered through a federate.

HelicsFilter **helicsFederateGetFilter**(HelicsFederate fed, const char *name, HelicsError *err)

Get a filter by its name, typically already created via registerInterfaces file or something of that nature.

Parameters

- **fed** – The federate object to use to get the filter.
- **name** – The name of the filter.
- **err** – **[inout]** The error object to complete if there is an error.

Returns
A HelicsFilter object, the object will not be valid and err will contain an error code if no filter with the specified name exists.

HelicsFilter **helicsFederateGetFilterByIndex**(HelicsFederate fed, int index, HelicsError *err)

Get a filter by its index, typically already created via registerInterfaces file or something of that nature.

Parameters

- **fed** – The federate object in which to create a publication.
- **index** – The index of the publication to get.
- **err** – **[inout]** A pointer to an error object for catching errors.

Returns
A HelicsFilter, which will be NULL if an invalid index is given.

Filter

void **helicsFilterSetCustomCallback**(HelicsFilter filter, HelicsMessage (*filtCall)(HelicsMessage message, void *userData), void *userdata, HelicsError *err)

Set a general callback for a custom filter.

Add a custom filter callback for creating a custom filter operation in the C shared library.

Parameters

- **filter** – The filter object to set the callback for.
- **filtCall** – A callback with signature helics_message_object(helics_message_object, void *); The function arguments are the message to filter and a pointer to user data. The filter should return a new message.
- **userdata** – A pointer to user data that is passed to the function when executing.
- **err** – **[inout]** A pointer to an error object for catching errors.

HelicsBool **helicsFilterIsValid**(HelicsFilter filt)

Check if a filter is valid.

Parameters
filt – The filter object to check.

Returns
HELICS_TRUE if the Filter object represents a valid filter.

const char ***helicsFilterGetName**(HelicsFilter filt)

Get the name of the filter and store in the given string.

get the name of the filter

Parameters
filt – The given filter.

Returns
A string with the name of the filter.

void **helicsFilterSet**(HelicsFilter filt, const char *prop, double val, HelicsError *err)

Set a property on a filter.

Parameters

- **filt** – The filter to modify.
- **prop** – A string containing the property to set.
- **val** – A numerical value for the property.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsFilterSetString**(HelicsFilter filt, const char *prop, const char *val, HelicsError *err)

Set a string property on a filter.

Parameters

- **filt** – The filter to modify.
- **prop** – A string containing the property to set.
- **val** – A string containing the new value.
- **err** – **[inout]** A pointer to an error object for catching errors.

double **helicsFilterGetPropertyDouble**(HelicsFilter filt, const char *prop)

Get a double property from a filter.

Parameters

- **filt** – The filter to retrieve a value from.
- **prop** – A string containing the property to get.

const char ***helicsFilterGetPropertyString**(HelicsFilter filt, const char *prop)

Get a string property on a filter. The string output memory is valid until a subsequent call to getPropertyString on the particular filter.

Parameters

- **filt** – The filter to retrieve a value from.
- **prop** – A string containing the property to get.

void **helicsFilterAddDestinationTarget**(HelicsFilter filt, const char *dst, HelicsError *err)

Add a destination target to a filter.

All messages going to a destination are copied to the delivery address(es).

Parameters

- **filt** – The given filter to add a destination target to.
- **dst** – The name of the endpoint to add as a destination target.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsFilterAddSourceTarget**(HelicsFilter filt, const char *source, HelicsError *err)

Add a source target to a filter.

All messages coming from a source are copied to the delivery address(es).

Parameters

- **filt** – The given filter.
- **source** – The name of the endpoint to add as a source target.
- **err** – **[inout]** A pointer to an error object for catching errors.

Warning

doxygenfunction: Cannot find function “helicsFilterAddDeliveryEndpoint” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsFilterRemoveTarget” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsFilterRemoveDeliveryEndpoint” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsFilterGetInfo” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsFilterSetInfo” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsFilterGetTag” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsFilterSetTag” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsFilterSetOption” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function “helicsFilterGetOption” in doxygen xml output for project “helics” from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Translator

void **helicsTranslatorSetCustomCallback**(HelicsTranslator translator, void (*toMessageCall)(HelicsDataBuffer value, HelicsMessage message, void *userData), void (*toValueCall)(HelicsMessage message, HelicsDataBuffer value, void *userData), void *userdata, HelicsError *err)

Set a general callback for a custom translator.

Add a pair of custom callbacks for running a translator operation in the C shared library.

Parameters

- **translator** – The translator object to set the callbacks for.

  –

- **toMessageCall** – A callback with signature void(HelicsDataBuffer, HelicsMessage, void *); The function arguments are raw Value data, the messageObject to fill out and a pointer to user data.
- **toValueCall** – A callback with signature void(HelicsMessage, HelicsDataBuffer, void *); The function arguments are a message object, the data buffer to fill out and a pointer to user data.
- **userdata** – A pointer to user data that is passed to the functions when executing.
- **err** – **[inout]** A pointer to an error object for catching errors.

HelicsBool **helicsTranslatorIsValid**(HelicsTranslator trans)

Check if a translator is valid.

Parameters

trans – The translator object to check.

Returns

HELICS_TRUE if the Translator object represents a valid translator.

const char ***helicsTranslatorGetName**(HelicsTranslator trans)

Get the name of the translator and store in the given string.

get the name of the translator

Parameters

trans – The given translator.

Returns

A string with the name of the translator.

void **helicsTranslatorSet**(HelicsTranslator trans, const char *prop, double val, HelicsError *err)

Set a property on a translator.

Parameters

- **trans** – The translator to modify.
- **prop** – A string containing the property to set.
- **val** – A numerical value for the property.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsTranslatorSetString**(HelicsTranslator trans, const char *prop, const char *val, HelicsError *err)

Set a string property on a translator.

Parameters

- **trans** – The translator to modify.
- **prop** – A string containing the property to set.
- **val** – A string containing the new value.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsTranslatorAddInputTarget**(HelicsTranslator trans, const char *input, HelicsError *err)

Add an input to send a translator output.

All messages sent to a translator endpoint get translated and published to the translators target inputs. This method adds an input to a translators which will receive translated messages.

Parameters

- **trans** – The given translator to add a destination target to.
- **input** – The name of the input which will be receiving translated messages
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsTranslatorAddPublicationTarget**(HelicsTranslator trans, const char *pub, HelicsError *err)

Add a source publication target to a translator.

When a publication publishes data the translator will receive it and convert it to a message sent to a translators destination endpoints. This method adds a publication which publishes data the translator receives and sends to its destination endpoints.

Parameters

- **trans** – The given translator.
- **pub** – The name of the publication to subscribe.
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsTranslatorAddSourceEndpoint**(HelicsTranslator trans, const char *ept, HelicsError *err)

Add a source endpoint target to a translator.

The translator will "translate" all message sent to it. This method adds an endpoint which can send the translator data.

Parameters

- **trans** – The given translator.
- **ept** – The name of the endpoint which will send the endpoint data
- **err** – **[inout]** A pointer to an error object for catching errors.

void **helicsTranslatorAddDestinationEndpoint**(HelicsTranslator trans, const char *ept, HelicsError *err)

Add a destination target endpoint to a translator.

The translator will "translate" all message sent to it. This method adds an endpoint which will receive data published to the translator.

Parameters

- **trans** – The given translator.
- **ept** – The name of the endpoint the translator sends data to.
- **err** – **[inout]** A pointer to an error object for catching errors.

Warning

doxygenfunction: Cannot find function "helicsTranslatorRemoveTarget" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsTranslatorGetInfo" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsTranslatorSetInfo" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsTranslatorGetTag" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsTranslatorSetTag" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsTranslatorSetOption" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Warning

doxygenfunction: Cannot find function "helicsTranslatorGetOption" in doxygen xml output for project "helics" from directory: /home/docs/checkouts/readthedocs.org/user_builds/helics/checkouts/latest/build-doxygen/docs/xml

Query

const char ***helicsQueryExecute**(HelicsQuery query, HelicsFederate fed, HelicsError *err)

Execute a query.

The call will block until the query finishes which may require communication or other delays.

Parameters

- **query** – The query object to use in the query.
- **fed** – A federate to send the query through.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

A pointer to a string. The string will remain valid until the query is freed or executed again. The return will be nullptr if fed or query is an invalid object, the return string will be "#invalid" if the query itself was invalid.

const char *helicsQueryCoreExecute(HelicsQuery query, HelicsCore core, HelicsError *err)

Execute a query directly on a core.

The call will block until the query finishes which may require communication or other delays.

Parameters

- **query** – The query object to use in the query.
- **core** – The core to send the query to.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

A pointer to a string. The string will remain valid until the query is freed or executed again. The return will be nullptr if core or query is an invalid object, the return string will be "#invalid" if the query itself was invalid.

const char *helicsQueryBrokerExecute(HelicsQuery query, HelicsBroker broker, HelicsError *err)

Execute a query directly on a broker.

The call will block until the query finishes which may require communication or other delays.

Parameters

- **query** – The query object to use in the query.
- **broker** – The broker to send the query to.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

A pointer to a string. The string will remain valid until the query is freed or executed again. The return will be nullptr if broker or query is an invalid object, the return string will be "#invalid" if the query itself was invalid

void helicsQueryExecuteAsync(HelicsQuery query, HelicsFederate fed, HelicsError *err)

Execute a query in a non-blocking call.

Parameters

- **query** – The query object to use in the query.
- **fed** – A federate to send the query through.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

const char *helicsQueryExecuteComplete(HelicsQuery query, HelicsError *err)

Complete the return from a query called with /ref helicsExecuteQueryAsync.

The function will block until the query completes /ref isQueryComplete can be called to determine if a query has completed or not.

Parameters

- **query** – The query object to complete execution of.
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

Returns

A pointer to a string. The string will remain valid until the query is freed or executed again. The return will be nullptr if query is an invalid object

HelicsBool helicsQueryIsCompleted(HelicsQuery query)

Check if an asynchronously executed query has completed.

This function should usually be called after a QueryExecuteAsync function has been called.

Parameters

query – The query object to check if completed.

Returns

Will return HELICS_TRUE if an asynchronous query has completed or a regular query call was made with a result, and false if an asynchronous query has not completed or is invalid

void helicsQuerySetTarget(HelicsQuery query, const char *target, HelicsError *err)

Update the target of a query.

Parameters

- **query** – The query object to change the target of.
- **target** – the name of the target to query
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void helicsQuerySetQueryString(HelicsQuery query, const char *queryString, HelicsError *err)

Update the queryString of a query.

Parameters

- **query** – The query object to change the target of.
- **queryString** – the new queryString
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void helicsQuerySetOrdering(HelicsQuery query, int32_t mode, HelicsError *err)

Update the ordering mode of the query, fast runs on priority channels, ordered goes on normal channels but goes in sequence

Parameters

- **query** – The query object to change the order for.
- **mode** – 0 for fast, 1 for ordered
- **err** – **[inout]** An error object that will contain an error code and string if any error occurred during the execution of the function.

void **helicsQueryFree**(HelicsQuery query)

Free the memory associated with a query object.

void **helicsQueryBufferFill**(HelicsQueryBuffer buffer, const char *queryResult, int strSize, HelicsError *err)

Set the data for a query callback.

There are many queries that HELICS understands directly, but it is occasionally useful to have a federate be able to respond to specific queries with answers specific to a federate.

Parameters

- **buffer** – The buffer received in a helicsQueryCallback.
- **queryResult** – Pointer to the data with the query result to fill the buffer with.
- **strSize** – The size of the string.
- **err** – **[inout]** A pointer to an error object for catching errors.

4.2.2 C++ API Reference (Doxygen)

The latest Doxygen generated docs for the C++ API can be found at https://docs.helics.org/en/latest/doxygen/index.html

4.2.3 REST Queries API

The REST Queries API is described by the OpenAPI spec in queries.yaml.

If the ReDoc Sphinx extension is available in the build environment, this page is enhanced with the generated interactive API reference automatically.

4.2.4 C API

4.2.5 Python API

4.2.6 matHELICS API

There's no dedicated documentation for the matHELICS API at this time as the API is virtually identical to the other higher level languages such as Python and Julia. Of particular note, as in those two languages, matHELICS supports the use of complex as a native data type. So `helicsInputGetComplex()` returns a complex value not a list of two floats, one for the real and one for the imaginary.

There is also inline documentation in the source code files that may be helpful in providing insight.

4.2.7 Julia API

4.2.8 REST Queries API

4.3 Configuration Options Reference

Many of the HELICS entities have significant configuration options. Rather than comprehensively list these options while explaining the features themselves, we've created this section of the User Guide to serve as a reference as to what they are, what they do, and how to use them. This reference is oriented-around the use of JSON files for configuration and is an attempt to be comprehensive in listing and explaining those options. As will be explained below, many of these options are accessible via direct API calls though some of these calls are general in nature (such as *helicsFederateInfoSetIntegrerProperty* to set the logging level, among other things).

4.3.1 Configuration methods

Generally, there are three ways in which a co-simulation can be configured and all the various options can be defined:

1. Using direct API calls in the federate source code.
2. Using command-line switches/flags while beginning execution of the federate
3. Using a JSON configuration file (and calling helicsCreateValueFederateFromConfig, helicsCreateMessageFederateFromConfig, or helicsCreateComboFederateFromConfig)

Not all configuration options are available in all three forms but often they are. For example, it is not possible (nor convenient) to configure a publication for a federate from the command line but it is possible to do so with the JSON config file and with API calls.

Choosing configuration method

Which method you use to configure your federate and co-simulation significantly depends on the circumstances of the co-simulation and details of any existing code-base being used. Here is some guidance, though, to help in guiding you're decision in how to do this:

- **If possible, use a JSON configuration file** - Using a JSON configuration file creates separation between the code base of the federation and its use in a particular co-simulation. This allows for a modularity between the functionality the federate provides and the particular co-simulation in which it is applied. For example, a power system federate can easily be reconfigured to work on one model vs another through the use of a JSON configuration file. The particular publications and subscriptions may change but the main functionality of the federate (solving the power flow) does not. To use the JSON file for configuration, one of three specific APIs needs to be called: in the file:
  - `helicsCreateValueFederateFromConfig` C++ | *C* | Python | Julia
  - `helicsCreateMessageFederateFromConfig` C++ | *C* | Python | Julia
  - `helicsCreateCombinationFederateFromConfig` C++ | *C* | Python | Julia
- **JSON configuration produces a natural artifact that defines the co-simulation** - Another advantage of the external configuration in the JSON file is that it is a human-readable artifact that can be distributed separately from the source code that provides a lot of information about how the co-simulation was run. In fact, its possible to just look at the configuration files of a federation and do some high-level debugging (checking to see that the subscriptions and publications are aligned, for example).
- **New federates in ill-defined use cases may benefit from API configuration** - The modularity that the JSON config provides may not offer many benefits if the federate is newly integrated into HELICS and/or is part of an evolving analysis. During these times the person(s) doing the integration may just want to make direct API calls instead of having to mess with writing the federate code and a configuration file. There will likely be a point in the future when the software is more codified and switching to a JSON configuration makes more sense.
- **Command-line configuration (where possible) allows for small, quick changes to the configuration** - Because the command line doesn't provide comprehensive access to the necessary configuration, it will never be a stand-alone configuration option but it does have the advantage of providing quick access right as a user is instantiating the federate. This is an ideal place to make small changes to the configuration (e.g. changing the minimum time step) without having to edit any files.
- **API configuration is most useful for dynamic configuration** - If there is a need to change the configuration of a given federate dynamically, the API is the only way to do that. Such needs are not common but there are cases where, for example, it may be necessary to define the configuration based on the participants in the federation (*e.g.* publications, subscriptions, timing). It's possible to use *queries* to understand the composition and configuration of the federation and then use the APIs to define the configuration of the federate in question.

How to Use This Reference

The remainder of this reference lists the configuration options that are supported in the JSON configuration file. Where possible, the corresponding C++ API calls and the links to that documentation will be provided. Generally, the command-line options use the exact same syntax as the JSON configuration options preceded by a `--` and followed by either an = or a space and then the parameter value (*i.e.* `--name testname`). In the cases where a single letter switch is available, that will be listed (*i.e.* `-n testname`).

Default values are shown in "[]" following the name(s) of the option.

When an API exists, its name is shown along with links to the specific API documentation for a few (but, sadly, not all) of the supported languages. Many of the options are set with generic functions (*e.g.* `helicsFederateInfoSetFlagOption`) and in those cases the option being set is specified by an enumerated value. In C, these values (*e.g.* `helics_flag_uninterruptible`) are set to integer value (*e.g.* 1); in this document that integer value follows the enumeration string in brackets. If using the C interface, the integer value must be used. The C++ interface supports the use of the enumerated value directly as do the Python and Julia interfaces with slight syntactical variations (Python: `helics.HELICS_FLAG_INTERRUPTIBLE` and Julia: `HELICS.HELICS_FLAG_INTERRUPTIBLE`).

4.3.2 Sample Configurations

The JSON configuration file below shows all the configuration options in a single file along with their default values and shows what section of the file they should be placed in. Most JSON configuration files will require far fewer options than shown here; items marked with "**" are required.

Many of the configuration parameters have alternate names that provide the same functionality. Only one of the names is shown in this configuration file but the alternative names are listed in the reference below. Generally, the supported names are the same string in nocase, camelCase, and snake_case.

An example of one publication, subscription, named input, endpoint, and filter is also shown. The values for each of these options is arbitrary and in the case of filters, many more values are supported and a description of each is provided.

