RadServe User Manual

Inras GmbH
Altenbergerstraße 69
4040 Linz, Austria
Email: office@inras.at
Tel. Nr.: +43 732 2468 6384

Linz, December 2019

Contents

1 RadServe

RadServe is a software tool which allows remote access to different radar systems from Inras. It is meant to run on a powerful computer to allow maximum data transfer rates from the board to the PC and within the PC to stream the recorded data to an HDF5 file. The program enables remote control of the radar system as shown in Fig. 1. In a typical setup the RadarLog is connected to a powerful PC1 equipped with an SSD (e.g. Intel NUC with a Linux operating system). RadServe handles the USB communication and is used to configure the radar system remotely via Matlab or Python scripts. In the shown setup the application is run on PC2 and a TCP/IP connection is used to configure the radar system at startup.

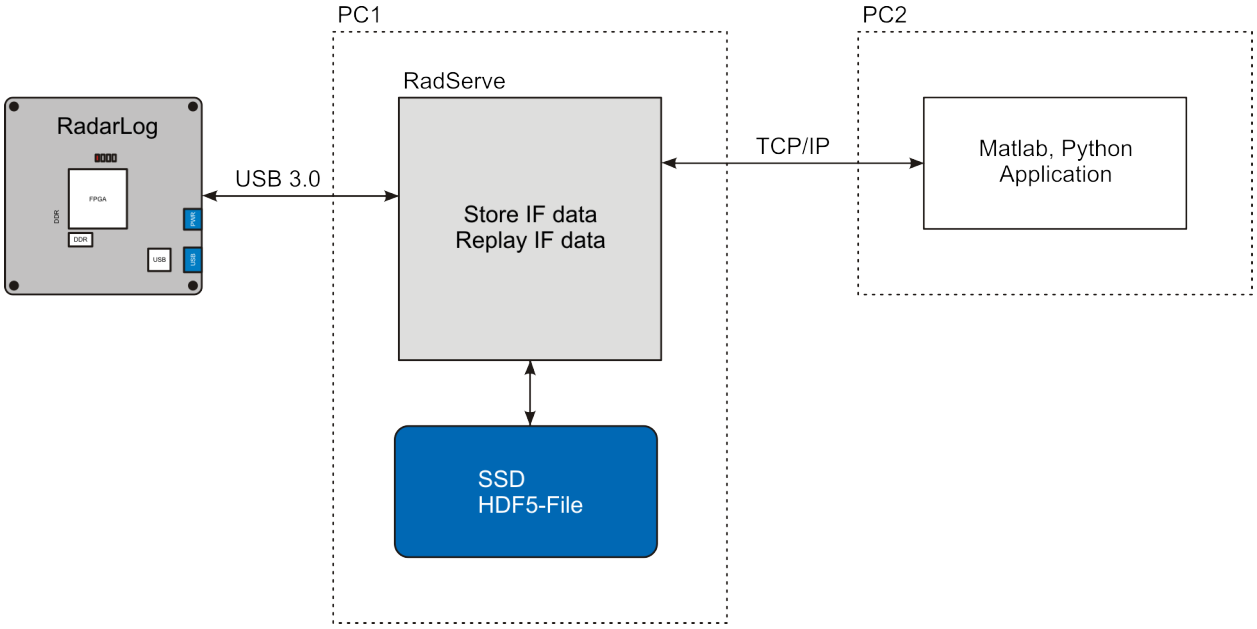


Figure 1: Typical setup with two PCs.

RadServe controls the USB 3.0 interface for accessing the measured IF data at high data rates. The recorded samples can be streamed to an HDF5 file or forwarded over the TCP/IP connection. For continuous data recording the maximum transfer rate can only be achieved by streaming the data directly to an HDF5 file. If frontends with multiple receive antennas are operated, the measurement data is stored to separate datasets in the HDF5 file for simplified data access.

As of version `3.3.0` RadServe also supports all features with a TCP/IP connection to Radarbook2.

RadServe is available in two versions: a console version with a simple command line interface and a GUI version for perpetual status observations and visualization of the measurement data.

2 Main Features

RadServe handles the USB 3.0 interface to the radar platforms (RadarLog and Radarbook with USB module), the TCP/IP connection to the Radarbook2 and the USB 2.0 interface to the TinyRad. It functions as a TCP/IP server for remote configuring and data access, and is available with graphical user interface or as console program for Linux.

Its main features (version `3.3.0`) include

- a buffered transfer mode over TCP/IP to minimize packet loss, optionally with logging to an HDF5 file,
- the creation of HDF5 files with tested transfer rates up to 3.0 GBit/s (USB connection to RadarLog),
- the inclusion of multiple camera streams while measuring (both, to HDF files and via DataPort),
- replaying of previously recorded HDF5 files, as if received directly from the board,
- data post processing to return the Range Profile, Range Doppler Map, Detection Lists, Target Tracks, and
- a data post processing interface for customer specific signal processing.

RadServe supports

- USB cameras, accessible with OpenCV,
- FLIR GigE cameras, via the FlyCapture2 SDK,
- Optris cameras, via the IRImagerDirect SDK,
- multiple devices (best connected to separate USB controllers),
- synchronization of radar data and video.

The GUI version provides

- a buffer state display to help in identifying bottlenecks in the file creation process,
- live visualization of the measured data and calculated data as well as
- a visualization of recorded HDF5 files.

3 Supported Operating Systems

In the subsequent table the supported and tested operating systems are listed.

Type	Operating System	Supported and tested
GUI	Windows 7 (64-Bit)	Yes
GUI	Windows 10 (64-Bit)	Yes
GUI	Ubuntu 16.04 (64-Bit)	Yes
Console	Ubuntu 16.04 (64-Bit)	Yes
Console	Vibrante Linux for NVidia Drive PX2 (64-Bit)	Yes (without FLIR camera support)
GUI	Vibrante Linux for NVidia Drive PX2 (64-Bit)	Yes (without FLIR camera support)

4 Starting RadServe

This section describes how to start RadServe either as GUI version or on the console. Main causes for server startup failures are providing an incorrect IP address, trying to use an unavailable port or a port that is already in use. After successfully starting the server, RadServe is listening on the configured TCP/IP port for new clients and their commands. The provided Matlab and Python application notes can be run.