(Note that the JSON standard does not support comments and thus the block below is not valid JSON. The JSON parser HELICS uses does support comments)

```
{
  // General
  **"name": "arbitrary federate name",**
  "core_type": "zmq",
  "core_name": "core name",
  "core_init_string" : "",
  "autobroker": false,
  "connection_required": false,
  "connection_optional": true,
  "strict_input_type_checking": false,
  "terminate_on_error": false,
  "source_only": false,
  "observer": false,
  "dynamic":false,
  "only_update_on_change": false,
  "only_transmit_on_change": false,
  "broker_key": "",

  //Logging
  "logfile": "output.log"
  "log_level": "warning",
```

(continues on next page)

(continued from previous page)

```
"force_logging_flush": false,
"file_log_level": "",
"console_log_level": "",
"dump_log": false,
"logbuffer": 10,
"value_buffer_warning": 100,

//Timing
"ignore_time_mismatch_warnings": false,
"uninterruptible": false,
"period": 0,
"offset": 0,
"time_delta": 0,
"minTimeDelta": 0,
"input_delay": 0,
"output_delay": 0,
"real_time": false,
"rt_tolerance": 0.2,
"rt_lag": 0.2,
"rt_lead": 0.2,
"grant_timeout": 0,
"max_cosim_duration":0,
"wait_for_current_time_update": false,
"restrictive_time_policy": false,
"slow_responding": false,

//Iteration
"rollback": false,
"max_iterations": 10,
"forward_compute": false,

// other
"indexgroup":5,

//Network
"interfaceNetwork": "local",
"brokeraddress": "127.0.0.1"
"reuse_address": false,
"noack": false,
"maxsize": 4096,
"maxcount": 256,
"networkretries": 5,
"osport": false,
"brokerinit": "",
"server_mode": "",
"interface": (local IP address),
"port": 1234,
"brokerport": 22608,
"localport": 8080,
"portstart": 22608,
"encrypted": false,
"encryption_config":"encryption_config.json",
```

(continues on next page)

(continued from previous page)

```
"publications" | "subscriptions" | "inputs": [
  {
    **"key": "publication key",**
    "type": "",
    "unit": "m",
    "global": false,
    "connection_optional": true,
    "connection_required": false,
    "tolerance": -1,
    // for targets can be singular or plural, if an array must use plural form
    "targets": "",
    "buffer_data":  false, indication the publication should buffer data
    "strict_input_type_checking": false,
    "alias": "",
    "ignore_unit_mismatch": false,
    "info": "",
  },
],
"publications" :[
    {
        "only_transmit_on_change": false,
    }
]    ,
"subscriptions": [
  {
    "only_update_on_change": false,
    "default": value,
  }
],
"inputs": [
  {
    "connections": 1,
    "input_priority_location": 0,
    "clear_priority_list": possible to have this as a config option?
    "single_connection_only": false,
    "multiple_connections_allowed": false
    "multi_input_handling_method": "average",
    // for targets can be singular or plural, if an array must use plural form
    "targets": ["pub1", "pub2"]
    "default": 5.5,
  }
],
"endpoints": [
  {
    "name": "endpoint name",
    "type": "endpoint type",
    "global": true,
    "destination" | "target" : "default endpoint destination",
    "alias": "",
    "subscriptions": "",
```

(continues on next page)

(continued from previous page)

```
      "filters": "",
      "info": ""
    }
  ],
  "filters": [
    {
      "name": "filter name",
      "source_targets": "endpoint name",
      "destination_targets": "endpoint name",
      "info": "",
      "operation": "randomdelay",
      "properties": {
        "name": "delay",
        "value": 600
      }
    }
  ]
  "translators": [
    {
      "name": "translator name",
      // can use singular form if only a single target
      "source_target": "publication name",
      "destination_targets": "endpoint name",
      "info": "",
      "type": "JSON",
    }
  ]
}
```

4.3.3 General Federate Options

There are a number of flags which control how a federate acts with respect to timing and its signal interfaces.

name | -n (required)

API: `helicsFederateInfoSetCoreName` C++ | *C* | Python | Julia

Every federate must have a unique name across the entire federation; this is functionally the address of the federate and is used to determine where HELICS messages are sent. An error will be generated if the federate name is not unique.

---

core_type | -t ["zmq"]

Alternative names: `coretype` | `coreType`

API: `helicsFederateInfoSetCoreTypeFromString` C++ | *C* | Python | Julia

There are a number of technologies or message buses that can be used to send HELICS messages among federates. Every HELICS enabled simulator has code in it that creates a core which connects to a HELICS broker using one of these messaging technologies. ZeroMQ (zmq) is the default core type and most commonly used but there are also cores that use TCP and UDP networking protocols directly (forgoing ZMQ's guarantee of delivery and reconnection functions), IPC (uses Boost's interprocess communication for fast in-memory message-passing but only works if all federates are running on the same physical computer), and MPI (for use on HPC clusters where MPI is installed). See the *User Guide page on core types* for more details.

`core_name` [HELICS-generated]

Alternative names: `corename` | `coreName`

API: helicsFederateInfoSetCoreName C++ | *C* | Python | Julia

Only applicable for `ipc` and `test` core types; otherwise can be left undefined.

`core_init_string` | `-i` [null]

Alternative names: `coreinitstring` | `coreInitString`

API: `helicsFederateInfoSetCoreInitString` C++ | *C* | Python | Julia

A command-line-like string that specifies options for the core as it connects to the federation. These options are:

- `--broker=` | `broker_address=` | `brokeraddress`: IP address of broker
- `--brokerport=`: Port number on which the broker is communicating
- `--broker_rank=`: For MPI cores only; identifies the MPI rank of the broker
- `--broker_tag=`: For MPI cores only; identifies the MPI tag of the broker
- `--localport=`: Port number to use when communicating with this core
- `--autobroker`: When included the core will automatically generate a broker (does not work for all core types)
- `--key=`: Specifies a key to use when communicating with the broker. Only federates with this key specified will be able to talk to the broker with the same `key` value. This is used to prevent federations running on the same hardware from accidentally interfering with each other.
- `--profiler=log` - Send the profiling messages to the default logging file. `log` can be replaced with a path to an alternative file where only the profiling messages will be sent. See the *User Guide page on profiling* for further details. If a file is specified it is cleared.
- `--profiler_append=somefile.txt` - Send the profiling messages to file and leave the existing contents appending new data. See the *User Guide page on profiling* for further details.

In addition to these options, all options shown in the `broker_init_string` are also valid.

`autobroker` [false]

API: (none)

Automatically generate a broker if one cannot be connected to. For federations with only one broker (often the case) and/or with federations containing custom federates that were developed for this particular application, it can be convenient to create the broker in the process of creating a specific federate; this option allows that to take place. The downside to this is it creates a federation with a small amount of mystery as the broker is not clearly shown to be launched as its own federate alongside the other federates and those unfamiliar with the federation composition may have to spend some extra time to understand where the broker is coming from. This option does not work for all core types, it is specifically designed for inproc and testing cores. Others may be added later.

---

broker_init_string [null]

Alternative names: brokerinitstring, brokerInitString

API: helicsFederateInfoSetBrokerInitString C++ | *C* | Python | Julia

String used to define the configuration of the broker if one is autogenerated. Such configuration typically includes things like broker IP addresses and port numbers. Again, if this is a co-simulation running on a single computer (and is the only HELICS co-simulation running on said computer) the default option is likely to be sufficient. The following options are available for this string:

- --federates= - The minimum total number of federates expected to join through this broker and any sub-brokers below it. Equivalent to -f.
- --required_federates= - A federate name or comma-separated list of federate names that must be registered before the broker or core enters initialization mode.
- --max_federates= - The maximum number of federates allowed to join through a broker.
- --name= - Name of the broker; can be used by federates to specify which broker to use
- --max_iterations= - Maximum iterations allowed when using the re-iteration API
- --min_broker_count= - The minimum number of brokers that the co-simulation must have to begin initialization. (This option is not available for cores)
- --max_broker_count= - The maximum number of brokers that the co-simulation should allow. (This option is not available for cores)
- --slow_responding - Removes the requirement for the broker to respond to pings from other entities in the co-simulation in a timely manner and forces the assumption that this broker is still connected to the federation.
- --restrictive_time_policy - Forces the broker to use the most restrictive (conservative) timing policy when granting times to federates. Has the potential to increase co-simulation time as time grants may happen later then they actually need to.
- --terminate_on_error - All errors from any member of the federation will cause the broker to terminate the co-simulation for the entire federation.
- --force_logging_flush - Force writing to the log after every message.
- --log_file= - Name of file use for logging for this broker.
- --log_level= - Specifies the level of logging (both file and console) for this broker.
- --file_log_level= - Specifies the level of logging to file for this broker.
- --console_log_level= - Specifies the level of logging to file for this broker.
- --dumplog - Captures a record of all logging messages and writes them out to file or console when the broker terminates.
- --globaltime - Specify that the broker should use a globalTime coordinator to coordinate a master clock time with all federates.
- --asynctime - Specify that the federation should use the asynchronous time coordinator (only minimal time management is handled in HELICS and federates are allowed to operate independently).
- --timing = ("async"|"global"|"default"|"distributed") - specify the timing mode to use for time coordination.
  - distributed - Time management is distributed and managed by each federates. This is the default
  - global - HELICS centrally manages the time coordination; for larger federations this is likely to be slower

  - `async` - federates manage their own time with minimal coordination from HELICS (such as when using real-time simulators)

- `--tick=` - Heartbeat period in ms. When brokers fail to respond after 2 ticks secondary actions are taking to confirm the broker is still connected to the federation. Times can also be entered as strings such as "15s" or "75ms".
- `--global_disconnect` - Specify that federates should wait to disconnect until the federation is completed
- `--timeout=` milliseconds to wait for all the federates to connect to the broker (can also be entered as a time like '10s' or '45ms')
- `--network_timeout=` - Time to establish a socket connection in ms. Times can also be entered as strings such as "15s" or "75ms".
- `--error_timeout=` - Time in ms to wait after an error state is reached before terminating. Times can also be entered as strings such as "15s" or "75ms".
- `--query_timeout=` - Time in ms to wait for a query to complete. Times can also be entered as strings such as "15s" or "75ms".
- `--grant_timeout=` - Time in ms to wait to allow a time request to wait before triggering diagnostic actions. Times can also be entered as strings such as "15s" or "75ms".
- `--max_cosim_duration=`- The time in ms the co-simulation should be allowed to run. If the time is exceeded the co-simulation will terminate automatically.
- `--children=` - The minimum number of child objects the broker should expect before allowing entry to the initializing state.
- `--subbrokers=` - The minimum number of child objects the broker should expect before allowing entry to the initializing state. Same as `--children` but might be clearer in some cases with multilevel hierarchies.
- `--local_federates=` - The minimum number of federates attached through direct child cores of this broker before allowing entry to the initializing state.
- `--local_subbrokers=` - The minimum number of direct child brokers, excluding cores, before allowing entry to the initializing state.
- `--brokerkey=` - A broker key to use for connections to ensure federates are connecting with a specific broker and only appropriate federates connect with the broker. See *simultaneous co-simulations* for more information.
- `--profiler=log` - Send the profiling messages to the default logging file. `log` can be replaced with a path to an alternative file where only the profiling messages will be sent. See the *User Guide page on profiling* for further details. If a file is specified it is cleared.
- `--profiler_append=somefile.txt` - Send the profiling messages to file and leave the existing contents appending new data. See the *User Guide page on profiling* for further details.
- `--time_monitor=` - Specify the name of the federate to monitor the time from and generate periodic log messages in the broker as the federate updates its time.
- `--time_monitor_period=` - can only be used with `--time_monitor`, set the minimum time period which must elapse in simulation before another log message from the time monitor is generated
- `--disable_timer` - Disables all timeout in broker operation
- `--debugging` - Equivalent to `--slow_responding --disable_timer`
- `--logbuffer` - Enable buffering recent log messages for retrieval with the "logs" query. Optionally specify the size of the circular log buffer; defaults to 10 messages if no size is supplied.
- `--allow_remote_control` - Enables the broker to respond to certain remote commands such as "disconnect"

- `--dynamic` - Allow for dynamic federations where federates can join after the co-simulation has begun. This capability is disabled by default.
- `--disable_dynamic_sources` - Prevents data sources from registering after the federation has entered initializing mode. This capability is **enabled** by default

The configuration options for a broker may now be fully specified in json or toml files instead of having to pass them as command line arguments.

`terminate_on_error` [false]

Alternative names: `terminateonerror`, `terminateOnError`

API: `helicsFederateInfoSetFlagOption` C++ | *C* | Python | Julia

Property's enumerated name: `HELICS_FLAG_TERMINATE_ON_ERROR` [72]

If the `terminate_on_error` flag is set then a federate encountering an internal error will trigger a global error and cause the entire federation to terminate. Errors of this nature are typically the result of configuration errors, such as having a required publication that is not used or incompatible units or types on publications and subscriptions. This is the same option that is available in the `broker_init_string` but applied only to a specific federate (whereas when applied at the broker level it is effectively applied to all federates).

`source_only` | [false]

Alternative names: `sourceonly`, `sourceOnly`

API: `helicsFederateInfoSetFlagOption` C++ | *C* | Python | Julia

Property's enumerated name: `HELICS_FLAG_SOURCE_ONLY` [4]

Used to indicate to the federation that this federate is only producing data and has no inputs/subscriptions. Specifying this when appropriate allows HELICS to more efficiently grant times to the federation.

`observer` [false]

API: `helicsFederateInfoSetFlagOption` C++ | *C* | Python | Julia

Property's enumerated name: `HELICS_FLAG_OBSERVER` [0]

Used to indicate to the federation that this federate produces no data and only has inputs/subscriptions. Specifying this when appropriate allows HELICS to more efficiently grant times to the federation.

`reentrant` [false]

API: `helicsFederateInfoSetFlagOption` C++ | *C* | Python | Julia

Property's enumerated name: `HELICS_FLAG_REENTRANT` [38]

Used to indicate to the broker that this federate may disconnect and reconnect at a later time using the same federate name. Without setting this flag, the federate would have to rejoin under a different name and would be considered a new federate by the federation. This flag only has an effect if the "dynamic" flag is also set on the broker.

---

broker_key [null]

API: helicsFederateSetBrokerKey C++ | *C* | Python | Julia

Setting a broker key only allows federates that have the same broker key to be part of the federation.

4.3.4 Logging Options

log_file [null]

Alternative names: logfile, logFile *API:* helicsFederateSetLogFile

C++ | *C* | Python | Julia

Specifies the name of the log file where logging messages will be written.

---

log_level [0]

Alternative names: loglevel, logLevel

API: helicsFederateInfoSetIntegerProperty C++ | *C* | Python | Julia

Property's enumerated name: HELICS_PROPERTY_INT_LOG_LEVEL [271]

Valid values:

- none - HELICS_LOG_LEVEL_NO_PRINT
- no_print - HELICS_LOG_LEVEL_NO_PRINT
- error - HELICS_LOG_LEVEL_ERROR
- profiling - HELICS_LOG_LEVEL_PROFILING
- warning - HELICS_LOG_LEVEL_WARNING
- summary - HELICS_LOG_LEVEL_SUMMARY
- connections - HELICS_LOG_LEVEL_CONNECTIONS
- interfaces - HELICS_LOG_LEVEL_INTERFACES
- timing - HELICS_LOG_LEVEL_TIMING
- data - HELICS_LOG_LEVEL_DATA
- debug - HELICS_LOG_LEVEL_DEBUG
- trace - HELICS_LOG_LEVEL_TRACE

Determines the level of detail for log messages. In the list above, the keywords on the left can be used when specifying the logging level via a JSON configuration file. The enumerations on the right are used when configuring via the API.

---

file_log_level [null]

Alternative names: fileloglevel, fileLogLevel

API: helicsFederateInfoSetIntegerProperty C++ | *C* | Python | Julia

Property's enumerated name: HELICS_PROPERTY_INT_FILE_LOG_LEVEL [272]

Valid values: Same as in loglevel

Allows a distinct log level for the written log file to be specified. By default the logging level to file and console are identical and will only differ if `file_log_level` or `console_log_level` are defined.

`console_log_level` [null]

Alternative names: `consoleloglevel`, `consoleLogLevel`

API: `helicsFederateInfoSetIntegerProperty` C++ | *C* | Python | Julia

Property's enumerated name: `HELICS_PROPERTY_INT_CONSOLE_LOG_LEVEL` [274]

Valid values: Same as in `loglevel`

Allows a distinct log level for the written log file to be specified. By default the logging level to file and console are identical and will only differ if `file_log_level` or `console_log_level` are defined.

`force_logging_flush` [false]

Alternative names: `forceloggingflush`, `forceLoggingFlush`

API: `helicsFederateInfoSetFlagOption` C++ | *C* | Python | Julia

Property's enumerated name: `HELICS_FLAG_FORCE_LOGGING_FLUSH` [88]

Setting this option forces HELICS logging messages to be flushed to file after each one is written. This prevents the buffered IO most OSs implement to be bypassed such that all messages appear in the log file immediately after being written at the cost of slower simulation times due to more time spent writing to file.

`dump_log` [false]

Alternative names: `dumplog`, `dumpLog`

API: `helicsFederateInfoSetFlagOption` C++ | *C* | Python | Julia

Property's enumerated name: `HELICS_FLAG_DUMPLOG` [89]

When set, a record of all messages is captured and written out to the log file at the conclusion of the co-simulation.

`logbuffer` [10]

Alternative names: `log_buffer`, `logBuffer`

API: `helicsFederateInfoSetIntegerProperty` C++ | *C* | Python | Julia

Property's enumerated name: `HELICS_PROPERTY_INT_LOG_BUFFER` [276]

When set to a number greater than 0 will enable the most recent X log messages of the object to be buffered for retrieval via the *"logs" query*. Also see discussion in *Logging*.

---

value_buffer_warning [100]

Alternative names: valuebufferwarning

API: helicsFederateInfoSetIntegerProperty C++ | C | Python | Julia

Property's enumerated name: HELICS_PROPERTY_INT_VALUE_BUFFER_WARNING [283]

Sets the federate warning threshold, in MiB, for queued future value payload data. The default threshold is 100 MiB. Set this option to 0 to disable the warning. This warning is intended to highlight cases where value publications are being buffered faster than the receiving federate is consuming them, such as when a producer advances ahead of a subscriber.

4.3.5 Other Options

indexgroup [0]

Alternative names: index_group, indexGroup

API: helicsFederateInfoSetIntegerProperty C++ | C | Python | Julia

Property's enumerated name: HELICS_PROPERTY_INT_INDEX_GROUP [282]

When set to a number greater than 0 will modify the internal federateId codes. Values allowed are from 0 to 16. Each group allows 100,000,000 federates. In a few select situations, such as ordering messages, or breaking a deadlock in a timing situation, the ordering ties are broken by federate index. It is possible in some scenarios that this ordering can be variable from run to run introducing a source of randomness since federate id's are assigned in the order processed. The indexGroup parameter allows the user to directly influence the internal id, which can eliminate a source of randomness in a few select situations. Specifically, each increment of the index group increases the internal id by 100,000,000 from what it would have been. This allows 17 total allowable option group values 0-16. In most cases, this option will have no observable impact on co-simulation results. If exceeding 100,000,000 care must be exercised in its use to ensure federate Id's do not conflict, by using only larger intervals of increments.

4.3.6 Timing Options

ignore_time_mismatch [false]

Alternative names: ignoretimemismatch, ignoreTimeMismatch

API: helicsFederateInfoSetFlagOption C++ | C | Python | Julia

Property's enumerated name: HELICS_FLAG_IGNORE_TIME_MISMATCH_WARNINGS [67]

If certain timing options (*i.e.* period, or minTimeDelta) are used it is possible for the time granted a federate to be greater than the requested time. This situation would normally generate a warning message, but if this flag is set those warnings are silenced.

---

uninterruptible [false]

API: helicsFederateInfoSetFlagOption C++ | C | Python | Julia

Property's enumerated name: HELICS_FLAG_UNINTERRUPTIBLE [1]

Normally, a federate will be granted a time earlier than it requested when it receives a message from another federate; the presence of any message implies there could be an action the federate needs to take and may generate new messages of its own. There are times, though, when it is important that the federate only be granted a time (and begin simulating/executing again) that it has previously requested. For example, there could be some controller that should only operate at fixed intervals even if new data arrives earlier. In these cases, setting the uninterruptible flag will prevent premature time grants.

period [1ns]

API: helicsFederateInfoSetTimeProperty C++ | *C* | Python | Julia

Property's enumerated name: HELICS_PROPERTY_TIME_PERIOD [140]

Many time-based simulators have a minimum time-resolution or a user-configurable step size. The period parameter can be used to effectively synchronize the times that are granted with the defined simulation period. The default units for period are in seconds but the string for this parameter can include its own units (e.g. "2 ms" or "1 hour"). Setting period will force all time grants to occur at times of n*period even if subscriptions are updated, messages arrive, or the federate requests a time between periods. This value effectively makes the federates uninterruptible during the times between periods. Relatedly…

---

offset [0]

API: helicsFederateInfoSetTimeProperty C++ | *C* | Python | Julia

Property's enumerated name: HELICS_PROPERTY_TIME_OFFSET [141]

There may be cases where it is preferable to have a simulator receive time grants that are offset slightly in time to one or more other federates. Defining an offset value allows this to take place; units are handled the same as in period. Setting both period and offset, will result in the all times granted to the federate in question being constrained to n*period + offset.

---

time_delta [1ns]

Alternative names: timeDelta, timedelta

API: helicsFederateInfoSetTimeProperty C++ | *C* | Python | Julia

Property's enumerated name: HELICS_PROPERTY_TIME_DELTA [137]

timeDelta has some similarities to period; where period constrained the granted time to regular intervals, timeDelta constrains the grant time to a minimum amount from the last granted time. Units are handled the same as in period.

---

input_delay [0]

Alternative names: inputdelay, inputDelay

API: helicsFederateInfoSetTimeProperty C++ | *C* | Python | Julia

Property's enumerated name: HELICS_PROPERTY_INPUT_TIME_DELAY [148]

inputDelay specifies a delay in simulated time between when a signal arrives at a federate and when that federate is notified that a new value is available. outputDelay is similar but applies to signals being sent by a federate. Note that this applies to both value signals and message signals.

---

output_delay [0]

Alternative names: outputdelay, outputDelay

API: helicsFederateInfoSetTimeProperty C++ | *C* | Python | Julia

Property's enumerated name: HELICS_TIME_PROPERTY_OUTPUT_TIME_DELAY [150]

outputDelay is similar to input_delay but applies to signals being sent by a federate. Note that this applies to both value signals and message signals.

---

real_time [false]

Alternative names: realtime, realTime

API: helicsFederateInfoSetFlagOption C++ | *C* | Python | Julia

Property's enumerated name: HELICS_FLAG_REALTIME [16]

If set to true the federate uses rt_lag and rt_lead to match the time grants of a federate to the computer wall clock. If the federate is running faster than real time this will insert additional delays. If the federate is running slower than real time this will cause a force grant, which can lead to non-deterministic behavior. rt_lag can be set to maxVal to disable force grant

---

rt_lag and rt_lead`[0.2]

Alternative names: rtlag, rtLag; rtlead, rtLead

API: helicsFederateInfoSetTimeProperty C++ | *C* | Python | Julia

Property's enumerated name: HELICS_PROPERTY_TIME_RT_LAG [143] and HELICS_PROPERTY_TIME_RT_LEAD [144]

Defines "real-time" for HELICS by setting tolerances for HELICS to use when running a real-time co-simulation. HELICS is forced to keep simulated time within this window of wall-clock time. Most general purpose OSes do not provide guarantees of execution timing and thus very small values of rt_lag and rt_lead (less than 0.005) are not likely to be achievable.

---

rt_tolerance [0.2]

Alternative names: rttolerance, rtTolerance

API: helicsFederateInfoSetTimeProperty C++ | *C* | Python | Julia

Property's enumerated name: HELICS_PROPERTY_TIME_RT_TOLERANCE [145]

Implements the same functionality of rt_lag and rt_lead but does so by using a single value to set symmetrical lead and lag constraints.

---

wait_for_current_time_update [false]

Alternative names: waitforcurrenttimeupdate, waitForCurrentTimeUpdate

API: helicsFederateInfoSetFlagOption C++ | *C* | Python | Julia

Property's enumerated name: HELICS_FLAG_WAIT_FOR_CURRENT_TIME_UPDATE [10]

If set to true, a federate will not be granted the requested time until all other federates have completed at least 1 iteration of the current time or have moved past it. If it is known that 1 federate depends on others in a non-cyclic fashion, this can be used to optimize the order of execution without iterating.

---

restrictive_time_policy [false]

Alternative names: restrictivetimepolicy | restrictiveTimePolicy

API: helicsFederateInfoSetFlagOption C++ | *C* | Python | Julia

Property's enumerated name: HELICS_FLAG_RESTRICTIVE_TIME_POLICY [11]

If set, a federate will not be granted the requested time until all other federates have completed at least 1 iteration of the current time or have moved past it. If it is known that 1 federate depends on others in a non-cyclic fashion, this can be used to optimize the order of execution without iterating.

Using the option restrictive-time-policy forces HELICS to use a fully conservative mode in granting time. This can be useful in situations beyond the current reach of the distributed time algorithms. It is generally used in cases where it is known that some federate is executing and will trigger someone else, but most federates won't know who that might be. This prevents extra messages from being sent and a potential for time skips. It is not needed if some federates are periodic and execute every time step. The flag can be used for federates, brokers, and cores to force very conservative timing with the potential loss of performance as well.

Only applicable to Named Input interfaces (*see section on value federate interface types*), if enabled this flag checks that data type of the incoming signals match that specified for the input.

---

slow_responding [false]

Alternative names: slowresponsing, slowResponding

API: helicsFederateInfoSetFlagOption C++ | *C* | Python | Julia

Property's enumerated name: HELICS_FLAG_SLOW_RESPONDING [29]

If specified on a federate, setting this flag indicates the federate may be slow in responding, and to not forcibly eject the federate from the federation for the slow response. This is an uncommon scenario.

If applied to a core or broker (--slow_responding in the core_init_string or broker_init_string, respectively), it is indicative that the broker doesn't respond to internal pings quickly and should not be disconnected from the federation for the slow response.

---

event_triggered [false]

API: helicsFederateInfoSetTimeProperty C++ | *C* | Python | Julia

Property's enumerated name: HELICS_FLAG_EVENT_TRIGGERED [81]

For federates that are event-driven rather than timing driven, this flag must be set (to increase timing efficiency and avoid timing lock-ups). Event-driven federates are those that don't progress through simulation time at regular timesteps but

instead wait for arriving messages to act. The most common examples are controller federates which generally request infinite time (well, HELICS_TIME_MAXTIME) and rely on HELICS to grant them an earlier time whenever a signal (often message) has arrived. Filter federates are another common federate type that must have this flag set.

4.3.7 Iteration

forward_compute [false]

Alternative names: forwardcompute | forwardCompute

API: helicsFederateInfoSetFlagOption C++ | *C* | Python | Julia

Property's enumerated name: HELICS_FLAG_FORWARD_COMPUTE [14]

Indicates to the broker and the rest of the federation that this federate computes ahead of its granted time and can/does roll back when necessary. Federates able to do this (and who set this flag) allow more efficient time grants to the federation as a whole.

---

rollback [false]

API: helicsFederateInfoSetFlagOption C++ | *C* | Python | Julia

Property's enumerated name: HELICS_FLAG_ROLLBACK [12]

Indicates to the broker and the rest of the federation that this federate can/does roll back when necessary. Federates able to do this (and who set this flag) allow more efficient time grants to the federation as a whole.

---

max_iterations [50]

Alternative names: maxiterations, maxIteration

API: helicsFederateInfoSetIntegerProperty C++ | *C* | Python | Julia

Property's enumerated name: HELICS_PROPERTY_INT_MAX_ITERATIONS [259]

For federates engaged in iteration (recomputing values based on updated inputs at a single simulation timestep) there may be a need to enforce a maximum number of iterations. This option allows that value to be set. When any federate reaches this number of iterations, HELICS will evaluate the federation as a whole and grant the next smallest time supported by the iterating federates. This time will only be granted to the federates that would be able to execute at this time.

4.3.8 General and Per Subscription, Input, or Publication

These options can be set globally for all subscriptions, inputs and publications for a given federate. Even after setting them globally, they can be included in the configuration for an individual subscription, input, or publication, over-riding the global setting.

only_update_on_change and only_transmit_on_change [false]

Alternative names: onlyupdateonchange, onlyUpdateOnChange;onlytransmitonchange, onlyTransmitOnChange

API: helicsFederateInfoSetFlagOption C++ | *C* | Python | Julia

Property's enumerated name: HELICS_FLAG_ONLY_UPDATE_ON_CHANGE [454] and HELICS_FLAG_ONLY_TRANSMIT_ON_CHANGE [452]

Setting these flags prevents new value signals with the same value from being received by the federate or sent by the federate. Setting these flags will reduce the amount of traffic on the HELICS bus and can provide performance improvements in co-simulations with large numbers of messages.

`tolerance` [0]

API: `helicsPublicationSetMinimumChange` and `helicsInputSetMinimumChange`

C++ input and C++ publication | *C input* and *C publication* | Python input and Python publication | Julia input and Julia publication

This option allows the specific numerical definition of "change" when using the `only_update_on_change` and `only_transmit_on_change` options.

`default` [0]

API: `helicsInputSetDefaultX` C++ input | *C input* | Python input | Julia input

This option allows specifying the default value used when no publication has been received. Each datatype has its own API call such as:

- `helicsInputSetDefaultBoolean()`
- `helicsInputSetDefaultBytes()`
- `helicsInputSetDefaultChar()`
- `helicsInputSetDefaultComplex()`
- `helicsInputSetDefaultComplexVector()`
- `helicsInputSetDefaulDouble()`
- `helicsInputSetDefaultInteger()`
- `helicsInputSetDefaultRaw()`
- `helicsInputSetDefaultString()` (used for JSONs)
- `helicsInputSetDefaultVector()`

Though they are not as obviously named, the following two APIs do provide a means of setting the default value as well:

- `helicsInputSetDefaultTime()` - set the default using a HelicsTime value
- `helicsInputSetDefaultNamedPoint()` -set the default NamedPoint which is a pair of a string and double used for tagged values or set points

---

`connection_required` [false]

Alternative names: `connectionrequired` | `connectionRequired`

API: `helicsFederateInfoSetFlagOption` C++ | *C* | Python | Julia

Property's enumerated name: `HELICS_HANDLE_OPTION_CONNECTION_REQUIRED` [397]

When a federate is initialized, one of its tasks is to make sure the recipients of directed signals exist. If, after the federation is initialized, the recipient can't be found, then by default a warning is generated and written to the log file. If the `connections_required` flag is set, this warning becomes a fatal error that stops the co-simulation.

- publications - At least one federate must subscribe to the publications.
- subscriptions - The message being subscribed to must be provided by some other publisher in the federation.

---

connection_optional [true]

Alternative names: connectionoptional, connectionOptional

API: helicsFederateInfoSetFlagOption C++ | *C* | Python | Julia

Property's enumerated name: HELICS_HANDLE_OPTION_CONNECTION_OPTIONAL [402] *Property's enumerated name:* HELICS_HANDLE_OPTION_CONNECTION_OPTIONAL [402]

When an interface requests a target it tries to find a match in the federation. If it cannot find a match at the time the federation is initialized, then the default is to generate a warning. This will not halt the federation but will display a log message. If the connections_optional flag is set on a federate all subsequent addTarget calls on any interface will not generate any message if the target is not available.

---

default_global [false]

Alternative names: defaultglobal, defaultGlobal

API: (no API interface)

Set to true to force all handles to act as globals; see "global" below for further details.

4.3.9 Subscription, Input, and/or Publication Options

These options are valid for subscriptions, inputs, and/or publications (generically called "handles"). The APIs for dealing with registering these handles combine multiple options in the JSON config file and have varying levels of specificity (defining the date type for the handle or defining the handle as global). Rather than listing all APIs for the following options, the main APIs will be listed here and those using them can consult the API references to see which specific APIs are most applicable.

helicsFederateRegisterPublication C++ | *C* | Python | Julia

helicsFederateRegisterSubscription C++ | *C* | Python | Julia

helicsFederateRegisterInput C++ | *C* | Python | Julia

key (required)

- publications - The string in this field is the unique identifier (at the federate level) for the value that will be published to the federation. If global is set (see below) it must be unique to the entire federation.
- subscriptions - This string identifies the federation-unique value that this federate wishes to receive. Unless global has been set to true in the publishings JSON configuration file, the name of the value is formatted as <federate name>/<publication key>. Both of these strings can be found in the publishing federate's JSON configuration file as the name and key strings, respectively. If global is true the string is just the key value.
- input - The string in this field is the unique identifier (at the federate level) that defines the input to receive value signals.

---

type [null]

HELICS supports data types and data type conversion (*as best it can*).

---

unit [null]

HELICS is able to do some levels of unit conversion, currently only on double type publications but more may be added in the future. The units can be any sort of unit string, a wide assortment is supported and can be compound units such as m/s^2 and the conversion will convert as long as things are convertible. The unit match is also checked for other types and an error if mismatching units are detected. A warning is also generated if the units are not understood and not matching. The unit checking and conversion is only active if both the publication and subscription specify units.

---

global [false]

(publications only) `global` is used to indicate that the value in `key` will be used as a global name when other federates are subscribing to the message. This requires that the user ensure that the name is used only once across all federates. Setting `global` to `true` is handy for federations with a small number of federates and a small number of message exchanges as it allows the `key` string to be short and simple. For larger federations, it is likely to be easier to set the flag to `false` and accept the extra naming.

---

buffer_data [false]

Alternative names: `bufferdata` | `bufferData`

API: `helicsInputSetOption` C++ | *C* | Python | Julia

Property's enumerated name: `HELICS_HANDLE_OPTION_BUFFER_DATA` [411]

(only valid for inputs and subscriptions) Setting this flag will buffer the last value sent during the initialization phase of HELICS co-simulations. When the execution phase begins, that value will be resent to the receiving handle.

---

ignore_units_mismatch [null]

Alternative names: `ignoreunitmismatch`, `ignoreUnitMismatch`

API: TODO Under normal operation, handles that are connected (value signals flowing between them) are required to have units that either match or can be directly converted between. If mismatching units are connected, an error is thrown; when this flag is set that error is suppressed.

---

info [""]

API: `helicsInputSetInfo` C++ | *C* | Python | Julia The `info` field is entirely ignored by HELICS and is used as a mechanism to pass configuration information to the federate so that it can properly integrate into the federation. Thus, there is no standard content or format for this field; it is entirely up to the individual simulators to decide how the data in this field (if any) should be used. Often it is used by simulators to map the HELICS names into internal variable names as shown in the above example.

---

strict_input_type_checking [false]

Alternative names: strictinputtypechecking, strictInputTypeChecking

API: helicsFederateInfoSetFlagOption C++ | *C* | Python | Julia

Property's enumerated name: HELICS_HANDLE_OPTION_STRICT_TYPE_CHECKING [414]

Generally, HELICS does *data type conversions where supported* on connected value handles. That is, if a publication is specified as an int and the subscription is specified as a double HELICS will convert the value behind the scenes. Some of these conversions, though, may not be expected; for example, how is HELICS going to convert a complex value to a double? To ensure that no surprises take place in the data type conversion, setting this flag tells HELICS to require the sending and receiving handles to match in datatype. If they do not, an error is thrown and the co-simulation halts.

4.3.10 Publication-only Options

targets [null]

C++ | *C* | Python | Julia

Used to specify which inputs should receive the values from this output. This can be a list of output keys/names.

4.3.11 Input-only Options

Inputs can receive values from multiple sending handles and the means by which those multiple data points for a single handle are managed can be specified with several options. See the *User Guide entry* for further details.

targets

C++ | *C* | Python | Julia

Inputs can specify which outputs (typically publications) they should be pulling from. This is similar to subscriptions but inputs can allow multiple outputs to feed to the same input. This can be a list of output keys/names.

---

connections [null]

API: helicsInputSetOption C++ | *C* | Python | Julia

Property's enumerated name: HELICS_HANDLE_OPTION_CONNECTIONS [522]

Allows an integer number of connections to be considered value for this input handle. Similar to multiple_connections_allowed but allows the number of sending handles to be defined to a specific number.

---

input_priority_location [null]

Alternative names: inputprioritylocation, inputPriorityLocation

API: helicsInputSetOption C++ | *C* | Python | Julia

Property's enumerated name: HELICS_HANDLE_OPTION_INPUT_PRIORITY_LOCATION [510]

When receiving values from multiple sending handles, when the values are received they or organized as a vector. This option is used to define which value in that vector has priority. The API can be called multiple times to set successive priorities.

---

clear_priority_list [false]

Alternative names: clearprioritylist, clearPriorityList

API: helicsInputSetOption C++ | *C* | Python | Julia

Property's enumerated name: HELICS_HANDLE_OPTION_CLEAR_PRIORITY_LIST [512]

When receiving values from multiple sending handles, when the values are received they or organized as a vector. This option is used to clear that priority list and redefine which values have priority.

---

single_connection_only [false]

Alternative names: singleconnectiononly, singleConnectionOnly

API: helicsInputSetOption C++ | *C* | Python | Julia

Property's enumerated name: HELICS_HANDLE_OPTION_SINGLE_CONNECTION_ONLY [407] When set, this forces the input handle to have only one sending handle it will receive from. Setting this flag serves as a sort of double-check to ensure that only one other handle is sending to this input handle and that the federation has been configured as expected.

---

multiple_connections_allowed [true]

Alternative names: multipleconnectionsallowed, multipleConnectionsAllowed

API: helicsInputSetOption C++ | *C* | Python | Julia

Property's enumerated name: HELICS_HANDLE_OPTION_MULTIPLE_CONNECTIONS_ALLOWED [409]

When set, this flag allows the input handle to receive valuesfrom multiple other handles.

---

multi_input_handling_method [none]

Alternative names: multiinputhandlingmethod, multiInputHandlingMethod

API: helicsInputSetOption C++ | *C* | Python | Julia

Property's enumerated name: HELICS_HANDLE_OPTION_MULTI_INPUT_HANDLING_METHOD [507] *Property values:*

- none | no_op
- or
- sum
- max
- min
- average
- mean
- vectorize
- diff

Given that an input can have multiple data sources, a method of reducing those multiple values into one needs to be defined. HELICS supports a number of mathematical operation to perform this reduction.

4.3.12 Endpoint Options

As in the value handles, the registration of endpoints is done through a single API that incorporates multiple options. And as in the value handles, there is a `global` API option to allow the name of the endpoint to be considered global to the federation.

API: `helicsFederateRegisterEndpoint` C++ | *C* | Python | Julia

`name` (required)

The name of the endpoint as it will be known to the rest of the federation.