4.1 GUI Version

After starting the GUI version of RadServe (RadServeGui.exe) the default window appears. After startup the server is not activated and must be started manually by clicking the *Start Server* button of the server configuration bar, as shown in Fig. 2. As of version `2.5.6` it is possible to set RadServe to auto-start on startup. Therefore, the checkbox on the left side of the configuration bar is used.

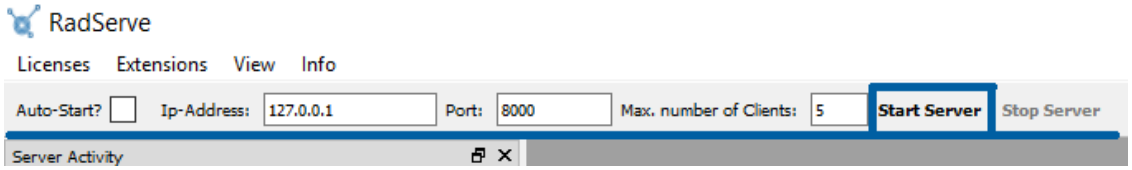


Figure 2: RadServe GUI with highlighted server configuration bar.

Before starting the server its IP address and port may be set to the desired values. The provided settings are stored to a configuration file to be loaded on restarting RadServe.
In the shown example the loopback address `127.0.0.1` is used. For accessing RadServe from a remote location the IP address of the computer running RadServe should be used. Port `8000` is set as the default configuration port in both the Matlab class and the Python class. If a different port is required, the application scripts must be adapted as well, which is described in Section **??**. If the server is running, the configuration fields disappear and the state of the server along with its active IP address and port are displayed instead. In the shown example the status message would be **RadServe State: Running. Listening on 127.0.0.1:8000**. If the server could not be started an error message is displayed.

4.2 Console Version

By using the console version of RadServe the server may be started either on application startup via command line arguments, such as

```
$ ./RadServeConsole start_server
```

for using the default configuration, or

```
$ ./RadServeConsole <ip-address>
```

for simultaneously setting a new server IP address, or after running the application via the command `start_server`. Similar to the GUI version, the console version can be set to automatically start, via the configuration command `config_server`. On startup the console version prints a list of all available commands for configuring and interacting with RadServe.

5 Connecting to RadServe and Selecting a Device

The configurations of the RF frontends are explained in the corresponding application notes for the frontends. If the data is accessed via the TCP/IP link of RadServe the same Matlab or Python scripts can be used. The only difference is the selection of the communication link.

5.1 RadarLog

If the board is operated directly from Matlab the board object has to be generated with

```
RadarLog('Usb');
```

where the first parameter selects the local USB interface. If the RadarLog is controlled via RadServe the board object has to be generated with

```
Brd = RadarLog('RadServe', '127.0.0.1');
```

where the first argument selects the server and the second argument is the IP address of the server. If RadServe runs on the same PC then the loopback address `'127.0.0.1'` can be used. All other commands are compatible and are explained in the application notes in more detail.

For setting the port, if one other than the default port of 8000 is required, a third argument can be given to specify the port

```
Brd = RadarLog('RadServe', '127.0.0.1', '8000');
```

The used RadarLog device is selected with the first command that requires interaction with the RadarLog. It is then used for the complete duration of the current connection to RadServe. By default, the first device is used. If more than one RadarLog is connected to RadServe two options for selecting the used RadarLog are available: it can either be selected via its UID or with its index. Both, UID and index, are given as a fourth argument.

```
Brd = RadarLog('RadServe', '127.0.0.1', '8000', '2E00000001A6D6A0');
```

```
Brd = RadarLog('RadServe', '127.0.0.1', '8000', 1);
```

Index 1 is used for selecting the second device connected to RadServe. The indices corresponding to the connected devices (identified via their `UID`) are listed in the *Connected Devices* view of the RadServe GUI (see Fig. 3).

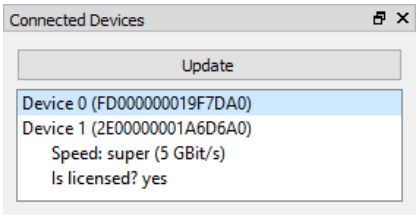


Figure 3: Connected Devices view with multiple devices.

Alternatively, the list of devices can be printed by connecting to RadServe and using the command:

```
Brd.ListDev();
```

Afterwards, the required device can be selected or changed (as long as no command, which requires an interaction with the board, was sent) with

```
Brd.SetIndex(<index>);
```

5.2 Radarbook2

As of `3.3.0` RadServe also supports the Radarbook2. To connect to the Radarbook2 via RadServe the board object has to be generated with

```
Brd = Rbk2('RadServe', '127.0.0.1');.
```

where the first argument selects the server and the second argument is the IP addresss of RadServe. By default, RadServe searches for the Radarbook2 at IP address 192.168.1.1, if the Radarbook2 has another IP address, the used IP address, such as 192.168.1.2, must be given as the fourth parameter:

```
Brd = Rbk2('RadServe', '127.0.0.1', 8000, '192.168.1.2');
```

The third parameter is the used port of RadServe, which is by default 8000.

5.3 TinyRad

As of `3.4.0` RadServe also supports the TinyRad. To connect to the TinyRad via RadServe the board object has to be generated with

```
Brd = TinyRad('RadServe', '127.0.0.1');.
```

where the first argument selects the server and the second argument is the IP addresss of RadServe. For setting the port, if one other than the default port of 8000 is required, a third argument can be given to specify the port

```
Brd = TinyRad('RadServe', '127.0.0.1', '8000');
```

The used TinyRad device is selected with the first command that requires interaction with the TinyRad. It is then used for the complete duration of the current connection to RadServe. By default, the first device is used. If more than one TinyRad is connected to RadServe two options for selecting the used TinyRad are available: it can either be selected via its UID or with its index. Both, UID and index, are given as a fourth argument.

```
Brd = TinyRad('RadServe', '127.0.0.1', '8000', '2E00000001A6D6A0');

Brd = TinyRad('RadServe', '127.0.0.1', '8000', 1);
```

Index 1 is used for selecting the second device connected to RadServe. The indices corresponding to the connected devices (identified via their `UID`) are also listed in the *Connected Devices* view of the RadServe GUI (see Fig. 4).

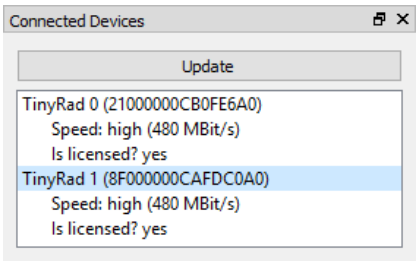


Figure 4: Connected Devices view with multiple TinyRad devices.

6 Accessing Data

After configuring the measurements, the measurement data can be received with

```
Data = Brd.BrdGetData(<number-of-frames>);
```

This command signifies to RadServe that a buffer of size `BufSiz` × `N` × `NrChn` × `Mult` should be created to store the received measurement data.

6.1 Data Buffer Behavior and Parameters

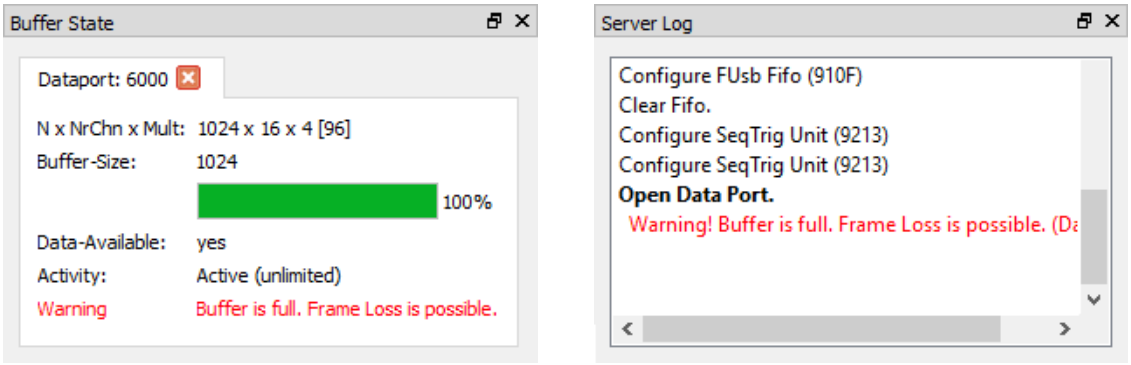
All data collection features of RadServe, such as data access via the data port or the creation of HDF files, start with the same buffering mechanism.
The data buffer contains `BufSiz` USB packets (`Mult` frames of size `N` × `NrChn`). A USB transfer, containing one USB packet, is scheduled by RadServe, processed by the operating system and then, after completion, timestamped by RadServe. For continuous data collection RadServe simultaneously schedules `UsbNrTx` USB transfers to avoid frame loss. The buffer of RadServe only provides full USB packets to the next processing step. Therefore, the buffer parameters `Mult`, `BufSize` and `UsbNrTx` may need adjusting when continuous and in-time processing is required:

- `Mult` should be decreased for slow measurements to reduce the size of the USB transfers. For fast measurements it can be increased to decrease USB transfer overhead.
- If data collection on the application side is slower than on the side of RadServe the `BufSiz` should be increased accordingly.
- `UsbNrTx` influences the reserve of scheduled USB transfers. For optimal data collection the operating system should always have unprocessed USB transfers waiting to bridge the time required by RadServe between finalizing finished USB transfers and scheduling new transfers. The maximum number of simultaneous transfers is limited by the OS. On Ubuntu it is dependant on the total size of the scheduled transfers. RadServe automatically reduces `UsbNrTx`, when scheduling of additional transfers fails.

The buffer parameters are used at buffer creation and can be set with the following commands:

```
Brd.ConSet('Mult', 32);
Brd.ConSet('BufSiz', 256);
Brd.ConSet('UsbNrTx', 24);
```

The state of the data buffer can be examined since RadServe version `2.5.5`, either via the *Buffer State* display of the GUI version or via printed status messages of the console version. If the used buffer is at full capacity a warning message will appear to signify that packet loss is possible or may have already happened (see Fig. 5).



(a) Buffer State view. (b) Server Log view.

Figure 5: Warnings shown when the buffer is full.

During data buffering RadServe disallows any configuration access via USB. To stop and release the USB lock the data buffering must be stopped either by closing the data port manually, with

```
Brd.CloseDataPort();,
```

or by predefining the number of packets to receive, with

```
Brd.ConSet('NumPackets', <number-of-packets>);.
```

After RadServe stops buffering data, additional configuration commands, such as `Brd.BrdPwrDi()` to disable the frontend's power supply, can be sent.

6.2 Switching between Data Access and Reconfiguration

As of version `2.6.1`, it is possible to receive a predefined number of packets from RadServe, reconfigure the measurements and restart receiving data again. Therefore, the number of packets must be set by using the command

```
Brd.ConSet('NumPackets', <number-of-packets>);
```

The data port has to be kept alive, before accessing any data with `Brd.BrdGetData()` to avoid automatical closure, with

```
Brd.ConSet('KeepAlive', true);
```

When no more data is required the data port should be closed with:

```
Brd.CloseDataPort();
```

6.3 Enabling Time Stamp Information

The RadarLog supports an independent 32-bit timestamp for each frame (for further information see `AN_02`). Additionally, RadServe stores a 64-bit system timestamp, in ms, with every received

USB packet. As of version `2.6.3`, this timestamp can be provided with the data. Therefore, the command

```
Brd.Set('EnaTimStamp');
```

must be called to enable timestamp information. The timestamp of the RadarLog is appended to the first channel, whereas the timestamp of RadServe is appended to the last enabled channel. Therefore, if only one channel is enabled, at every `Mult` frame the timestamp of the RadarLog is overwritten with the timestamp of RadServe. If two, or more, channels are enabled both timestamps are available. The timestamp of RadServe is appended to the last frame of every USB packet (see Fig. 6).