`type` [null]

API: (none)

HELICS supports data types and data type conversion (as best it can).

`destination` [null]

Alternative names: `target`

API: `helicsEndpointSetDefaultDestination` C++ | *C* | Python | Julia

Defines the default destination for a message sent from this endpoint.

`alias` [null]

API: (none)

Creates a local alias for a handle that may have a long name.

`subscriptions` [null]

API: `helicsEndpointSubscribe` C++ | *C* | Python | Julia

`filters` [null]

See section on Filter Options.

`info` [“”]

API: `helicsEndpointSetInfo` C++ | *C* | Python | Julia The `info` field is entirely ignored by HELICS and is used as a mechanism to pass configuration information to the federate so that it can properly integrate into the federation. Thus, there is no standard content or format for this field; it is entirely up to the individual simulators to decide how the data in this field (if any) should be used. Often it is used by simulators to map the HELICS names into internal variable names as shown in the above example.

4.3.13 Filter Options

Filters are registered with the core or through the application API. There are also Filter object that hide some of the API calls in a slightly nicer interface. Generally a filter will be associated with a specific endpoint as either a source filter or destination filter. Source filters can be chained, as in there can be more than one of them. At present there can only be a single non-cloning destination filter attached to an endpoint.

Non-cloning filters can modify the message in some ways, cloning filters just copy the message and may send it to multiple destinations. Cloning is not considered the "operation" of the filter and can be specified in parallel to another filter operation.

On creation, filters have a target endpoint and an optional name. Custom filters may have input and output types associated with them; this is used for chaining and automatic ordering of filters. Filters do not have to be defined on the same core as the source or destination endpoint on which they act; they can be defined anywhere in the federation and all appropriate messages will be automatically routed through them appropriately.

API:

`helicsFederateRegisterFilter` (C++ | *C* | Python | Julia) to create/register the filter and then

`helicsFilterAddSourceTarget` (C++ | *C* | Python | Julia) or `helicsFilterAddDestinationTarget` (C++ | *C* | Python | Julia) to associate it with a specific endpoint

`name` [null]

API: (none)

Name of the filter; must be unique to a federate.

`source_targets` [null]

Alternative names: `sourcetargets`, `sourceTargets`

API: `helicsFilterAddSourceTarget` C++ | C | Python | Julia

Defines the endpoint sending messages that the filter should act on. All messages coming from the specified endpoint will run through the filter first before being sent on to their specified destination.

`destination_targets` [null]

Alternative names: `destinationtargets`, `destinationtargets`

API: `helicsFilterAddDestinationTarget` C++ | C | Python | Julia

Defines the endpoint receiving messages that the filter should act on. All messages going to the specified endpoint will run through the filter first.

`info` [null]

API: `helicsFilterSetInfo` C++ | *C* | Python | Julia The `info` field is entirely ignored by HELICS and is used as a mechanism to pass configuration information to the federate so that it can properly integrate into the federation. Thus, there is no standard content or format for this field; it is entirely up to the individual simulators to decide how the data in this field (if any) should be used. Often it is used by simulators to map the HELICS names into internal variable names as shown in the above example.

---

operation [null]

API: helicsFederateRegisterFilter C++ | *C* | Python | Julia

Filters have a predefined set of operations they can perform. The following list defines the valid operations for filters. Most filters require additional specifications in properties data structure, an example of which is shown for each filter type.

reroute

This filter reroutes a message to a new destination. it also has an optional filtering mechanism that will only reroute if some patterns are matching. The patterns should be specified by "condition" in the set string the conditions are regular expression pattern matching strings.

Example property object:

```
"operation": "reroute",
 "properties": [
     {
         "name": "newdestination",
         "value": "endpoint name    "
     },
     {
         "name": "condition",
         "value": "regular expression string"
     }
 ]
```

delay

This filter will delay a message by a certain amount fo time.

Example property object:

```
"operation": "delay",
 "properties": {
     "name": "delay",
     "value": "76 ms",
 },
```

random_delay [null]

Alternative names: randomdelay, randomDelay This filter will randomly delay a message according to specified random distribution available options include distribution selection, and 2 parameters for the distribution some distributions only take one parameter in which case the second is ignored. The distributions available are based on those available in the C++ random library.

- **constant**
  - param1="value" this just generates a constant value
- **uniform**
  - param1="min"
  - param2="max"

- **bernoulli** - the bernoulli distribution will return param2 if the bernoulli trial returns true, 0.0 otherwise. Param1 is the probability of returning param2
  - param1="prob"
  - param2="value"
- **binomial**
  - param1=t (cast to int)
  - param2="p"
- **geometric**
  - param 1="prob" the output is param2*geom(param1) so multiplies the integer output of the geometric distribution by param2 to get discrete chunks
- **poisson**
  - param1="mean"
- **exponential**
  - param1="lambda"
- **gamma**
  - param1="alpha"
  - param2="beta"
- **weibull**
  - param1="a"
  - param2="b"
- **extreme_value**
  - param1="a"
  - param2="b"
- **normal**
  - param1="mean"
  - param2="stddev"
- **lognormal**
  - param1="mean"
  - param2="stddev"
- **chi_squared**
  - param1="n"
- **cauchy**
  - param1="a"
  - param2="b"
- **fisher_f**
  - param1="m"
  - param2="n"

- **student_t**
  - param1="n"

```
"operation": "randomdelay",
"properties": [
  {
    "name": "distribution",
    "value": "normal"
  },
  {
    "name": "mean",
    "value": 0
  },
  {
    "name": "stdev",
    "value": 1
  }
]
```

random_drop | randomdrop | randomDrop

This filter will randomly drop a message, the drop probability is specified, and is modeled as a uniform distribution between zero and one.

```
"operation": "random_drop",
 "properties": {
     "name": "prob",
     "value": 0.5,
 },
```

clone

Unlike other filters, cloning is not considered an "operation" and is enabled by setting the "clone" flag. (Prior to version 3.6, it was required to also define the filter "operation" to "clone".)

The clone filter takes any message sent from endpoints listed in the `source_targets` object and sends a copy to endpoints in the `delivery` object. Note that both of these can be strings for a single endpoint or lists for multiple endpoints. In the example below, all messages originating from the endpoints named "source_endpoint_1_name" and "source_endpoint_2_name" will be copied and sent on to the endpoint named "ep_that_receives_cloned_messages".

```
"clone": true,
"source_targets":[
  "source_endpoint_1_name",
  "source_endpoint_2_name"
],
"delivery": "ep_that_receives_cloned_messages"
```

4.3.14 Translator Options

Translators are used to bridge the gap between the value and message interfaces allowing publications to be sent to endpoints and endpoint messages to be sent to inputs as values. A translator functions as publication, input, and endpoint that other interfaces including filters can connect to. Further details can be found on the *documentation page covering translators*.

API: `helicsCoreRegisterTranslator` C++ | C | Python | Julia

or

API: `helicsFederateRegisterGlobalTranslator` C++ | C | Python | Julia

`name` [null]

API: (none, done as part of registering the translator)

Name of the filter; must be unique to a federate.

`type` [null]

API: (none, done as part of registering the translator)

Type of translator; determines the format of the data on the endpoint side of the translator. Must be one of the following: `HELICS_TRANSLATOR_TYPE_CUSTOM`, `HELICS_TRANSLATOR_TYPE_JSON`, or `HELICS_TRANSLATOR_TYPE_BINARY`.

`source_targets` [null]

Alternative names: `sourcetargets`, `sourceTargets`

API: `helicsTranslatorAddPublicationTarget` C++ | C | Python | Julia

or

API: `helicsTranslatorAddSourceEndpoint` C++ | C | Python | Julia

Connects the specified publication to the translator's input or adds the translator's endpoint as a destination for all messages coming from the specified endpoint.

`destination_targets` [null]

Alternative names: `destinationtargets`, `destinationTargets`

API: `helicsTranslatorAddInputTarget` C++ | C | Python | Julia

or

API: `helicsTranslatorAddDestinationEndpoint` C++ | C | Python | Julia

Connects the specified input to the translator's publication (output) or adds the specified endpoint as a destination for all messages coming from the translator's endpoint.

4.3.15 Profiling

HELICS has a profiling capability that allows users to measure the time spent waiting for HELICS to grant it time and how much time is spent executing native code. These measurements are the foundation to understanding how to improve computation performance in a federation. Further details are provided in the *Profiling page in the User Guide.* When enabling profiling at the federate level there are a few APIs that can be utilized.

profiling [false]

API: helicsFederateInfoSetFlagOption C++ | *C* | Python | Julia

Property's enumerated name: HELICS_FLAG_PROFILING [93]

Setting this flag enables profiling for the federate.

---

profiler [null]

Turns on profiling for the federate and allows the specification of the log file where profiling messages will be written. No API is possible with this option as it must be specified prior to the creation of the federates.

---

local_profiling_capture [false]

API: helicsFederateInfoSetFlagOption C++ | *C* | Python | Julia

Property's enumerated name: HELICS_FLAG_LOCAL_PROFILING_CAPTURE [96]

Setting this flag sends the profiling messages to the local federate log rather than propagating them up to the core and/or broker.

---

profiling_marker [false]

API: helicsFederateInfoSetFlagOption C++ | *C* | Python | Julia

Property's enumerated name: HELICS_FLAG_PROFILING_MARKER [95]

Generates an additional marker message if logging is enabled.

4.3.16 Network

For most HELICS users, most of the time, the following network options are not needed. They are most likely to be needed when working in complex networking environments, particularly when running co-simulations across multiple sites with differing network configurations. Many of these options require non-trivial knowledge of network operations, and it is assumed that those who need these options will understand what they do, even with the minimal descriptions given.

interface network

API:

See multiple options for –local, –ipv4, –ipv6, –all

---

reuse_address [false]

Alternative names: reuseaddress, reuseAddress

API: (none)

Allows the server to reuse a bound address, mostly useful for tcp cores.

---

noack_connect [false]

Alternative names: noackconnect, noackConnect Specify that a connection_ack message is not required to be connected with a broker.

---

max_size [4096]

Alternative names: maxsize, maxSize

API: (none)

Message buffer size. Can be increased for large messages to limit the number of retries by the underlying networking protocols.

---

max_count [256]

Alternative names: maxcount, maxCount

API: (none)

Maximum number of messages in queue. Can be increased for large volumes of messages to limit the number of retries by the underlying networking protocols.

---

network_retries [5]

Alternative names: networkretries, networkRetries

API: (none) Maximum number of network retry attempts.

---

encrypted [false]

API: (none) set to true to enable encryption on network types that support encryption

---

encryption_config

Alternative names: encryptionconfig, encryptionConfig

API: (none) specify json or a file containing the configuration options for defining the encrypted interface

---

use_os_port [false]

Alternative names: useosport, useOsPort

API: (none) Setting this flag specifies that the OS should set the port for the HELICS message bus. HELICS will ask the operating system which port to use and will use the indicated port.

---

client or server [null]

API: (none) specify that the network connection should be a server or client. By default neither option is enabled.

---

local_interface [local IP address]

Alternative names: localinterface, localInterface

API: (none) Specifies the IP address (and optionally port) the rest of the federation should use when contacting this federate.

---

broker_address [local IP address]

Alternative names: brokeraddress, brokerAddress

API: (none)

Specifies the IP address (and optionally port) a federate or sub-broker should use when contacting its parent broker

---

broker_port []

Alternative names: brokerport, brokerPort

API: (none)

Specifies the port a federate or sub-broker should use when contacting its parent broker

---

broker_name [null]

Alternative names: brokername, brokerName

API: (none)

---

local_port []

Alternative names: localport, localPort *API:* (none)

Specifies the port the rest of the federation should use when contacting this federate.

---

port_start []

Alternative names: portstart, portStart

API: (none) starting port for automatic port definitions.

---

force [false]

API: (none) Flag specifying that the broker network connection should attempt to override and terminate any existing broker using the specified port.

4.4 Tools with HELICS Support

The following list of tools is a list of tools that have worked with HELICS at some level either on current projects or in the past, or in some cases funded projects that will be working with certain tools. These tools are in various levels of development. Check the corresponding links for more information.

4.4.1 Power Systems Tools

Electric Distribution System Simulation

- GridLAB-D, an open-source tool for distribution power-flow, DER models, basic house thermal and end-use load models, and more. HELICS support currently (8/15/2018) provided in the `develop` branch which you have to build yourself as described here. Or a CMake based branch maintained as part of the GMLC-TDC organization.
- OpenDSS, an open-source tool for distribution powerflow, DER models, harmonics, and other capabilities traditionally found in commercial distribution analysis tools. There are two primary interfaces with HELICS support:
  - OpenDSSDirect.py which provides a "direct" interface to interact with the OpenDSS engine enabling support for non-Windows (Linux, OSX) systems.
  - PyDSS which builds on OpenDSSDirect to provide enhanced advanced inverter models and significantly more robust convergence with high-penetration DER controls along with flexible support for user-defined controls and visualization.
- CYME has been used in connection with a python wrapper interface and through FMI wrapper.

Electric Transmission System Simulation

- GridDyn, an open-source transmission power flow and dynamics simulator. HELICS support provided through the `cmake_updates` branch.
- PSST, an open-source python-based unit-commitment and dispatch market simulator. HELICS examples are included in the HELICS-Tutorial.
- MATPOWER, an open-source Matlab based power flow and optimal power flow tool. HELICS support under development.
- InterPSS, a Java-based power systems simulator. HELICS support under development. Use case instructions can be found here.
- PSLF has some level of support using the experimental python interface.
- PSS/E through the PyPSSE python wrapper (Note: pip install as NREL-pypsse)
- PowerWorld Simulator is an interactive power system simulation package designed to simulate high voltage power system operation on a time frame ranging from several minutes to several days.
- SAInt is a commercial tool that provides planning and analysis across multiple energy systems including electric system production cost, power flow, and dynamics in addition to natural gas, district heat, and other networks. A HELICS use case is described here with source code in this repository.
- PyPower does not have a standard HELICS integration but it has been used on various projects. PYPOWER is a power flow and Optimal Power Flow (OPF) solver. It is a port of MATPOWER to the Python programming language. Current features include:
  - DC and AC (Newton's method & Fast Decoupled) power flow and
  - DC and AC optimal power flow (OPF)

Real time simulators

- OpalRT A few projects are using HELICS to allow connections between Opal RT and other simulations
- RTDS Some planning or testing for RTDS linkages to HELICS is underway and will be required for some known projects

Electric Power Market simulation

- FESTIV, the Flexible Energy Scheduling Tool for Integrating Variable Generation, provides multi-timescale steady-state power system operations simulations that aims to replicate the full time spectrum of scheduling and reserve processes (multi-step commitment and dispatch plus simplified AGC) to meet energy and reliability needs of the bulk power system.
- PLEXOS, a commercial production cost simulator. Support via OpenPLEXOS is under development.
- MATPOWER (described above) also includes basic optimal powerflow support.
- PyPower (described above) also includes basic AC and DC optimal powerflow solvers.

Contingency Analysis tools

- CAPE protection system modeling.
- DCAT Dynamic contingency analysis tool.

4.4.2 Communication Tools

- HELICS provides built-in support for simple communications manipulations such as delays, lossy channels, etc. through its built-in filters.
- ns-3, a discrete-event communication network simulator. Supported via the HELICS ns-3 module.
- OMNet++ is a public-source, component-based, modular and open-architecture simulation environment with strong GUI support and an embeddable simulation kernel. Its primary application area is the simulation of communication networks, but it has been successfully used in other areas like the simulation of IT systems, queueing networks, hardware architectures and business processes as well. Early stage development with OMNET++ and HELICS is underway and a prototype example is available in HELICS-omnetpp.

4.4.3 Gas Pipeline Modeling

- NGFAST.
- GasModels.jl.
- SAInt is a commercial tool that includes detailed support for natural gas simulation and planning in addition to other energy systems. See additional information, including a gas-grid use case, under Electric Transmission System Simulation above.

4.4.4 Optimization packages

- GAMS.
- JuMP support is provided through the HELICS Julia interface.

4.4.5 Transportation modeling

- BEAM.
- POLARIS.

4.5 Built-In HELICS Apps

Included with HELICS are a number of apps that provide useful utilities and test programs for getting started and running with HELICS

4.5.1 Recorder

The Recorder application is one of the HELICS apps available with the library Its purpose is to provide a easy way to capture data from a federation It acts as a federate that can "capture" values or messages from specific publications or direct endpoints or cloned endpoints which exist elsewhere

Command line arguments

```
allowed options:

command line only:
  -? [ --help ]          produce help message
  -v [ --version ]       display a version string
  --config-file arg      specify a configuration file to use

configuration:
  --local                specify otherwise unspecified endpoints and
                         publications as local( i.e.the keys will be prepended
                         with the player name
  --stop arg             the time to stop the player
  --quiet                turn off most display output

allowed options:

configuration:
  -b [ --broker ] arg    address of the broker to connect
  -n [ --name ] arg      name of the player federate
  --corename arg         the name of the core to create or find
  -c [ --core ] arg      type of the core to connect to
  --offset arg           the offset of the time steps
  --period arg           the period of the federate
  --timedelta arg        the time delta of the federate
  --rttolerance arg      the time tolerance of the real time mode
  -i [ --coreinit ] arg  the core initialization string
  --separator arg        separator character for local federates
  --inputdelay arg       the input delay on incoming communication of the
                         federate
  --outputdelay arg      the output delay for outgoing communication of the
                         federate
  -f [ --flags ] arg     named flag for the federate

allowed options:

configuration:
  --tags arg             tags to record, this argument may be specified any
                         number of times
  --endpoints arg        endpoints to capture, this argument may be specified
                         multiple time
```

(continues on next page)

(continued from previous page)

```
  --sourceclone arg      existing endpoints to capture generated packets from,
                         this argument may be specified multiple time
  --destclone arg        existing endpoints to capture all packets with the
                         specified endpoint as a destination, this argument may
                         be specified multiple time
  --clone arg            existing endpoints to clone all packets to and from
  --capture arg          capture all the publications of a particular federate
                         capture="fed1;fed2"  supports multiple arguments or a
                         semicolon/comma separated list
  -o [ --output ] arg    the output file for recording the data
  --allow_iteration      allow iteration on values
  --verbose              print all value results to the screen
  --marker arg           print a statement indicating time advancement     every <arg>␣
↪seconds of the simulation
                         is the period of the marker
  --mapfile arg          write progress to a map file for concurrent progress
                         monitoring
```

also permissible are all arguments allowed for federates and any specific broker specified:

Command line reference

the player executable also takes an untagged argument of a file name for example

```
helics_recorder record_file.txt --stop 5
```

Recorders support both delimited text files and json files some examples can be found in

Player configuration examples

config file detail

subscriptions

a simple example of a recorder file specifying some subscriptions

```
#FederateName topic1

sub pub1
subscription pub2
```

signifies a comment

if only a single column is specified it is assumed to be a subscription

for two column rows the second is the identifier arguments with spaces should be enclosed in quotes

interface	description
s, sub, subscription	subscribe to a particular publication
endpoint, ept, e	generate an endpoint to capture all targeted packets
source, sourceclone,src	capture all messages coming from a particular endpoint
dest, destination, destclone	capture all message going to a particular endpoint
capture	capture all data coming from a particular federate
clone	capture all message going from or to a particular endpoint

for 3 column rows the first must be either clone or capture for clone the second can be either source or destination and the third the endpoint name [for capture it can be either “endpoints” or “subscriptions”] NOTE:not fully working yet for capture

JSON configuration

recorders can also be specified via JSON files

here are two examples of the text format and equivalent JSON

```
#list publications and endpoints for a recorder

pub1
pub2
e src1
```

JSON example

```
{
  "subscriptions": [
    {
      "key": "pub1",
      "type": "double"
    },
    {
      "key": "pub2",
      "type": "double"
    }
  ],
  "endpoints": [
    {
      "name": "src1",
      "global": true
    }
  ]
}
```

some configuration can also be done through JSON through elements of “stop”,”local”,”separator”,”timeunits” and file elements can be used to load up additional files

output

Recorders capture files in a format the Player can read see *Player* the `--verbose` option will also print the values to the screen.