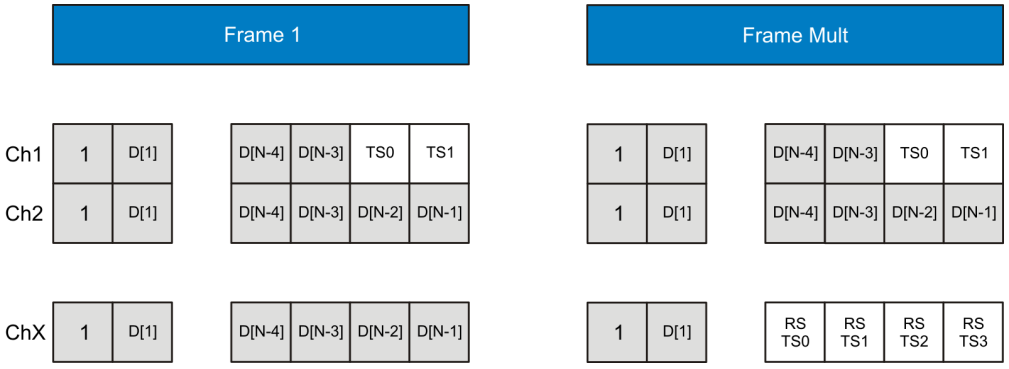


Figure 6: Location of the RadServe timestamp within a USB packet.

The listed commands for reading the RadServe timestamp only provide valid output, when a multiple of `Mult` packages is read.

In Matlab the timestamp can be read with

```
[Data TimStamp TimStampRS] = Brd.BrdGetData().
```

In Python, the timestamp can be read, after calling `Data = Brd.BrdGetData()` with

```
TimStampRS = Brd.BrdGetRadServeTimStamp().
```

Otherwise, the received timestamp can be printed, both in Matlab and Python, with

```
Brd.PrintRadServeTimStamp().
```

This function can also be used to print any other timestamp in ms. With the command

```
Brd.Set('DiTimStamp');
```

the timestamp information can be disabled.

6.4 Data Post Processing Interface - RadServe Extensions

As of version `3.0.0` RadServe contains a data post processing interface for customer specific signal processing called RadServe Extensions. An extension is a user-created DLL or shared object that is appended, via a socket connection, to the data port of RadServe to manipulate the raw measurement data and provide results to the application, as depicted in Fig. 7.

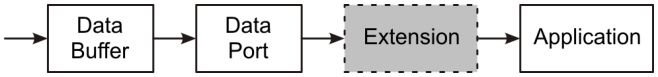


Figure 7: Block diagram of the data collection using RadServe extensions.

An extension is configured with the command

```
Brd.CfgRadServe_Extension('General', '<Path>', <Selection>, <Parameters>);
```

providing the path to the DLL or shared object file, as well as, an optional selection value and a parameter array. Since RadServe and the application do not have knowledge of the structure of the extension data the received data is provided as is, when using the command

```
Brd.BrdGetData(1, 'Extension', 1);
```

for collecting the data. To receive the data correctly structured and typecast the command

```
Data = Brd.BrdGetMatrix(<number-of-frames>, 'DataType', <datatype>,
                                            'Dimensions', <dimensions-array>);
```

may be used. The number of frames parameter defines how many frames should be collected at once. The datatype can be any of the Matlab type strings (`'uint8'`, `'int8'`, `'uint16'`, `'int16'`, `'uint32'`, `'int32'`, `'uint64'`, `'int64'`, `'single'`, or `'double'`) or one of the two strings, `'complex'` and `'complex_single'`, for typecasting to double-precision complex or single-precision complex numbers. When casting to complex values the real and imaginary parts of the complex numbers must be interleaved, starting with the real part. The third parameter is used for defining the required output shape of the matrix. It is given as a dimensions-array which is then used for reshaping the received data.

Further information on the creation and execution of an extension is given in the **RadServe Extension Guide**.

6.5 Range Profile, Range Doppler, Detection List & Target Track Computation

By default, RadServe provides the raw measurement data via the dataport, as depicted in Fig. 8.

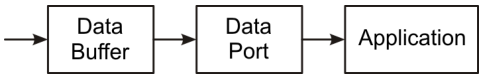


Figure 8: Block diagram of the raw data collection.

As of version `3.2.0` it was possible to append three computation levels to the buffered data of RadServe. As of version `3.3.0` a target tracker was added. The first level is used for computing the `Range Profile`, the second level for computing the `Range-Doppler Map`, the third level for computing a `Detection List` of detection and the fourth level computes `Target Tracks`. Fig. 9 depicts the four possible layers of computation.

Figure 9: Block diagram of the extended data collection.

The required configuration of the four levels is shown in separate application notes. Changing the current data port level is done with the command

```
Brd.Computation.SetType(<level>);
```

where the level parameter is one of the following values: `'Raw'`, `'RangeProfile'`, `'RangeDoppler'`, `'DetectionList'` and `'TargetTracker'`. Each of these commands sets the current ouput level of RadServe, by default raw data is provided. By calling `Brd.BrdGetData();` afterwards, the computation result is collected.

Additionally to collecting one of the different data types it is possible to append an extension to the computation (see Fig. 10).

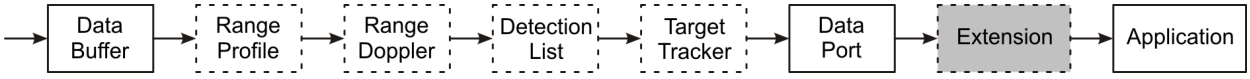


Figure 10: Block diagram of the extended data collection using RadServe extensions.

Further information on the configuration and usage of the internal computation is given in the **RadServe Computation Guide**.

6.6 LogFile Creation

As of version `3.0.0` it is possible to create a log file of the raw data when streaming data via the socket connection. As of version `3.2.0` it is possible to log not just raw data but any of the different computational steps. It is currently only possible to log one step of the data post processing (raw data, range profile, range-Doppler map, the target list or the extension results). Logging data to an HDF file is activated with the command

```
Brd.CfgRadServe('LogDataPortToFile', 'LogFile', 'RangeProfile');.
```

This generates a log file `'LogFile.h5'` for the range profile. The provided datatype sets the highest level of data to log, if it is not available the next lower data level is logged instead. In the given example the raw data would be logged if no range profile is computed. Options for the data type parameter are: `'Raw'`, `'RangeProfile'`, `'RangeDoppler'`, `'DetectionList'`, `'TargetTracker'` and `'Extension'`. Fig. 11 depicts the log file creation, only one log file can be created at once.

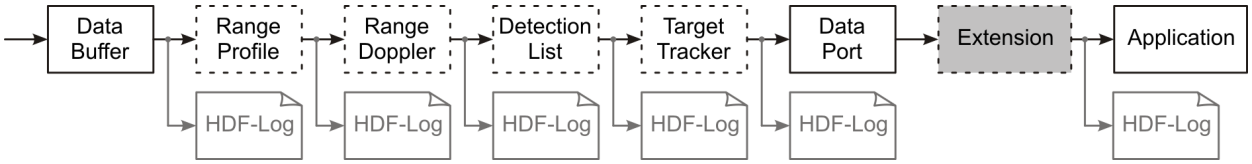


Figure 11: Block diagram of the data port with log-file creation.

To deactivate logging the following command is used

```
Brd.CfgRadServe('LogDataPortToFile', '');.
```

The configuration for creating a log file is persistent, if the creation of a log file was activated and a log file is no longer required it must be manually deactivated, as RadServe stores this current configuration.

7 File Creation and Replay

Instead of streaming the measurement data via a socket connection to the application RadServe can store the measurement data directly to an HDF file (see Fig. 12).

Figure 12: Block diagram of the raw data file creation.

For optimizing the overall continuous transfer rate, writing to an SSD is recommended, as writing to a simple HDD can result in packet loss during write access even at lower measurement speeds.

To create an HDF file the following command is used after configuring the measurement:

```
Brd.CreateFile(<filename>, <number-of-packets>);
```

This command starts the buffering and creates a new file `'<filename>.h5'` in the HDF directory specified in RadServe. Depending on the configuration of RadServe a timestamp can be appended to the file name to avoid overriding previously created files. The number of packets parameter defines the maximum number of USB packets to be written to the file. The buffering and file creation process is stopped when the maximum number of packets was written, or when the command

```
Brd.CloseFile();
```

was sent. Using the close-command cancels all pending transfers and closes the file. When unlimited streaming to the file is desired the number of packets parameter can be omitted or set to 0. If unlimited streaming is activated the file must be closed with the close-command.

The following example shows how a file recording is stopped after receiving 1024 packets, or after 10 seconds, whichever comes first:

```
% Setup and Initialization excluded
Brd.CreateFile('Traf', 1024);
pause(10);
Brd.CloseFile();
```

7.1 Adding Additional Attributes to HDF Files

Before streaming data to an HDF file, additional file attributes may be sent to RadServe to be stored within the HDF file. Therefore, the following command is used to provide a key-value pair, which is written directly to the file:

```
Brd.SetFileParam(<key>, <value>, <type>);
```

The key must be a string-value to be used as the attribute name within the HDF file, by default the value is stored as type of `'STRING'`. The following types of data are supported:

- `'STRING'`: default, for storing the value as a string,

- `'INT'`: for storing a single 32-bit values,
- `'DOUBLE'`: for storing a single 64-bit values,
- `'ARRAY32'`: for storing an array of 32-bit values, and
- `'ARRAY64'`: for storing an array of 64-bit values, such as `double` or `uint64`.

7.2 Writing Extensions to an HDF File

Additionally to storing raw measurement data it is also possible to store the results of an extension to an HDF file (see Fig. 13).

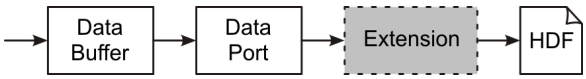


Figure 13: Block diagram of the file creation using RadServe extensions.

The command required for creating an extension file is

```
Brd.CreateFile(<filename>, <number-of-packets>, 1);
```

The name and number of packets parameters are the same as for creating a regular file. The third parameter is required to notify RadServe that the extension should be created for this file.

Since RadServe has no knowledge of the resulting data the results are stored as is.

7.3 Writing Computational Results to an HDF File

Since RadServe is capable of computing the range profile, the range-Doppler map, list of detections and target tracks, these results can also be stored to an HDF file (see Fig. 14).

Figure 14: Block diagram of the extended file creation.

Equal to appending an extension to the data port of RadServe an extension can also be appended to each computation level of the measurement data when streaming data to a file (see Fig. 15).

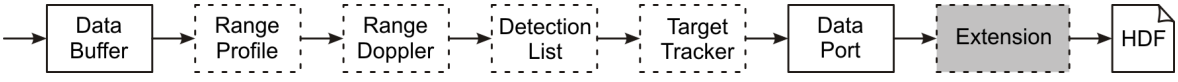


Figure 15: Block diagram of the extended file creation using RadServe extensions.

7.4 File Replay

RadServe can be used to replay the created HDF files. When replaying a file the recorded data is provided as if received directly from the board, see Fig. 16.

Figure 16: Block diagram of a replaying an HDF file.