Map file output

the recorder can generate a live file that can be used in process to see the progress of the Federation This is occasionally useful, though for many uses the *Tracer* will be more useful when it is completed

4.5.2 Player

The player application is one of the HELICS apps available with the library Its purpose is to provide a easy way to generate data into a federation It acts as a federate that can "play" values or messages at specific times It exists as a standalone executable but also as library object so could be integrated into other components

Command line arguments

```
command line only:
  -? [ --help ]          produce help message
  -v [ --version ]       display a version string
  --config-file arg      specify a configuration file to use

configuration:
  --local                specify otherwise unspecified endpoints and
                         publications as local( i.e.the keys will be prepended
                         with the player name
  --stop arg             the time to stop the player
  --quiet                turn off most display output


configuration:
  -b [ --broker ] arg    address of the broker to connect
  -n [ --name ] arg      name of the player federate
  --corename arg         the name of the core to create or find
  -c [ --core ] arg      type of the core to connect to
  --offset arg           the offset of the time steps
  --period arg           the period of the federate
  --timedelta arg        the time delta of the federate
  --rttolerance arg      the time tolerance of the real time mode
  -i [ --coreinit ] arg  the core initialization string
  --separator arg        separator character for local federates
  --inputdelay arg       the input delay on incoming communication of the
                         federate
  --outputdelay arg      the output delay for outgoing communication of the
                         federate
  -f [ --flags ] arg     named flag for the federate

allowed options:

configuration:
  --datatype arg         type of the publication data type to use
  --marker arg           print a statement indicating time advancement every  arg seconds
                         is the period of the marker
  --time_units arg        the default units on the timestamps used in file based
                         input
```

also permissible are all arguments allowed for federates and any specific broker specified:

Command line reference

the player executable also takes an untagged argument of a file name for example

```
$ helics_player player_file.txt --stop 5
```

Players support both delimited text files and JSON files some examples can be found in

Player configuration examples

Config File Detail

publications

a simple example of a player file publishing values

```
#second    topic                type(opt)                    value
-1.0, pub1, d, 0.3
1, pub1, 0.5
3, pub1 0.8
2, pub1 0.7
# pub 2
1, pub2, d, 0.4
2, pub2, 0.6
3, pub2, 0.9
4, 0.7  # this statement is assumed to refer to pub 2
```

signifies a comment the first column is time in seconds unless otherwise specified via the `--time_units` flag or other configuration means the second column is publication name the final column is the value the optional third column specifies a type valid types are

time specifications are typically numerical with optional units `5` or `"500 ms"` or `23.7us` if there is a space between the number and units it must be enclosed in quotes if no units are specified the time defaults to units specified via `--time_units` or seconds if none were specified valid units are "s", "ms", "us", "min", "day", "hr", "ns", "ps" the default precision in HELICS is ns so time specified in ps is not guaranteed to be precise

identifier	type	Example
d,f, double	double	45.1
s,string	string	"this is a test"
i, i64, int	integer	456
c, complex	complex	23+2j, -23.1j, 1+3i
v, vector	vector of doubles	[23.1,34,17.2,-5.6]
cv, complex_vector	vector of complex numbers	[23+2j, -23.1j, 1+3i]

capitalization does not matter

values with times <0 are sent during the initialization phase values with time==0 are sent immediately after entering execution phase

Messages

messages are specified in one of two forms

```
m <time> <source> <dest>  <data>
```

or

```
m <sendtime> <deliverytime> <source> <dest> <time> <data>
```

the second option allows sending events at a different time than they are triggered the data portion of messages can be encoded in base64 by marking as b64[] or base64[X] all data between the brackets will be converted to raw binary. A ']' must be last. The string interpreter can also handle messages with any escapable characters including tab ("\t"),

newline (“\n”), and quote (“”), this can be marked by using quotes as in "<message>" to make it interpret the message as a JSON quoted string.

JSON configuration

player values can also be specified via JSON files

here are two examples of the text format and equivalent JSON

```
#example player file
mess 1.0 src dest "this is a test message"
mess 1.0 2.0 src dest "this is test message2"
M 2.0 3.0 src dest "this is message 3"
```

JSON example

```
{
  "messages": [
    {
      "source": "src",
      "dest": "dest",
      "time": 1.0,
      "data": "this is a test message"
    },
    {
      "source": "src",
      "dest": "dest",
      "time": 1.0,
      "encoding": "base64"
    },
    {
      "source": "src",
      "dest": "dest",
      "time": 2.0,
      "data": "this is test message 2"
    },
    {
      "source": "src",
      "dest": "dest",
      "time": 3.0,
      "data": "this is message 3"
    }
  ]
}
```

```
#second    topic               type(opt)                   value
-1 pub1 d 0.3
1 pub1 d 0.5
2 pub1 d 0.7
3 pub1 d 0.8
1 pub2 d 0.4
2 pub2 d 0.6
3 pub2 d 0.9
```

Example JSON

```
{
  "points": [
    {
      "key": "pub1",
      "type": "double",
      "value": 0.3,
      "time": -1
    },
    {
      "key": "pub2",
      "type": "double",
      "value": 0.4,
      "time": 1.0
    },
    {
      "key": "pub1",
      "value": 0.5,
      "time": 1.0
    },
    {
      "key": "pub1",
      "value": 0.8,
      "time": 3.0
    },
    {
      "key": "pub1",
      "value": 0.7,
      "time": 2.0
    },
    {
      "key": "pub2",
      "value": 0.6,
      "time": 2.0
    },
    {
      "key": "pub2",
      "value": 0.9,
      "time": 3.0
    }
  ]
}
```

some configuration can also be done through JSON through elements of “stop”,”local”,”separator”,”time_units” and file elements can be used to load up additional files

4.5.3 Source

The Source app generates signals for other federates, it functions similarly to the player but doesn't take a prescribed file instead it generates signals according to some mathematical function, like sine, ramp, pulse, or random walk. This can be useful for sending probing signals or just testing responses of the federate to various stimuli.

Command line arguments

```
allowed options:

command line only:
  -? [ --help ]          produce help message
  -v [ --version ]       display a version string
  --config-file arg      specify a configuration file to use

configuration:
  --datatype arg         type of the publication data type to use
  --local                specify otherwise unspecified endpoints and
                         publications as local( i.e.the keys will be prepended
                         with the player name
  --separator arg        specify the separator for local publications and
                         endpoints
  --time_units arg        the default units on the timestamps used in file based
                         input
  --stop arg             the time to stop the player

federate configuration
  -b [ --broker ] arg    address of the broker to connect
  -n [ --name ] arg      name of the player federate
  --corename arg         the name of the core to create or find
  -c [ --core ] arg      type of the core to connect to
  --offset arg           the offset of the time steps
  --period arg           the period of the federate
  --timedelta arg        the time delta of the federate
  -i [ --coreinit ] arg  the core initialization string
  --inputdelay arg       the input delay on incoming communication of the
                         federate
  --outputdelay arg      the output delay for outgoing communication of the
                         federate
  -f [ --flags ] arg     named flags for the federate
```

also permissible are all arguments allowed for federates and any specific broker specified:

Command line reference

4.5.4 helics_app

The HELICS apps executable is one of the HELICS apps available with the library Its purpose is to provide a common executable for running any of the other as

typical syntax is as follows

```
helics-app.exe <app> <app arguments ...>
```

possible apps are

Echo

The *Echo* app is a responsive app that will echo any message sent to its endpoints back to the original source with a specified delay

This is useful for testing communication pathways and in combination with filters can be used to create some interesting situations

Player

The *Player* app will generate signals through specified interfaces from prescribed data This is used for generating test signals into a federate

Recorder

The *Recorder* app captures signals and data on specified interfaces and can record then to various file formats including text files and JSON files The files saved can then be used by the Player app at a later time

Tracer

The *Tracer* app functions much like the recorder when run as a standalone app with the exception that it displays information to a text window and doesn't capture to a file The additional purpose is used as a library object as the basis for additional display purposes and interfaces

Source

The *Source* app is a signal generator like the player except that is can generate signals from defined patterns including some random signals in value and timing, and other patterns like sine, square wave, ramps and others. Used much like the player in situations some test signals are needed.

Broker

The *Broker* executes a broker like the stand alone Broker app, it does not include the broker terminal application.

Clone

The *Clone* has the ability to copy another federate and record it to a file that can be used by a Player. It will duplicate all publications and subscriptions of a federate.

MultiBroker

The Multibroker is an in progress development of a broker that can interact with multiple communication modes. Such as a single broker that can act as a bridge between MPI and ZeroMQ or other network protocols. More documentation will be available as the multibroker is developed

4.5.5 Command Line Arguments

```
allowed options:

command line only:
  -? [ --help ]          produce help message
  -v [ --version ]       display a version string
  --config-file arg      specify a configuration file to use

configuration:
  -n [ --name ] arg      name of the broker
  -t [ --type ] arg      type of the broker ("(zmq)", "ipc", "test", "mpi",
                         "test", "tcp", "udp")

 Help for Zero MQ Broker:
```

(continues on next page)

(continued from previous page)

```
configuration:
  --interface arg        the local interface to use for the receive ports
  -b [ --broker ] arg    identifier for the broker
  --broker_address arg   location of the broker i.e network address
  --brokerport arg       port number for the broker priority port
  --localport arg        port number for the local receive port
  --port arg             port number for the broker's port
  --portstart arg        starting port for automatic port definitions

 Help for Interprocess Broker:

configuration:
  --queueloc arg         the named location of the shared queue
  -b [ --broker ] arg    identifier for the broker
  --broker_address arg   location of the broker i.e network address
  --brokerinit arg       the initialization string for the broker

 Help for Test Broker:

configuration:
  --brokername arg       identifier for the broker-same as broker
  -b [ --broker ] arg    identifier for the broker
  --broker_address arg   location of the broker i.e network address
  --brokerinit arg       the initialization string for the broker

 Help for UDP Broker:

configuration:
  --interface arg        the local interface to use for the receive ports
  -b [ --broker ] arg    identifier for the broker
  --broker_address arg   location of the broker i.e network address
  --brokerport arg       port number for the broker priority port
  --localport arg        port number for the local receive port
  --port arg             port number for the broker's port
  --portstart arg        starting port for automatic port definitions
```

4.5.6 Echo

The Echo application is one of the HELICS apps available with the library Its purpose is to provide a easy way to generate an echo response to a message Mainly for testing and demos

Command line arguments

```
allowed options:

command line only:
  -? [ --help ]          produce help message
  -v [ --version ]       display a version string
  --config-file arg      specify a configuration file to use

configuration:
```

(continues on next page)

(continued from previous page)

```
  --local                specify otherwise unspecified endpoints and
                         publications as local( i.e.the keys will be prepended
                         with the echo name
  --stop arg             the time to stop the app


configuration:
  -b [ --broker ] arg    address of the broker to connect
  -n [ --name ] arg      name of the player federate
  --corename arg         the name of the core to create or find
  -c [ --core ] arg      type of the core to connect to
  --offset arg           the offset of the time steps
  --period arg           the period of the federate
  --timedelta arg        the time delta of the federate
  -i [ --coreinit ] arg  the core initialization string
  --separator arg        separator character for local federates
  --inputdelay arg       the input delay on incoming communication of the
                         federate
  --outputdelay arg      the output delay for outgoing communication of the
                         federate
  -f [ --flags ] arg     named flag for the federate


configuration:
  --delay arg            the delay with which the echo app will echo message
```

also permissible are all arguments allowed for federates and any specific broker specified:

Command line reference

the echo executable also takes an untagged argument of a file name for example

```
helics_app echo echo_file.txt --stop 5
```

The Echo app supports JSON files some examples can be found in

Echo configuration examples

the main property of the echo app is the delay time which messages are echoed.

4.5.7 Tracer

The Tracer application is one of the HELICS apps available with the library Its purpose is to provide a easy way to display data from a federation It acts as a federate that can "capture" values or messages from specific publications or direct endpoints or cloned endpoints which exist elsewhere and either trigger callbacks or display it to a screen The main use is a simple visual indicator and a monitoring app

Command line arguments

```
allowed options:

command line only:
  -? [ --help ]          produce help message
  -v [ --version ]       display a version string
```

(continues on next page)

(continued from previous page)

```
  --config-file arg      specify a configuration file to use

configuration:
  --stop arg             the time to stop recording
  --tags arg             tags to record, this argument may be specified any
                         number of times
  --endpoints arg        endpoints to capture, this argument may be specified
                         multiple time
  --sourceclone arg      existing endpoints to capture generated packets from,
                         this argument may be specified multiple time
  --destclone arg        existing endpoints to capture all packets with the
                         specified endpoint as a destination, this argument may
                         be specified multiple time
  --clone arg            existing endpoints to clone all packets to and from
  --capture arg          capture all the publications of a particular federate
                         capture="fed1;fed2"  supports multiple arguments or a
                         semicolon/comma separated list
  -o [ --output ] arg    the output file for recording the data
  --mapfile arg          write progress to a memory mapped file


federate configuration
  -b [ --broker ] arg    address of the broker to connect
  -n [ --name ] arg      name of the player federate
  --corename arg         the name of the core to create or find
  -c [ --core ] arg      type of the core to connect to
  --offset arg           the offset of the time steps
  --period arg           the period of the federate
  --timedelta arg        the time delta of the federate
  -i [ --coreinit ] arg  the core initialization string
  --inputdelay arg       the input delay on incoming communication of the
                         federate
  --outputdelay arg      the output delay for outgoing communication of the
                         federate
  -f [ --flags ] arg     named flags for the federate
```

also permissible are all arguments allowed for federates and any specific broker specified:

Command line reference

the tracer executable also takes an untagged argument of a file name for example

```
helics_app tracer tracer_file.txt --stop 5
```

Tracers support both delimited text files and JSON files some examples can be found in, they are otherwise the same as options for recorders.

Tracer configuration examples

Config File Detail

subscriptions

a simple example of a recorder file specifying some subscriptions

```
#FederateName topic1

sub pub1
subscription pub2
```

signifies a comment

if only a single column is specified it is assumed to be a subscription

for two column rows the second is the identifier arguments with spaces should be enclosed in quotes

interface	description
s, sub, subscription	subscribe to a particular publication
endpoint, ept, e	generate an endpoint to capture all targeted packets
source, sourceclone,src	capture all messages coming from a particular endpoint
dest, destination, destclone	capture all message going to a particular endpoint
capture	capture all data coming from a particular federate
clone	capture all message going from or to a particular endpoint

for 3 column rows the first must be either clone or capture for clone the second can be either source or destination and the third the endpoint name [for capture it can be either "endpoints" or "subscriptions"]

JSON configuration

Tracers can also be specified via JSON files

here are two examples of the text format and equivalent JSON

```
#list publications and endpoints for a recorder

pub1
pub2
e src1
```

JSON example

```
{
  "subscriptions": [
    {
      "key": "pub1",
      "type": "double"
    },
    {
      "key": "pub2",
      "type": "double"
    }
  ],
  "endpoints": [
    {
      "name": "src1",
      "global": true
    }
  ]
}
```

some configuration can also be done through JSON through elements of “stop”,”local”,”separator”,”timeunits” and file elements can be used to load up additional files

4.5.8 Broker

Brokers function as intermediaries or roots in the HELICS hierarchy The Broker can be run through the helics_broker or via helics-app

Command line arguments

```
helics_broker term <broker args...> will start a broker and open a terminal control␣
↪window for the broker run help in a terminal for more commands
helics_broker --autorestart <broker args ...> will start a continually regenerating␣
↪broker there is a 3 second countdown on broker completion to halt the program via ctrl-
↪C
helics_broker <broker args ..> just starts a broker with the given args and waits for it␣
↪to complete
allowed options:

command line only:
  -? [ --help ]          produce help message
  -v [ --version ]       display a version string
  --config-file arg      specify a configuration file to use

configuration:
  -n [ --name ] arg      name of the broker
  -t [ --type ] arg      type of the broker ("(zmq)", "ipc", "test", "mpi","test", "tcp",
↪ "udp")

 Help for Zero MQ Broker:
allowed options:

configuration:
  --interface arg        the local interface to use for the receive ports
  -b [ --broker ] arg    identifier for the broker
  --broker_address arg   location of the broker i.e network address
  --brokername arg       the name of the broker
  --local                use local interface(default)
  --ipv4                 use external ipv4 addresses
  --ipv6                 use external ipv6 addresses
  --external             use all external interfaces
  --brokerport arg       port number for the broker priority port
  --localport arg        port number for the local receive port
  --port arg             port number for the broker's port
  --portstart arg        starting port for automatic port definitions

 Help for Interprocess Broker:
allowed options:

configuration:
  --queueloc arg         the named location of the shared queue
  -b [ --broker ] arg    identifier for the broker
  --broker_address arg   location of the broker i.e network address
```

(continues on next page)

(continued from previous page)

```
  --brokerinit arg       the initialization string for the broker

 Help for Test Broker:
allowed options:

configuration:
  --brokername arg       identifier for the broker-same as broker
  -b [ --broker ] arg    identifier for the broker
  --broker_address arg   location of the broker i.e network address
  --brokerinit arg       the initialization string for the broker

 Help for TCP Broker:
allowed options:

configuration:
  --interface arg        the local interface to use for the receive ports
  -b [ --broker ] arg    identifier for the broker
  --broker_address arg   location of the broker i.e network address
  --brokername arg       the name of the broker
  --local                use local interface(default)
  --ipv4                 use external ipv4 addresses
  --ipv6                 use external ipv6 addresses
  --external             use all external interfaces
  --brokerport arg       port number for the broker priority port
  --localport arg        port number for the local receive port
  --port arg             port number for the broker's port
  --portstart arg        starting port for automatic port definitions

 Help for UDP Broker:
allowed options:

configuration:
  --interface arg        the local interface to use for the receive ports
  -b [ --broker ] arg    identifier for the broker
  --broker_address arg   location of the broker i.e network address
  --brokername arg       the name of the broker
  --local                use local interface(default)
  --ipv4                 use external ipv4 addresses
  --ipv6                 use external ipv6 addresses
  --external             use all external interfaces
  --brokerport arg       port number for the broker priority port
  --localport arg        port number for the local receive port
  --port arg             port number for the broker's port
  --portstart arg        starting port for automatic port definitions

Broker Specific options:

configuration:
  --root                 specify whether the broker is a root

configuration:
  -n [ --name ] arg      name of the broker/core
```