To replay a recorded HDF file the command

```
Brd.ReplayFileN(<filename>);
```

is used. This function takes named parameters: Optional parameters are `'FrameIdx'`, which is used for indicating the first frame that should be read from the file, and `'WithVideo'` for also replaying any stored video images of the file. If available, the recorded video will be provided on a separate port. For example, the command

```
Brd.ReplayFileN('Traf', 'FrameIdx', 5, 'WithVideo', 'yes');
```

is used for replaying the file `'Traf.h5'`, starting with frame 5 and additionally, if available, streaming the contained video. By default, replaying starts at frame 1 without searching for any video data.

After having sent the replay command the parameters used for creating the HDF file can be read via

```
Brd.Get(<parameter-name>);.
```

It is required to read the parameters `'N'` and `'NrChn'` to correctly initialize the Matlab/Python classes. The parameter `'FileSize'` returns the number of frames stored in the HDF file. The parameter `'MeasStart'` returns the time in ms of the first received data packet. In Matlab, this time may be converted and displayed with

```
msg = Brd.Get('MeasStart');
disp(['Measurement Start: 'datestr( datetime(ms/1000,
                    'ConvertFrom', 'posixtime'), 'dd-mmm-yyyy HH:MM:SS:FFF')]);
```

Additional data, which was stored via `Brd.SetFileParam()` can be read with

```
data = Brd.GetFileParam(<key>, <type>);.
```

Data stored as a 64-bit array is returned as a double-array. To change these values to uint64, they can be casted, in Matlab, with `data = typecast(data, 'uint64');`.

The replayed data can be received with the same command as when receiving directly from the board:

```
Brd.BrdGetData(<number-of-frames>);.
```

Keep in mind, that the timeout of the socket connection is large enough to allow for receiving the complete number of frames, otherwise a 'socket timeout' may happen. The timeout can be set with

```
Brd.SetSocketTimeout(<timeout>);
```

before the `Brd.ReplayFileN()` command is executed.

The replay mode of RadServe can be stopped with

```
Brd.StopReplayFile();.
```

As of version `2.6.3` the timestamps contained within the created HDF-Files can be appended to the measurement data (see 6.3) during replay. The RadServe timestamp is always available within an extra dataset, even if the file was created without timestamp information of the RadarLog. During replay the RadServe timestamp is added to the last channel of each `Mult` frame, instead of the last four data values. If the parameter `'FrameIdx'` is set to a frame that is not a multiple of `Mult`, all provided timestamps are offset.

7.5 File Replay of Extension Data

For replaying an extension file (see Fig. 17) the replay command is changed to

```
Brd.ReplayFileN('Traf', 'Extension', 'yes');.
```

Figure 17: Block diagram of replaying an extension file.

Since RadServe and the application script do not have any knowledge of the extension data the

```
Brd.BrdGetData(1, 'Extension', 1)
```

command does return the data, as is, without any structure.

To receive a structured and correctly typecasted version of the data the command

```
Data = Brd.BrdGetMatrix(<number-of-frames>, 'DataType', <datatype>,
                                            'Dimensions', <dimensions-array>);
```

should be used.

7.6 File Replay of Computation Data

For replaying a file containing one of the computation results, i.e. the range profile, range-Doppler map, detection list or target tracks (see Fig. 18) the replay command must be adapted to one of the following

```
Brd.ReplayFileAs('RangeProfile', <filename>);,
Brd.ReplayFileAs('RangeDoppler', <filename>);,
Brd.ReplayFileAs('DetectionList', <filename>);,
Brd.ReplayFileAs('TargetTracker', <filename>);.
```

(a) Block diagram of a replaying the stored range profile.

(b) Block diagram of a replaying the stored range-Doppler map.

(c) Block diagram of a replaying the stored detection list.

(d) Block diagram of a replaying the stored target tracks.

Figure 18: Replaying computation data.

The optional parameters are the same as for replaying raw data. The parameters used for creating the results can be read with the command

```
Brd.GetFileParam(<parameter-name>);.
```

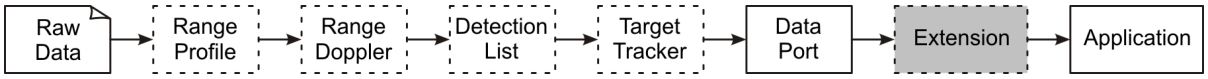
After reading any file parameters the data can be collected, as usual, with

```
Brd.BrdGetData(<number-of-frames>);.
```

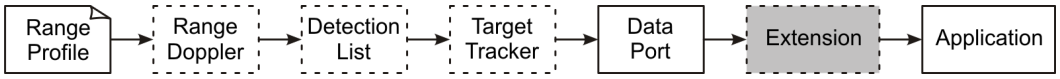
7.7 Replay Data with Computation or Extension

As of RadServe version `3.2.0` it is possible to use a raw data HDF file for computing the range profile, range-Doppler map, detection lists, target tracks or even an extension. Fig. 19 displays the possible options for replaying the data files.
To replay the computation results of a file, one of the following `ReplayFileAs`-commands is used:

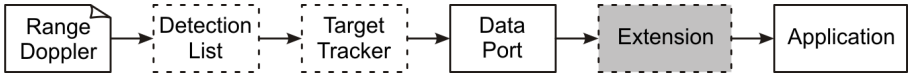
```
Brd.ReplayFileAs('RangeProfile', <filename>);,
Brd.ReplayFileAs('RangeDoppler', <filename>);,
Brd.ReplayFileAs('DetectionList', <filename>);,
```



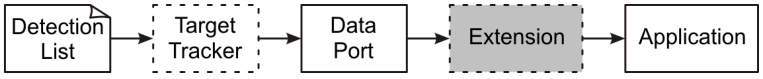
(a) Block diagram of the possible options for replaying raw data.



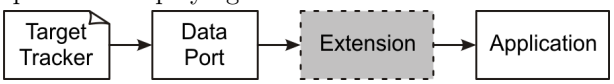
(b) Block diagram of the possible options for replaying range profiles.



(c) Block diagram of the possible options for replaying range-Doppler maps.



(d) Block diagram of the possible options for replaying detection lists.



(e) Block diagram of the possible options for replaying target tracks.

Figure 19: Replaying data with computations or extensions.

```
Brd.ReplayFileAs('TargetTracker', <filename>);.
```

The first parameter provides the required data type for replaying the file given by the second parameter. The required file attributes for computing the range profile, range-Doppler map or the target list of an HDF file are described in the **RadServe Computation Guide**. After reading any required file attributes and parameters the resulting data can be collected, as usual, with

```
Data = Brd.BrdGetData(1);.
```

It is also possible to append the extension to the computation, therefore the `ReplayFileAs` command must be extended with the extension parameter:

```
Brd.ReplayFileAs('Raw', <filename>, 'Extension', 'yes');,
Brd.ReplayFileAs('RangeProfile', <filename>, 'Extension', 'yes');,
Brd.ReplayFileAs('RangeDoppler', <filename>, 'Extension', 'yes');,
Brd.ReplayFileAs('TargetList', <filename>, 'Extension', 'yes');.
```

7.8 HDF File Format

For rapidly storing the measurement data, HDF5 files are created. The data is split according to the channels and stored separately in HDF objects named `'/Chn1'` to `'/ChnX'`.

7.8.1 Packet Tables, Datasets

In files of file version 1.0.0 the separate channels are stored as HDF `Packet Tables` (named `'/Chn1'` to `'/ChnX'`). As of file version 2.0.0 chunked (`Mult` × `N`) HDF `Datasets` are used instead of Packet Tables. Additionally a dataset `'/ChnTime'` was added. It contains `uint64` timestamps (POSIX time) in ms of whenever `Mult` frames have been stored to the file. The camera

streams are stored as a Packet Table. As of file version 3.0.0 the camera streams are also stored as datasets.

7.8.2 Camera Streams

If required, camera images are stored as well. In files of version 2.0.0, or earlier, a single camera stream is stored in an HDF Packet Table named `'/Camera'`. As of file version 2.0.1 multiple camera streams can be stored, each to its own HDF Packet Table, with a user-defined name. As of file version 3.0.0 the camera streams are stored as HDF Datasets. The camera stream information is stored as attributes `'<camera-name>_Cols'`, `'<camera-name>_Rows'`, `'<camera-name>_Channels'`, `'<camera-name>_Type'`, and `'<camera-name>_Rate'`.
The video frame is provided in *OpenCV*-format. The *OpenCV*-matrix can be reconstructed by calling `cv::Mat(rows, cols, type, frameIn);`.

7.8.3 Specific File Attributes

`'RadServe_Filename'` contains the name of the file. `'RadServe_Time'` specifies the creation time of the file. `'RadServe_Version'` signifies the file version. The number of samples is stored within `'RadServe_N'`, whereas `'RadServe_NrChn'` contains the number of channels and, therefore, the number of separate HDF objects. As of RadServe version `2.5.5` the attribute `'RadServe_MeasurementStart_ms'` was added. It includes an `uint64` posix timestamp in ms of the first entry of `Mult` frames.

7.8.4 Timestamp Information

All RadServe files contain an attribute with the timestamp of the start of the file creation process. As of RadServe `2.5.5` an attribute containing the timestamp in ms was added. Additional to this timestamp a separate dataset named `'/ChnTime'` was added. This dataset contains a timestamp in ms for each packet of received data. Additionally, for each camera stream a dataset `'/<camera-name>Time_ms'` containing the timestamps of the inserted camera images is included.

7.9 Accessing HDF5 Files from Matlab

For files of version 1.0.0 the data, of a single channel, may be access from Matlab with the following function:

```
Chn1 = h5read('Traf1.h5', '/Chn1', StartFrm, NumFrms);
```

The third and fourth arguments are optional and can be used to read only parts of the table, which is useful for large files.

For files of version 2.0.0, and following, these optional arguments must be given as arrays of rank 2. For reading the complete channel the following values are required:

```
Chn1 = h5read('Traf.h5', '/Chn1', [1 1], [<N> <NumFrms>]);
```

For only reading the first 1024 frame counters the following arrays can be used: `StartFrm ... = [1 1]` and `NrFrms = [1 1024]`, whereas `StartFrm = [2 1]` and `NumFrms = [<N>-1 1]` can

be used to read the complete first frame without the frame counter.

To convert the timestamps, given as attribute `'RadServe_MeasurementStart_ms'` or within the timestamp-dataset `'/ChnTime'`, to readable date and time the following function can be used:

```
datestr(datetime(ms/1000, 'ConvertFrom', 'posixtime'),
                                    'dd-mmm-yyyy HH:MM:SS:FFF');
```

8 Camera Support

RadServe supports synchronization of image streams of USB and FLIR GigE cameras, as well as Optris cameras. Available USB cameras and detected FLIR GigE cameras are automatically inserted into the list of available cameras. Optris cameras must be added manually. One or more cameras can be selected fo adding. Each selected camera must be given a unique name as well as a camera rate. The camera rate is set in `Mult` frames. Whenever `Mult` frames were collected via USB a camera frame is requested and buffered. As of version 3.0.0 it is possible to provide the camera streams via DataPort, equal to the provided video frames during file replay. To configure the application script to additionally connect to the video ports the command

```
Brd.CfgRadServe('AddVideoToDataPort', true);
```

must be called. Otherwise, the application will not connect to the video ports when requesting data. To read the video properties (Rate, Cols, Rows, Chns, and Name) the following two commands can be used:

```
Brd.GetVideoProperties(<VidIdx>);
```

for reading the properties of a single video stream, and

```
Brd.GetAllVideoProperties();
```

for reading all available video properties.

The camera images can be display either with `Brd.DispVideo(<VidIdx>, <figure-number>)` or `Brd.DispAllVideos(<figure-numbers>)`. Instead of displaying the camera images they can also be returned by calling `Brd.GetVideo(<VidIdx>)` or `Brd.GetAllVideos()`.