(continues on next page)

(continued from previous page)

```
  --federates arg        the minimum total number of federates that will be
                         connecting through this broker and any sub-brokers
  --required_federates arg
                         federate name or list of federate names that must be
                         registered before entering init mode
  --minfed arg           the minimum total number of federates that will be
                         connecting through this broker and any sub-brokers
  --maxiter arg          maximum number of iterations
  --logfile arg          the file to log message to
  --loglevel arg         the level which to log the higher this is set to the
                         more gets logs (-1) for no logging
  --fileloglevel arg     the level at which messages get sent to the file
  --consoleloglevel arg  the level at which message get sent to the console
  --minbrokers arg       the minimum number of core/brokers that need to be
                         connected (ignored in cores)
  --local_federates arg  the minimum number of federates attached through direct
                         child cores before entering init mode
  --local_subbrokers arg the minimum number of direct child brokers before
                         entering init mode
  --identifier arg       name of the core/broker
  --tick arg             number of milliseconds per tick counter if there is no
                         broker communication for 2 ticks then secondary actions
                         are taken (can also be entered as a time like '10s' or '45ms')
  --dumplog              capture a record of all messages and dump a complete log to␣
↪file or console on termination
  --terminate_on_error   Specify that the co-simulation should terminate if any error␣
↪occurs
  --timeout arg          milliseconds to wait for a broker connection (can also
                         be entered as a time like '10s' or '45ms')

  --error_timeout arg    milliseconds to wait before disconnecting after an error
                         (can also be entered as a time like '10s' or '45ms')
```

If the Broker is started with `term` as the first option, a terminal is opened for user entry of commands all command line arguments following term are passed to the broker.

```
starting broker
helics>>help
`quit` -> close the terminal application and wait for broker to finish
`terminate` -> force the broker to stop
`terminate*` -> force the broker to stop and exit application
`help`,`?` -> this help display
`restart` -> restart a completed broker
`status` -> will display the current status of the broker
`info` -> will display info about the broker
`force restart` -> will force terminate a broker and restart it
`query` <queryString> -> will query a broker for <queryString>
`query` <queryTarget> <queryString> -> will query <queryTarget> for <queryString>
helics>>
```

`status` will print out current status of the brokers including counts of federates, brokers, and handles

```
helics>>status
Broker (643204-ibrVd-14EWH-unKfh-hExUP) is connected and is accepting new federates
{"brokers":0,
"federates":0,
"handles":0}
helics>>
```

info prints out name, connection status, and connection information

```
helics>>info
Broker (643204-ibrVd-14EWH-unKfh-hExUP) is connected and is accepting new federates
address=tcp://127.0.0.1:23404
```

The `query` command allows any query to be executed from the command line, `query counts` displays the same count numbers as `status`.

Other available queries are described in *Queries*.

various restart options are also available, `terminate`, `restart`, `force restart`. And finally `quit` will exit the terminal and wait for the broker to complete. enter `terminate` before quit or `terminate*` to terminate and quit.

4.5.9 Broker Server

Brokers function as intermediaries or roots in the HELICS hierarchy The broker server is an executable that can be used to automatically generate brokers on an as needed basis and coordinate their control and management. It is considered experimental as version 2.2 only works with the ZMQ core type. Future versions will expand this significantly.

Future plans include expanding to all networking core types (ZMQ, ZMQSS, TCP, TCPSS, UDP, and MPI), expanding the abilities of a terminal program and making a Restful interface to the server and underlying brokers.

Command line arguments

```
The Broker server is a helics broker coordinator that can generate brokers on request
Usage:helics_broker_server [OPTIONS] [config]

Positionals:
  config TEXT                 load a config file for the broker server

Options:
  -h,-?,--help                Print this help message and exit
  -v,--version
  -z,--zmq                    start a broker-server for the zmq comms in helics
  --zmqss                     start a broker-server for the zmq single socket comms in␣
↪helics
  -t,--tcp                    start a broker-server for the tcp comms in helics
  -u,--udp                    start a broker-server for the udp comms in helics
  --mpi                       start a broker-server for the mpi comms in helics
[Option Group: quiet]
  Options:
    --quiet                     silence most print output

helics broker server command line
helics_broker_server [OPTIONS] [SUBCOMMAND]
```

(continues on next page)

(continued from previous page)

```
Options:
  -h,-?,--help                Print this help message and exit
  -v,--version
  --duration TIME=30 minutes  specify the length of time the server should run
[Option Group: quiet]
  Options:
    --quiet                     silence most print output

Subcommands:
  term                        helics_broker_server term will start a broker server and␣
↪open a terminal control window for the broker server, run help in a terminal for more␣
↪commands


helics_broker_server server types starts a broker with the given args and waits for it␣
↪to complete
```

If the Broker_server is started with `term` as the first option, a terminal is opened for user entry of commands all command line arguments following term are passed to the broker.

```
starting broker Server
servers started
helics-broker-server>>help
`quit` -> close the terminal application and wait for broker to finish
`terminate` -> force the broker server to stop
`terminate*` -> force the broker server to stop and all existing brokers to terminate
`help`,`?` -> this help display

helics-broker-server>>
```

more commands will be added in future releases

4.5.10 Clone

The Clone application is one of the HELICS apps available with the library Its purpose is to provide a easy way to clone a federate for later playback It acts as a federate that can "capture" values or messages from a single federate It also captures the interfaces and subscriptions of a federate and will store those in a configuration file that can be used by the *Player*. The clone app will try to match the federate being cloned as close as possible in timing of messages and publications and subscriptions. At present it does not match nameless publications or filters.

Command line arguments

```
Helics Clone App
Usage: helics_app clone [OPTIONS]

Command line options for the Clone App
Usage: [OPTIONS] [capture]

Positionals:
  capture TEXT                name of the federate to clone

Options:
```

(continues on next page)

(continued from previous page)

```
  --allow_iteration           allow iteration on values
  -o,--output TEXT=clone.json the output file for recording the data
  --capture TEXT                name of the federate to clone

Options:
  -h,-?,--help                Print this help message and exit
```

also permissible are all arguments allowed for federates and any specific broker specified:

Command line reference

the clone app is accessible through the helics_app

```
helics_app clone fed1 -o fed1.json -stop 10
```

output

The Clone app captures output and configuration in a JSON format the *Player* can read. All publications of a federate are created as global with the name of the original federate, so a player could be named something else if desired and not impact the transmission.

4.5.11 Connector

The Connector app can automatically connect interfaces together. It does this using the query mechanisms inside HELICS to detect all the unconnected interfaces in a co-simulation. Then, using given configuration rules, it will establish connections between those interfaces.

It can also run in a two-phase mode to have the federates create then connect interfaces. In the first phase, the connector app will query the federates for their potential interfaces and then go through those interfaces to see if there is a potential connection to be made using the same rules. If a potential interface has a connection available, it will send a command to the appropriate federate to create the interface. Once the interfaces exist, the Connector enters the second phase and makes the requested connections from the new and existing unconnected interfaces.

Connector configuration

The main mechanism to load connection information is through configuration files called "match-files", typically a plain text file. The format is:

`<origin> <target> <*direction> <*tags...>`

"origin" is the interface that is currently unconnected and "target" is the interface to connect it to. "direction" is optional and is assumed to be bidirectional matching ("bi"); in this case the direction of the match does not indicate the flow of the data but rather which interface is unmatched (the "from"). For example, if the match string looked like "V_out, V_in from_to" the Connector treats "V_out" as unconnected and will match it with "V_in" even if "V_in" is already connected. Bidirectional matching allows either interface to be unconnected to create a match.

"tags" are optional and allow for filtering the candidate connections; see below for further details on their use. Comments lines are supported and begin with #. Currently only publications, inputs, and endpoints are supported for matching by Connector.

Here's a simple example of a plain text match-file.

```
#comment line for simple file test
inp1 pub1 from_to
```

The following example uses a cascaded matching definition.

```
# comment line for cascade file test
# second comment line
inp1 intermediate1
intermediate1 intermediate1 bi
intermediate1 intermediate2 from_to
intermediate2 intermediate3 from_to
# comment line in the middle
intermediate3 pub1 bi
inp2 intermediate2 from_to
publication3 input3 to_from
```

The following example uses tags in the matching process, see more on tags below.

```
#comment line for simple file test
inp1 pub1 from_to tag1
inp2 pub1 tag2 tag3
```

The match-file can also be JSON formatted.

```
{
  "connections": [
    ["inp1", "pub1", "FROM_TO", "tag1"],
    ["inp2", "pub2", "tag2", "tag3"]
  ]
}
```

Notes on Tags

- Connections specified with no tags or "default" tag will match with everything as if the tag were not there. If a connection specified by a match in the match-file uses a tag, a connection will only be made if the specified tag is used by a federate, core, or broker.
- A tag can be specified by a "global_value". The tag used for the connector is the name of the global or tag and the value can be anything other than a "false" value; if the tag is specified with a "false" value it is not used in the matching. Tags used in the match-file can also be specified in the value of the "tags" global or local tag. In this case they are specified with a comma separated list. The complete list of "false" valued strings is as follows:

```
    "0",        "",         "false",
   "False",    "FALSE",    "off",
   "Off",      "OFF",      "disabled",
   "Disabled", "DISABLED", "disable",
   "Disable",  "DISABLE",  "f",
   "F",        "0",        std::string_view(reinterpret_cast<const char*>(&
↪nullstringRep), 1),
   " ",        "no",       "NO",
   "No",       "-"
```

Regular Expression (regex) Matching

In addition to directly defining the connections to be made between interfaces, one-by-one, it is also possible to use regular expressions to define the connections. Regular expressions allow a large number of similarly-name interfaces to be matched with a single statement. The file is formatted as follows:

```
REGEX:pub_num_(?<interface_num>\d*)_(?<alpha_index>[A-Za-z]*), REGEX:input_num_(?
↪<interface_num>\d*)_(?<alpha_index>[A-Za-z]*)
```

In this example, "interface_num" and "alpha_index" are user-defined strings that gives a name to the portion of the regex that needs to match. The funny stuff immediately after it ("\d*", "[A-Za-z]*") is the regular expression proper that the regular expression will use to determine how to make matches. Using the above example, the following matches would be made:

```
pub_num_1_a input_num_1_a
pub_num_204_voltage input_num_204_voltage
```

Writing regular expressions quickly and accurately is a learned skill and depending the names of the interfaces, it can be difficult to craft one that does exactly what you need. The use of tags may be helpful in preventing matches between federates when they are not needed. Additionally, it may be easier to write a regular expression that makes most of the matches you need and then use direct matches for the remainder.

Interface Creation and Matching

As mentioned in the introduction, it is also possible for the Connector app to interact with federates that are created with no exposed interfaces and jointly work through a process where those interfaces are created and then connected. A Python example of this process in action can be found in the HELICS Examples repository but a conceptual overivew of the process is as follows:

Federate creation

On launch of the federation, the federates are created with no exposed interfaces BUT with an understanding of what interfaces it can create. These interfaces may, for example, be hard-coded or based on the system model it reads on start-up.

Interface Query

The Connector queries the federates to determine which interfaces each one can create. The query is made after the federate enter initializing mode and the federate must enter initializing mode iteratively (`helicsFederateEnterInitializingModeIterative()`) to synchronize the query responses across the federation. Every federate that is going to create interfaces needs to register a callback function to handle this custom query by the Connector and respond appropriately. The Connector will query the federate with "potential_interfaces" and the federate must respond with a properly formatted JSON:

```
{
  "publications": [<list of names of publications that can be created>]
  "inputs": [<list of names of inputs that can be created>]
  "endpoints": [<list of names of endpoints that can be created>]
}
```

As this is a query operation, which are executed asynchronously with the simulation time, it is undefined when the query will be made and thus a callback function must be used to respond to the query.

Connector Interface Creation Command

After receiving the query responses from all the federates, the connector performs its standard matching operation using a match-file. Once the matches are made, it determines which connections need to be made and sends a command to each federate telling it which interfaces to create. As with the query, the commands are received asynchronously but are guaranteed to be present after calling `helicsFederateEnterInitializingModeIterative()` twice. At that

point, the federate can get the command and parse the returned JSON to determine which interfaces to create. The format is the same as the query response:

```
{
  "publications": [<list of names of publications to be created>]
  "inputs": [<list of names of inputs to be created>]
  "endpoints": [<list of names of endpoints to be created>]
}
```

Interface Creation and Co-Simulation Execution

The federate takes the JSON command and, using its own internal knowledge of the interface (global or not, data type, units) and creates the interfaces. After that, the federate doesn't need to do anything else for the interface connections to be connected and can call `helicsFederateEnterExecutingMode()` (assuming it has nothing else to do as a part of initializing). The Connector will make the connections between the interfaces as they are created and when complete, exit the federation.

Use of the Connector

To use the Connector to create the interface connections, simply call it as part of your federation, adding the matchfile as a command-line argument. The connector app will start up when the federation is launched and, using the match-file, create the connections between interfaces behind the scenes. Once the work it complete (by the "execution" mode of the federation), it exits the federation and allows now connected federates to proceed. A sample call looks like:

```
helics_connector matchfile.txt
```

Command line arguments

Options specific to the connector are as follows:

```
Options:
  --version                   Display program version information and exit
  --connection [INTERFACE1,INTERFACE2,DIRECTIONALITY,TXT...] ...
                              specify connections to make in the cosimulation
  --match_target_endpoints    set to true to enable connection of unconnected target␣
↪endpoints
  --match_multiple            set to true to enable matching of multiple connections␣
↪(default false)
  --always_check_regex        set to true to enable regex matching even if other matches␣
↪are defined
```

The full CLI list is shown below including helics connection options and general options.

```
Common options for all Helics Apps
Usage: [HELICS_APP] [OPTIONS] [input]

Positionals:
  input TEXT:FILE             The primary input file

Options:
  -h,-?,--help                Print this help message and exit
  --config-file,--config [helics_config.toml]
                              specify base configuration file
  --version                   Display program version information and exit
```

(continues on next page)

(continued from previous page)

```
  --local                     Specify otherwise unspecified endpoints and publications␣
↪as local (i.e. the names will be prepended with the player name)
  --stop TIME                 The time to stop the app
  --input TEXT:FILE           The primary input file
[Option Group: quiet]
  Options:
    --quiet                   silence most print output
[Option Group: Subcommands]
  Federate Info Parsing
  Positionals:
    config [helicsConfig.ini]   specify a configuration file
  Options:
    --version                 Display program version information and exit
    --config-file,--config [helicsConfig.ini]
                              specify a configuration file
    --config_section TEXT     specify the section of the config file to use
    --config_index INT        specify the section index of the config file to use for␣
↪configuration arrays
    -n,--name TEXT            name of the federate
    --corename TEXT           the name of the core to create or find
    -i,--coreinitstring TEXT (Env:HELICS_CORE_INIT_STRING)
                              The initialization arguments for the core
    --brokerinitstring TEXT   The initialization arguments for the broker if␣
↪autogenerated
    --broker,--brokeraddress TEXT
                              address or name of the broker to connect
    --brokerport INT:POSITIVE   Port number of the Broker
    --port INT:POSITIVE       Specify the port number to use
    --localport TEXT          Port number to use for connections to this federate
    --autobroker              tell the core to automatically generate a broker if␣
↪needed
    --debugging               tell the core to allow user debugging in a nicer fashion
    --observer                tell the federate/core that this federate is an observer
    --allow_remote_control,--disable_remote_control{false}
                              enable the federate to respond to certain remote␣
↪operations such as disconnect
    --json                    tell the core and federate to use JSON based␣
↪serialization for all messages, to ensure compatibility
    --profiler TEXT [log]     Enable profiling and specify a file name (NOTE: use --
↪profiler_append=<filename> in the core init string to append to an existing file)
    --broker_key,--brokerkey,--brokerKey TEXT
                              specify a key to use to match a broker should match the␣
↪broker key
    --offset TIME             the offset of the time steps (default in ms)
    --period TIME             the execution cycle of the federate (default in ms)
    --stoptime TIME           the maximum simulation time of a federate (default in ms)
    --timedelta TIME          The minimum time between time grants for a Federate␣
↪(default in ms)
    --inputdelay TIME         the INPUT delay on incoming communication of the␣
↪federate (default in ms)
    --outputdelay TIME        the output delay for outgoing communication of the␣
↪federate (default in ms)
```

(continues on next page)

(continued from previous page)

```
    --grant_timeout TIME        timeout to trigger diagnostic action when a federate␣
↪time grant is not available within the timeout period (default in ms)
    --maxiterations INT:POSITIVE
                                the maximum number of iterations a federate is allowed␣
↪to take
    --loglevel INT:{summary,none,connections,no_print,profiling,interfaces,error,timing,
↪warning,data,debug,trace}:value in {summary->6,none->-4,connections->9,no_print->-4,
↪profiling->2,interfaces->12,error->0,timing->15,warning->3,data->18,debug->21,trace->
↪24} OR {6,-4,9,-4,2,12,0,15,3,18,21,24} (Env:HELICS_LOG_LEVEL)
                                the logging level of a federate
    --separator CHAR [/]        separator character for local federates
    -f,--flags,--flag ...       named flag for the federate
  [Option Group: quiet]
    Options:
      --quiet                     silence most print output
  [Option Group: network type]
    Options:
      --core TEXT [()]            type or name of the core to connect to
      --force_new_core            if set to true will force the federate to generate a␣
↪new core
      -t,--coretype TEXT [()]  (Env:HELICS_CORE_TYPE)
                                  type  of the core to connect to
  [Option Group: encryption]
    options related to encryption
    Options:
      --encrypted (Env:HELICS_ENCRYPTION)
                                  enable encryption on the network
      --encryption_config TEXT (Env:HELICS_ENCRYPTION_CONFIG)
                                  set the configuration file for encryption options
  [Option Group: realtime]
    Options:
      --rtlag TIME                the amount of the time the federate is allowed to lag␣
↪realtime before corrective action is taken (default in ms)
      --rtlead TIME               the amount of the time the federate is allowed to lead␣
↪realtime before corrective action is taken (default in ms)
      --rttolerance TIME          the time tolerance of the real time mode (default in␣
↪ms)
Command line options for the Connector App
Usage: [OPTIONS]

Options:
  --version                   Display program version information and exit
  --connection [INTERFACE1,INTERFACE2,DIRECTIONALITY,TXT...] ...
                              specify connections to make in the cosimulation
  --match_target_endpoints    set to true to enable connection of unconnected target␣
↪endpoints
  --match_multiple            set to true to enable matching of multiple connections␣
↪(default false)
  --always_check_regex        set to true to enable regex matching even if other matches␣
↪are defined
```

also permissible are all arguments allowed for federates and any specific broker specified:

Command line reference

Federate Support

The federate object has support for linking with connector operation. Interface definitions placed in a "potential_interfaces" object in a json configuration file will activate the potential interfaces sequence automatically. And based on the response of the connector will automatically create the interfaces defined if they are used. The interface objects can later be retrieved through normal means. No additional sequence or callbacks is needed on the federate. Json configuration is currently the only means to trigger this feature. The definitions for the interfaces in the potential interfaces are exactly the same as normally defining an interface in json.

interface Templates

The federate object and the connector also support interface definition templates, for example

```
{
  "potential_interfaces": {
    "endpoint_templates": [
      {
        "name": "obj${number}/ept${letter}/type${letter}/mode${letter}",
        "number": ["1", "2", "3", "4", "5", "6", "7", "8", "9", "0"],
        "letter": [
          "A",
          "B",
          "C",
          "D",
          "E",
          "F",
          "G",
          "H",
          "I",
          "J",
          "K",
          "L",
          "M",
          "N",
          "O",
          "P",
          "Q",
          "R",
          "S",
          "T",
          "U",
          "V",
          "W",
          "X",
          "Y",
          "Z"
        ],
        "template": { "global": true }
      }
    ]
  }
}
```

or

```
{
  "potential_interfaces": {
    "publications": [
      { "name": "pub1", "global": true, "type": "double" },
      { "name": "pub2", "global": true, "type": "double" }
    ],
    "inputs": [
      {
        "name": "inp2",
        "global": true,
        "type": "double",
        "target": "publicationB"
      },
      {
        "name": "inp1",
        "global": true,
        "type": "double",
        "target": "publicationA"
      }
    ],
    "publication_templates": [
      {
        "name": "${field1}/${field2}",
        "field1": ["obj1", "obj2", "obj3"],
        "field2": [
          ["type1", "double"],
          ["type2", "int"],
          ["type3", "double"]
        ],
        "template": { "global": true }
      }
    ],
    "input_templates": [
      {
        "name": "${field1}/${field2}",
        "field1": ["objA", "objB", "objC"],
        "field2": [
          ["typeA", "double", "W"],
          ["typeB", "int"],
          ["typeC", "double", "kV"]
        ],
        "template": { "global": true }
      }
    ]
  }
}
```

The connector will evaluate all possibilities for the template for possible connections. The field name in `${fieldName}` is searched for in the json file. They may be duplicated, but are treated as independent for evaluation purposes as in the first example. The first template example defines over 175,000 different possible interfaces. The type of the interface can be defined as part of the template name, with the particular name as the key.

For potential interfaces if they specify a targeting field, such as “target”,”targets”, “input_target(s)”, or other valid

targeting fields the potential interface will be created if the target exists or is itself a potential interface or template. The connections will then be made at the creation of the interface.

CHAPTER

FIVE

QUICK LINKS

INDEX

H

HELICS_CORE_TYPE_DEFAULT (*C++ enumerator*), 308
HELICS_CORE_TYPE_EMPTY (*C++ enumerator*), 309
HELICS_CORE_TYPE_EXTRACT (*C++ enumerator*), 309
HELICS_CORE_TYPE_HTTP (*C++ enumerator*), 308
HELICS_CORE_TYPE_INPROC (*C++ enumerator*), 309
HELICS_CORE_TYPE_INTERPROCESS (*C++ enumerator*), 308
HELICS_CORE_TYPE_IPC (*C++ enumerator*), 308
HELICS_CORE_TYPE_MPI (*C++ enumerator*), 308
HELICS_CORE_TYPE_NNG (*C++ enumerator*), 308
HELICS_CORE_TYPE_NULL (*C++ enumerator*), 309
HELICS_CORE_TYPE_TCP (*C++ enumerator*), 308
HELICS_CORE_TYPE_TCP_SS (*C++ enumerator*), 308
HELICS_CORE_TYPE_TEST (*C++ enumerator*), 308
HELICS_CORE_TYPE_UDP (*C++ enumerator*), 308
HELICS_CORE_TYPE_WEBSOCKET (*C++ enumerator*), 309
HELICS_CORE_TYPE_ZMQ (*C++ enumerator*), 308
HELICS_CORE_TYPE_ZMQ_SS (*C++ enumerator*), 308
HELICS_DATA_TYPE_ANY (*C++ enumerator*), 310
HELICS_DATA_TYPE_BOOLEAN (*C++ enumerator*), 309
HELICS_DATA_TYPE_CHAR (*C++ enumerator*), 310
HELICS_DATA_TYPE_COMPLEX (*C++ enumerator*), 309
HELICS_DATA_TYPE_COMPLEX_VECTOR (*C++ enumerator*), 309
HELICS_DATA_TYPE_DOUBLE (*C++ enumerator*), 309
HELICS_DATA_TYPE_INT (*C++ enumerator*), 309
HELICS_DATA_TYPE_JSON (*C++ enumerator*), 310
HELICS_DATA_TYPE_MULTI (*C++ enumerator*), 310
HELICS_DATA_TYPE_NAMED_POINT (*C++ enumerator*), 309
HELICS_DATA_TYPE_RAW (*C++ enumerator*), 310
HELICS_DATA_TYPE_STRING (*C++ enumerator*), 309
HELICS_DATA_TYPE_TIME (*C++ enumerator*), 309
HELICS_DATA_TYPE_UNKNOWN (*C++ enumerator*), 309
HELICS_DATA_TYPE_VECTOR (*C++ enumerator*), 309
HELICS_ERROR_CONNECTION_FAILURE (*C++ enumerator*), 314
HELICS_ERROR_DISCARD (*C++ enumerator*), 314
HELICS_ERROR_EXECUTION_FAILURE (*C++ enumerator*), 313
HELICS_ERROR_EXTERNAL_TYPE (*C++ enumerator*), 313
HELICS_ERROR_FATAL (*C++ enumerator*), 313
HELICS_ERROR_INSUFFICIENT_SPACE (*C++ enumerator*), 313
HELICS_ERROR_INVALID_ARGUMENT (*C++ enumerator*), 314
HELICS_ERROR_INVALID_FUNCTION_CALL (*C++ enumerator*), 313
HELICS_ERROR_INVALID_OBJECT (*C++ enumerator*), 314
HELICS_ERROR_INVALID_STATE_TRANSITION (*C++ enumerator*), 313
HELICS_ERROR_OTHER (*C++ enumerator*), 313
HELICS_ERROR_REGISTRATION_FAILURE (*C++ enumerator*), 314
HELICS_ERROR_SYSTEM_FAILURE (*C++ enumerator*), 313
HELICS_ERROR_TERMINATED (*C++ enumerator*), 314
HELICS_ERROR_USER_ABORT (*C++ enumerator*), 314
HELICS_FILTER_TYPE_CLONE (*C++ enumerator*), 317
HELICS_FILTER_TYPE_CUSTOM (*C++ enumerator*), 317
HELICS_FILTER_TYPE_DELAY (*C++ enumerator*), 317
HELICS_FILTER_TYPE_FIREWALL (*C++ enumerator*), 317
HELICS_FILTER_TYPE_RANDOM_DELAY (*C++ enumerator*), 317
HELICS_FILTER_TYPE_RANDOM_DROP (*C++ enumerator*), 317
HELICS_FILTER_TYPE_REROUTE (*C++ enumerator*), 317
HELICS_FLAG_ALLOW_REMOTE_CONTROL (*C++ enumerator*), 312
HELICS_FLAG_AUTOMATED_TIME_REQUEST (*C++ enumerator*), 311
HELICS_FLAG_CALLBACK_FEDERATE (*C++ enumerator*), 311
HELICS_FLAG_DEBUGGING (*C++ enumerator*), 312
HELICS_FLAG_DELAY_INIT_ENTRY (*C++ enumerator*), 311
HELICS_FLAG_DISABLE_REMOTE_CONTROL (*C++ enumerator*), 312

HELICS_FLAG_DUMPLOG (C++ enumerator), 312
HELICS_FLAG_ENABLE_INIT_ENTRY (C++ enumerator), 311
HELICS_FLAG_EVENT_TRIGGERED (C++ enumerator), 311
HELICS_FLAG_FORCE_LOGGING_FLUSH (C++ enumerator), 312
HELICS_FLAG_FORWARD_COMPUTE (C++ enumerator), 310
HELICS_FLAG_IGNORE (C++ enumerator), 311
HELICS_FLAG_IGNORE_TIME_MISMATCH_WARNINGS (C++ enumerator), 311
HELICS_FLAG_INTERRUPTIBLE (C++ enumerator), 310
HELICS_FLAG_LOCAL_PROFILING_CAPTURE (C++ enumerator), 311
HELICS_FLAG_MULTI_THREAD_CORE (C++ enumerator), 311
HELICS_FLAG_OBSERVER (C++ enumerator), 310
HELICS_FLAG_ONLY_TRANSMIT_ON_CHANGE (C++ enumerator), 310
HELICS_FLAG_ONLY_UPDATE_ON_CHANGE (C++ enumerator), 310
HELICS_FLAG_PROFILING (C++ enumerator), 312
HELICS_FLAG_PROFILING_MARKER (C++ enumerator), 312
HELICS_FLAG_REALTIME (C++ enumerator), 311
HELICS_FLAG_REENTRANT (C++ enumerator), 311
HELICS_FLAG_RESTRICTIVE_TIME_POLICY (C++ enumerator), 310
HELICS_FLAG_ROLLBACK (C++ enumerator), 310
HELICS_FLAG_SINGLE_THREAD_CORE (C++ enumerator), 311
HELICS_FLAG_SINGLE_THREAD_FEDERATE (C++ enumerator), 311
HELICS_FLAG_SLOW_RESPONDING (C++ enumerator), 312
HELICS_FLAG_SOURCE_ONLY (C++ enumerator), 310
HELICS_FLAG_STRICT_CONFIG_CHECKING (C++ enumerator), 311
HELICS_FLAG_TERMINATE_ON_ERROR (C++ enumerator), 312
HELICS_FLAG_UNINTERRUPTIBLE (C++ enumerator), 310
HELICS_FLAG_USE_JSON_SERIALIZATION (C++ enumerator), 311
HELICS_FLAG_WAIT_FOR_CURRENT_TIME_UPDATE (C++ enumerator), 310
HELICS_HANDLE_OPTION_BUFFER_DATA (C++ enumerator), 316
HELICS_HANDLE_OPTION_CLEAR_PRIORITY_LIST (C++ enumerator), 317
HELICS_HANDLE_OPTION_CONNECTION_OPTIONAL (C++ enumerator), 316
HELICS_HANDLE_OPTION_CONNECTION_REQUIRED (C++ enumerator), 316
HELICS_HANDLE_OPTION_CONNECTIONS (C++ enumerator), 317
HELICS_HANDLE_OPTION_IGNORE_INTERRUPTS (C++ enumerator), 317
HELICS_HANDLE_OPTION_IGNORE_UNIT_MISMATCH (C++ enumerator), 316
HELICS_HANDLE_OPTION_INPUT_PRIORITY_LOCATION (C++ enumerator), 317
HELICS_HANDLE_OPTION_MULTI_INPUT_HANDLING_METHOD (C++ enumerator), 317
HELICS_HANDLE_OPTION_MULTIPLE_CONNECTIONS_ALLOWED (C++ enumerator), 316
HELICS_HANDLE_OPTION_ONLY_TRANSMIT_ON_CHANGE (C++ enumerator), 316
HELICS_HANDLE_OPTION_ONLY_UPDATE_ON_CHANGE (C++ enumerator), 317
HELICS_HANDLE_OPTION_RECEIVE_ONLY (C++ enumerator), 316
HELICS_HANDLE_OPTION_RECONNECTABLE (C++ enumerator), 316
HELICS_HANDLE_OPTION_SINGLE_CONNECTION_ONLY (C++ enumerator), 316
HELICS_HANDLE_OPTION_SOURCE_ONLY (C++ enumerator), 316
HELICS_HANDLE_OPTION_STRICT_TYPE_CHECKING (C++ enumerator), 316
HELICS_HANDLE_OPTION_TIME_RESTRICTED (C++ enumerator), 317
HELICS_ITERATION_REQUEST_ERROR (C++ enumerator), 306
HELICS_ITERATION_REQUEST_FORCE_ITERATION (C++ enumerator), 306
HELICS_ITERATION_REQUEST_HALT_OPERATIONS (C++ enumerator), 306
HELICS_ITERATION_REQUEST_ITERATE_IF_NEEDED (C++ enumerator), 306
HELICS_ITERATION_REQUEST_NO_ITERATION (C++ enumerator), 306
HELICS_ITERATION_RESULT_ERROR (C++ enumerator), 307
HELICS_ITERATION_RESULT_HALTED (C++ enumerator), 307
HELICS_ITERATION_RESULT_ITERATING (C++ enumerator), 307
HELICS_ITERATION_RESULT_NEXT_STEP (C++ enumerator), 307
HELICS_LOG_LEVEL_CONNECTIONS (C++ enumerator), 313
HELICS_LOG_LEVEL_DATA (C++ enumerator), 313
HELICS_LOG_LEVEL_DEBUG (C++ enumerator), 313
HELICS_LOG_LEVEL_DUMPLOG (C++ enumerator), 312
HELICS_LOG_LEVEL_ERROR (C++ enumerator), 312
HELICS_LOG_LEVEL_INTERFACES (C++ enumerator),

313
HELICS_LOG_LEVEL_NO_PRINT (C++ enumerator), 312
HELICS_LOG_LEVEL_PROFILING (C++ enumerator), 312
HELICS_LOG_LEVEL_SUMMARY (C++ enumerator), 312
HELICS_LOG_LEVEL_TIMING (C++ enumerator), 313
HELICS_LOG_LEVEL_TRACE (C++ enumerator), 313
HELICS_LOG_LEVEL_WARNING (C++ enumerator), 312
HELICS_MULTI_INPUT_AND_OPERATION (C++ enumerator), 315
HELICS_MULTI_INPUT_AVERAGE_OPERATION (C++ enumerator), 316
HELICS_MULTI_INPUT_DIFF_OPERATION (C++ enumerator), 316
HELICS_MULTI_INPUT_MAX_OPERATION (C++ enumerator), 316
HELICS_MULTI_INPUT_MIN_OPERATION (C++ enumerator), 316
HELICS_MULTI_INPUT_NO_OP (C++ enumerator), 315
HELICS_MULTI_INPUT_OR_OPERATION (C++ enumerator), 315
HELICS_MULTI_INPUT_SUM_OPERATION (C++ enumerator), 315
HELICS_MULTI_INPUT_VECTORIZE_OPERATION (C++ enumerator), 315
HELICS_PROPERTY_INT_CONSOLE_LOG_LEVEL (C++ enumerator), 315
HELICS_PROPERTY_INT_CURRENT_ITERATION (C++ enumerator), 315
HELICS_PROPERTY_INT_FILE_LOG_LEVEL (C++ enumerator), 315
HELICS_PROPERTY_INT_INDEX_GROUP (C++ enumerator), 315
HELICS_PROPERTY_INT_LOG_BUFFER (C++ enumerator), 315
HELICS_PROPERTY_INT_LOG_LEVEL (C++ enumerator), 315
HELICS_PROPERTY_INT_MAX_ITERATIONS (C++ enumerator), 315
HELICS_PROPERTY_INT_VALUE_BUFFER_WARNING (C++ enumerator), 315
HELICS_PROPERTY_TIME_DELTA (C++ enumerator), 314
HELICS_PROPERTY_TIME_GRANT_TIMEOUT (C++ enumerator), 315
HELICS_PROPERTY_TIME_INPUT_DELAY (C++ enumerator), 314
HELICS_PROPERTY_TIME_OFFSET (C++ enumerator), 314
HELICS_PROPERTY_TIME_OUTPUT_DELAY (C++ enumerator), 314
HELICS_PROPERTY_TIME_PERIOD (C++ enumerator), 314
HELICS_PROPERTY_TIME_RT_LAG (C++ enumerator), 314
HELICS_PROPERTY_TIME_RT_LEAD (C++ enumerator), 314
HELICS_PROPERTY_TIME_RT_TOLERANCE (C++ enumerator), 314
HELICS_PROPERTY_TIME_STOPTIME (C++ enumerator), 315
HELICS_SEQUENCING_MODE_DEFAULT (C++ enumerator), 318
HELICS_SEQUENCING_MODE_FAST (C++ enumerator), 318
HELICS_SEQUENCING_MODE_ORDERED (C++ enumerator), 318
HELICS_STATE_ERROR (C++ enumerator), 307
HELICS_STATE_EXECUTION (C++ enumerator), 307
HELICS_STATE_FINALIZE (C++ enumerator), 307
HELICS_STATE_FINISHED (C++ enumerator), 308
HELICS_STATE_INITIALIZATION (C++ enumerator), 307
HELICS_STATE_PENDING_EXEC (C++ enumerator), 307
HELICS_STATE_PENDING_FINALIZE (C++ enumerator), 307
HELICS_STATE_PENDING_INIT (C++ enumerator), 307
HELICS_STATE_PENDING_ITERATIVE_TIME (C++ enumerator), 307
HELICS_STATE_PENDING_TIME (C++ enumerator), 307
HELICS_STATE_STARTUP (C++ enumerator), 307
HELICS_STATE_UNKNOWN (C++ enumerator), 307
HELICS_TRANSLATOR_TYPE_BINARY (C++ enumerator), 318
HELICS_TRANSLATOR_TYPE_CUSTOM (C++ enumerator), 317
HELICS_TRANSLATOR_TYPE_JSON (C++ enumerator), 318
HELICS_USER_EXCEPTION (C++ enumerator), 313
helicsAbort (C++ function), 319
helicsAppDestroy (C++ function), 335
helicsAppEnabled (C++ function), 334
helicsAppFinalize (C++ function), 335
helicsAppFree (C++ function), 335
helicsAppGetFederate (C++ function), 334
helicsAppInitialize (C++ function), 334
helicsAppIsActive (C++ function), 335
helicsAppLoadFile (C++ function), 334
helicsAppRun (C++ function), 335
helicsAppRunTo (C++ function), 335
helicsBrokerAddAlias (C++ function), 328
helicsBrokerAddDestinationFilterToEndpoint (C++ function), 326
helicsBrokerAddSourceFilterToEndpoint (C++ function), 326
helicsBrokerClearTimeBarrier (C++ function), 329
helicsBrokerClone (C++ function), 326
helicsBrokerDataLink (C++ function), 326

helicsBrokerDestroy (*C++ function*), 327
helicsBrokerDisconnect (*C++ function*), 327
helicsBrokerFree (*C++ function*), 327
helicsBrokerGetAddress (*C++ function*), 327
helicsBrokerGetIdentifier (*C++ function*), 327
helicsBrokerGlobalError (*C++ function*), 329
helicsBrokerIsConnected (*C++ function*), 326
helicsBrokerIsValid (*C++ function*), 326
helicsBrokerMakeConnections (*C++ function*), 327
helicsBrokerSendCommand (*C++ function*), 328
helicsBrokerSendOrderedCommand (*C++ function*), 328
helicsBrokerSetGlobal (*C++ function*), 327
helicsBrokerSetLogFile (*C++ function*), 328
helicsBrokerSetLoggingCallback (*C++ function*), 329
helicsBrokerSetTimeBarrier (*C++ function*), 328
helicsBrokerWaitForDisconnect (*C++ function*), 327
helicsCallbackFederateInitializeCallback (*C++ function*), 321
helicsCallbackFederateNextTimeCallback (*C++ function*), 320
helicsCallbackFederateNextTimeIterativeCallback (*C++ function*), 321
helicsCleanupLibrary (*C++ function*), 320
helicsClearSignalHandler (*C++ function*), 318
helicsCloseLibrary (*C++ function*), 320
helicsCoreAddAlias (*C++ function*), 332
helicsCoreAddDestinationFilterToEndpoint (*C++ function*), 330
helicsCoreAddSourceFilterToEndpoint (*C++ function*), 330
helicsCoreClone (*C++ function*), 329
helicsCoreConnect (*C++ function*), 331
helicsCoreDataLink (*C++ function*), 330
helicsCoreDestroy (*C++ function*), 331
helicsCoreDisconnect (*C++ function*), 331
helicsCoreFree (*C++ function*), 331
helicsCoreGetAddress (*C++ function*), 331
helicsCoreGetIdentifier (*C++ function*), 331
helicsCoreGlobalError (*C++ function*), 333
helicsCoreIsConnected (*C++ function*), 330
helicsCoreIsValid (*C++ function*), 329
helicsCoreMakeConnections (*C++ function*), 330
helicsCoreRegisterCloningFilter (*C++ function*), 333
helicsCoreRegisterFilter (*C++ function*), 333
helicsCoreRegisterTranslator (*C++ function*), 334
helicsCoreSendCommand (*C++ function*), 332
helicsCoreSendOrderedCommand (*C++ function*), 332
helicsCoreSetGlobal (*C++ function*), 332
helicsCoreSetLogFile (*C++ function*), 332
helicsCoreSetLoggingCallback (*C++ function*), 333
helicsCoreSetReadyToInit (*C++ function*), 331
helicsCoreWaitForDisconnect (*C++ function*), 330
helicsCreateApp (*C++ function*), 325
helicsCreateBroker (*C++ function*), 322
helicsCreateBrokerFromArgs (*C++ function*), 322
helicsCreateCallbackFederate (*C++ function*), 324
helicsCreateCallbackFederateFromConfig (*C++ function*), 325
helicsCreateCombinationFederate (*C++ function*), 324
helicsCreateCombinationFederateFromConfig (*C++ function*), 324
helicsCreateCore (*C++ function*), 321
helicsCreateCoreFromArgs (*C++ function*), 322
helicsCreateDataBuffer (*C++ function*), 325
helicsCreateFederateInfo (*C++ function*), 325
helicsCreateMessageFederate (*C++ function*), 323
helicsCreateMessageFederateFromConfig (*C++ function*), 323
helicsCreateQuery (*C++ function*), 325
helicsCreateValueFederate (*C++ function*), 323
helicsCreateValueFederateFromConfig (*C++ function*), 323
helicsDataBufferCapacity (*C++ function*), 381
helicsDataBufferClone (*C++ function*), 381
helicsDataBufferConvertToType (*C++ function*), 385
helicsDataBufferFillFromBoolean (*C++ function*), 382
helicsDataBufferFillFromChar (*C++ function*), 383
helicsDataBufferFillFromComplex (*C++ function*), 383
helicsDataBufferFillFromComplexObject (*C++ function*), 383
helicsDataBufferFillFromComplexVector (*C++ function*), 384
helicsDataBufferFillFromDouble (*C++ function*), 382
helicsDataBufferFillFromInteger (*C++ function*), 382
helicsDataBufferFillFromNamedPoint (*C++ function*), 384
helicsDataBufferFillFromRawString (*C++ function*), 382
helicsDataBufferFillFromString (*C++ function*), 382
helicsDataBufferFillFromTime (*C++ function*), 383
helicsDataBufferFillFromVector (*C++ function*), 383
helicsDataBufferFree (*C++ function*), 381
helicsDataBufferIsValid (*C++ function*), 381
helicsDataBufferReserve (*C++ function*), 381
helicsDataBufferSize (*C++ function*), 381
helicsDataBufferStringSize (*C++ function*), 384

helicsDataBufferToBoolean (*C++ function*), 384
helicsDataBufferToChar (*C++ function*), 384
helicsDataBufferToComplex (*C++ function*), 384
helicsDataBufferToComplexObject (*C++ function*), 384
helicsDataBufferToComplexVector (*C++ function*), 385
helicsDataBufferToDouble (*C++ function*), 384
helicsDataBufferToInteger (*C++ function*), 384
helicsDataBufferToNamedPoint (*C++ function*), 385
helicsDataBufferToRawString (*C++ function*), 384
helicsDataBufferToString (*C++ function*), 384
helicsDataBufferToTime (*C++ function*), 384
helicsDataBufferToVector (*C++ function*), 384
helicsDataBufferType (*C++ function*), 384
helicsDataBufferVectorSize (*C++ function*), 384
helicsEndpointAddDestinationFilter (*C++ function*), 377
helicsEndpointAddDestinationTarget (*C++ function*), 377
helicsEndpointAddSourceFilter (*C++ function*), 377
helicsEndpointAddSourceTarget (*C++ function*), 376
helicsEndpointClearMessages (*C++ function*), 375
helicsEndpointCreateMessage (*C++ function*), 375
helicsEndpointGetDefaultDestination (*C++ function*), 372
helicsEndpointGetInfo (*C++ function*), 375
helicsEndpointGetMessage (*C++ function*), 375
helicsEndpointGetName (*C++ function*), 375
helicsEndpointGetOption (*C++ function*), 376
helicsEndpointGetTag (*C++ function*), 376
helicsEndpointGetType (*C++ function*), 375
helicsEndpointHasMessage (*C++ function*), 374
helicsEndpointIsValid (*C++ function*), 372
helicsEndpointPendingMessageCount (*C++ function*), 375
helicsEndpointRemoveTarget (*C++ function*), 377
helicsEndpointSendBytes (*C++ function*), 373
helicsEndpointSendBytesAt (*C++ function*), 374
helicsEndpointSendBytesTo (*C++ function*), 373
helicsEndpointSendBytesToAt (*C++ function*), 373
helicsEndpointSendMessage (*C++ function*), 374
helicsEndpointSendMessageZeroCopy (*C++ function*), 374
helicsEndpointSendString (*C++ function*), 372
helicsEndpointSendStringAt (*C++ function*), 373
helicsEndpointSendStringTo (*C++ function*), 372
helicsEndpointSendStringToAt (*C++ function*), 373
helicsEndpointSetDefaultDestination (*C++ function*), 372
helicsEndpointSetInfo (*C++ function*), 375
helicsEndpointSetOption (*C++ function*), 376
helicsEndpointSetTag (*C++ function*), 376
helicsEndpointSubscribe (*C++ function*), 374
helicsErrorClear (*C++ function*), 318
helicsErrorInitialize (*C++ function*), 318
helicsFederateAddAlias (*C++ function*), 348
helicsFederateAddDependency (*C++ function*), 349
helicsFederateClearMessages (*C++ function*), 371
helicsFederateClearUpdates (*C++ function*), 359
helicsFederateClone (*C++ function*), 339
helicsFederateCosimulationTerminationCallback (*C++ function*), 353
helicsFederateCreateMessage (*C++ function*), 371
helicsFederateDestroy (*C++ function*), 339
helicsFederateDisconnect (*C++ function*), 341
helicsFederateDisconnectAsync (*C++ function*), 341
helicsFederateDisconnectComplete (*C++ function*), 341
helicsFederateEnterExecutingMode (*C++ function*), 342
helicsFederateEnterExecutingModeAsync (*C++ function*), 342
helicsFederateEnterExecutingModeComplete (*C++ function*), 343
helicsFederateEnterExecutingModeIterative (*C++ function*), 343
helicsFederateEnterExecutingModeIterativeAsync (*C++ function*), 343
helicsFederateEnterExecutingModeIterativeComplete (*C++ function*), 343
helicsFederateEnterInitializingMode (*C++ function*), 341
helicsFederateEnterInitializingModeAsync (*C++ function*), 341
helicsFederateEnterInitializingModeComplete (*C++ function*), 341
helicsFederateEnterInitializingModeIterative (*C++ function*), 341
helicsFederateEnterInitializingModeIterativeAsync (*C++ function*), 342
helicsFederateEnterInitializingModeIterativeComplete (*C++ function*), 342
helicsFederateErrorHandlerCallback (*C++ function*), 353
helicsFederateExecutingEntryCallback (*C++ function*), 353
helicsFederateFinalize (*C++ function*), 341
helicsFederateFinalizeAsync (*C++ function*), 341
helicsFederateFinalizeComplete (*C++ function*), 341
helicsFederateFree (*C++ function*), 341
helicsFederateGetCommand (*C++ function*), 350
helicsFederateGetCommandSource (*C++ function*), 350

helicsFederateGetCore (*C++ function*), 344
helicsFederateGetCurrentTime (*C++ function*), 348
helicsFederateGetEndpoint (*C++ function*), 370
helicsFederateGetEndpointByIndex (*C++ function*), 371
helicsFederateGetEndpointCount (*C++ function*), 372
helicsFederateGetFilter (*C++ function*), 386
helicsFederateGetFilterByIndex (*C++ function*), 386
helicsFederateGetFilterCount (*C++ function*), 386
helicsFederateGetFlagOption (*C++ function*), 347
helicsFederateGetInput (*C++ function*), 358
helicsFederateGetInputByIndex (*C++ function*), 359
helicsFederateGetInputByTarget (*C++ function*), 359
helicsFederateGetInputCount (*C++ function*), 360
helicsFederateGetIntegerProperty (*C++ function*), 348
helicsFederateGetMessage (*C++ function*), 371
helicsFederateGetName (*C++ function*), 346
helicsFederateGetPublication (*C++ function*), 358
helicsFederateGetPublicationByIndex (*C++ function*), 358
helicsFederateGetPublicationCount (*C++ function*), 359
helicsFederateGetState (*C++ function*), 343
helicsFederateGetTag (*C++ function*), 349
helicsFederateGetTimeProperty (*C++ function*), 347
helicsFederateGetTranslator (*C++ function*), 354
helicsFederateGetTranslatorByIndex (*C++ function*), 355
helicsFederateGetTranslatorCount (*C++ function*), 354
helicsFederateGlobalError (*C++ function*), 340
helicsFederateHasMessage (*C++ function*), 371
helicsFederateInfoClone (*C++ function*), 335
helicsFederateInfoFree (*C++ function*), 336
helicsFederateInfoLoadFromArgs (*C++ function*), 336
helicsFederateInfoLoadFromString (*C++ function*), 336
helicsFederateInfoSetBroker (*C++ function*), 337
helicsFederateInfoSetBrokerInitString (*C++ function*), 336
helicsFederateInfoSetBrokerKey (*C++ function*), 337
helicsFederateInfoSetBrokerPort (*C++ function*), 337
helicsFederateInfoSetCoreInitString (*C++ function*), 336
helicsFederateInfoSetCoreName (*C++ function*), 336
helicsFederateInfoSetCoreType (*C++ function*), 337
helicsFederateInfoSetCoreTypeFromString (*C++ function*), 337
helicsFederateInfoSetFlagOption (*C++ function*), 338
helicsFederateInfoSetIntegerProperty (*C++ function*), 339
helicsFederateInfoSetLocalPort (*C++ function*), 338
helicsFederateInfoSetSeparator (*C++ function*), 338
helicsFederateInfoSetTimeProperty (*C++ function*), 338
helicsFederateInitializingEntryCallback (*C++ function*), 353
helicsFederateIsAsyncOperationCompleted (*C++ function*), 342
helicsFederateIsProtected (*C++ function*), 340
helicsFederateIsValid (*C++ function*), 340
helicsFederateLocalError (*C++ function*), 340
helicsFederateLogDebugMessage (*C++ function*), 349
helicsFederateLogErrorMessage (*C++ function*), 349
helicsFederateLogInfoMessage (*C++ function*), 349
helicsFederateLogLevelMessage (*C++ function*), 350
helicsFederateLogWarningMessage (*C++ function*), 349
helicsFederatePendingMessageCount (*C++ function*), 371
helicsFederateProcessCommunications (*C++ function*), 346
helicsFederateProtect (*C++ function*), 339
helicsFederatePublishJSON (*C++ function*), 359
helicsFederateRegisterCloningFilter (*C++ function*), 386
helicsFederateRegisterEndpoint (*C++ function*), 369
helicsFederateRegisterFilter (*C++ function*), 385
helicsFederateRegisterFromPublicationJSON (*C++ function*), 359
helicsFederateRegisterGlobalCloningFilter (*C++ function*), 386
helicsFederateRegisterGlobalEndpoint (*C++ function*), 369
helicsFederateRegisterGlobalFilter (*C++ function*), 385
helicsFederateRegisterGlobalInput (*C++ function*), 357
helicsFederateRegisterGlobalPublication (*C++ function*), 356

helicsFederateRegisterGlobalTargetedEndpoint (*C++ function*), 370
helicsFederateRegisterGlobalTranslator (*C++ function*), 354
helicsFederateRegisterGlobalTypeInput (*C++ function*), 358
helicsFederateRegisterGlobalTypePublication (*C++ function*), 356
helicsFederateRegisterInput (*C++ function*), 357
helicsFederateRegisterInterfaces (*C++ function*), 340
helicsFederateRegisterPublication (*C++ function*), 355
helicsFederateRegisterSubscription (*C++ function*), 355
helicsFederateRegisterTargetedEndpoint (*C++ function*), 370
helicsFederateRegisterTranslator (*C++ function*), 354
helicsFederateRegisterTypeInput (*C++ function*), 357
helicsFederateRegisterTypePublication (*C++ function*), 356
helicsFederateRequestNextStep (*C++ function*), 344
helicsFederateRequestTime (*C++ function*), 344
helicsFederateRequestTimeAdvance (*C++ function*), 344
helicsFederateRequestTimeAsync (*C++ function*), 345
helicsFederateRequestTimeComplete (*C++ function*), 345
helicsFederateRequestTimeIterative (*C++ function*), 345
helicsFederateRequestTimeIterativeAsync (*C++ function*), 345
helicsFederateRequestTimeIterativeComplete (*C++ function*), 346
helicsFederateSendCommand (*C++ function*), 350
helicsFederateSetFlagOption (*C++ function*), 346
helicsFederateSetGlobal (*C++ function*), 348
helicsFederateSetIntegerProperty (*C++ function*), 347
helicsFederateSetLogFile (*C++ function*), 349
helicsFederateSetLoggingCallback (*C++ function*), 351
helicsFederateSetQueryCallback (*C++ function*), 351
helicsFederateSetSeparator (*C++ function*), 347
helicsFederateSetStateChangeCallback (*C++ function*), 352
helicsFederateSetTag (*C++ function*), 348
helicsFederateSetTimeProperty (*C++ function*), 346
helicsFederateSetTimeRequestEntryCallback (*C++ function*), 351
helicsFederateSetTimeRequestReturnCallback (*C++ function*), 352
helicsFederateSetTimeUpdateCallback (*C++ function*), 352
helicsFederateUnProtect (*C++ function*), 339
helicsFederateWaitCommand (*C++ function*), 350
helicsFilterAddDestinationTarget (*C++ function*), 388
helicsFilterAddSourceTarget (*C++ function*), 388
helicsFilterGetName (*C++ function*), 387
helicsFilterGetPropertyDouble (*C++ function*), 388
helicsFilterGetPropertyString (*C++ function*), 388
helicsFilterIsValid (*C++ function*), 387
helicsFilterSet (*C++ function*), 387
helicsFilterSetCustomCallback (*C++ function*), 387
helicsFilterSetString (*C++ function*), 387
helicsGetBuildFlags (*C++ function*), 318
helicsGetCompilerVersion (*C++ function*), 318
helicsGetDataType (*C++ function*), 320
helicsGetFederateByName (*C++ function*), 319
helicsGetFlagIndex (*C++ function*), 320
helicsGetOptionIndex (*C++ function*), 320
helicsGetOptionValue (*C++ function*), 320
helicsGetPropertyIndex (*C++ function*), 319
helicsGetSystemInfo (*C++ function*), 318
helicsGetVersion (*C++ function*), 318
helicsInputClearUpdate (*C++ function*), 369
helicsInputIsUpdated (*C++ function*), 369
helicsInputLastUpdateTime (*C++ function*), 369
helicsIsCoreTypeAvailable (*C++ function*), 319
helicsLoadSignalHandler (*C++ function*), 318
helicsLoadSignalHandlerCallback (*C++ function*), 319
helicsLoadSignalHandlerCallbackNoExit (*C++ function*), 319
helicsLoadThreadedSignalHandler (*C++ function*), 318
helicsQueryBrokerExecute (*C++ function*), 393
helicsQueryBufferFill (*C++ function*), 395
helicsQueryCoreExecute (*C++ function*), 393
helicsQueryExecute (*C++ function*), 392
helicsQueryExecuteAsync (*C++ function*), 393
helicsQueryExecuteComplete (*C++ function*), 393
helicsQueryFree (*C++ function*), 394
helicsQueryIsCompleted (*C++ function*), 394
helicsQuerySetOrdering (*C++ function*), 394
helicsQuerySetQueryString (*C++ function*), 394
helicsQuerySetTarget (*C++ function*), 394

helicsTranslatorAddDestinationEndpoint (*C++ function*), 391
helicsTranslatorAddInputTarget (*C++ function*), 391
helicsTranslatorAddPublicationTarget (*C++ function*), 391
helicsTranslatorAddSourceEndpoint (*C++ function*), 391
helicsTranslatorGetName (*C++ function*), 390
helicsTranslatorIsValid (*C++ function*), 390
helicsTranslatorSet (*C++ function*), 390
helicsTranslatorSetCustomCallback (*C++ function*), 390
helicsTranslatorSetString (*C++ function*), 390
helicsWrapDataInBuffer (*C++ function*), 321