RadServe sends a video frame whenever the corresponding radar frame was sent. The figure number is used to display the image without interfering with other plots. In Python the figure number is not required.

9 Remote Configuration of RadServe

With version 3.0.0 Python/Matlab commands where added to allow configuration of all GUI/console controlled options of RadServe.

`CfgRadServe(<option>, <value>)`, with the following options and values:

- `'AddVideoToFile'`: `True` or `False`, for defining, if the selected video streams should be added to the generated HDF file.
- `'AddVideoToDataPort'`: `True` or `False`, for setting, if the video streams should be provided via DataPort.
- `'AddTimStmpToFilename'`: `True` or `False`, for defining, if the names of generated HDF files should contain a timestamp.
- `'LogDataPortToFile'`: `<filename>` or `False`, for providing a name for HDF logfiles or disabling the logging to an HDF file.
- `'HdfPath'`: `<path>`, for setting the path for the generated HDF files.

`CfgRadServe_Camera(<option>, [<parameters>])`, with the following options/parameters:

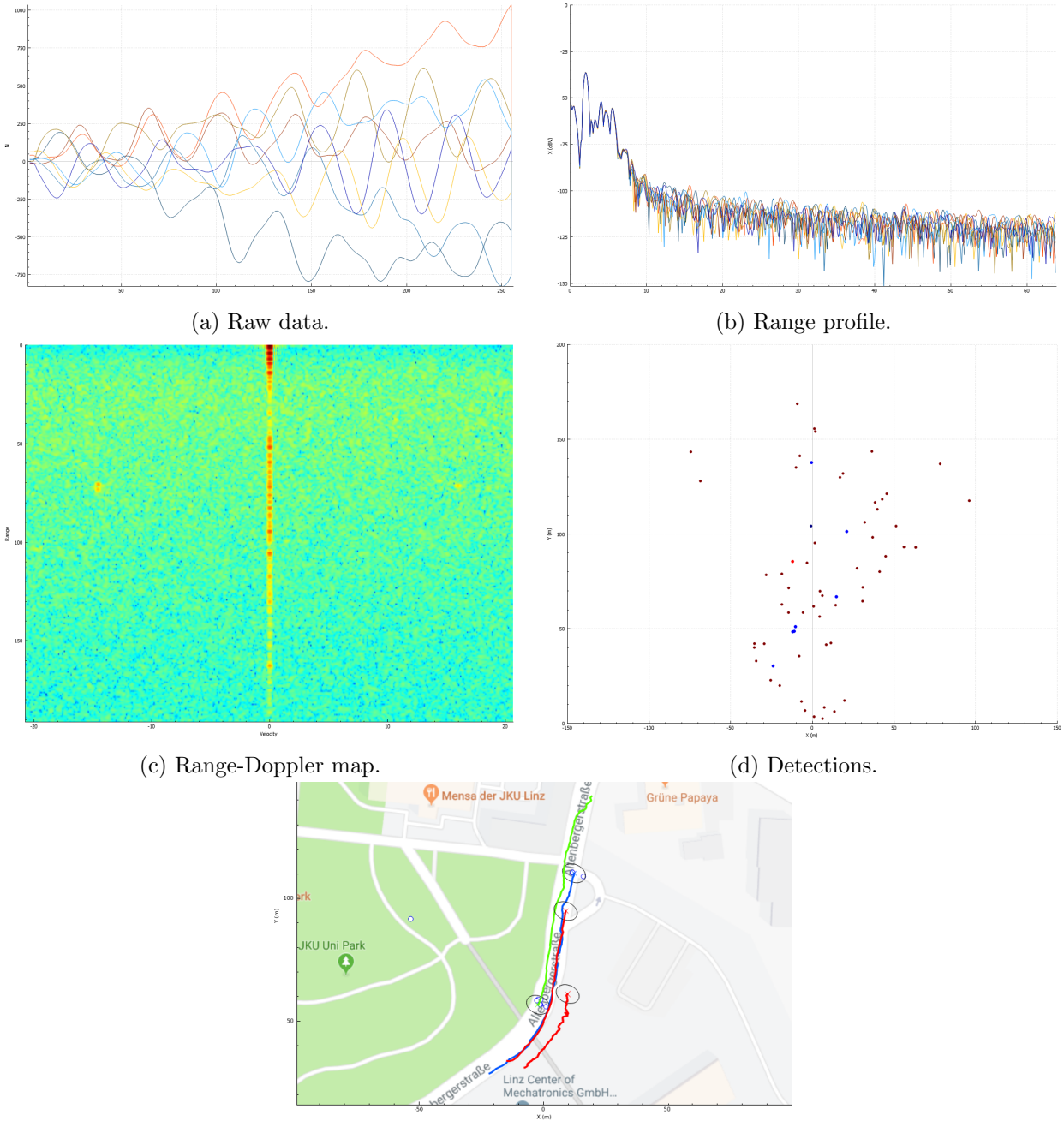
- `'List'`: without parameters, for printing the list of all selected and available cameras.
- `'DeselectAll'`: without parameters, for deselecting all cameras.
- `'Select'`: with parameters `Id`, `Rate` and `HdfName`, for selecting an available camera.
- `'Deselect'`: with parameter `Id`, for deselecting a selected camera.
- `'AddOptris'`: with parameters `Serial`, `ConfigFile`, `Emissivity`, `Transmissivity`, `ColoringPalette` and `ScalingMethod`, for adding an Optris camera and its configuration-file. Valid values for the coloring palette are: `'AlarmRed'`, `'AlarmBlue`, `'AlarmBlueHi`, `'AlarmGreen`, `'GrayBW`, `'GrayWB`, `'Iron`, `'IronHi`, `'Medical`, `'Rainbow`, and `'RainbowHi`. Valid values for the scaling method are: `'Manual'`, `'MinMax'`, `'Sigma1'` and `'Sigma3'`, as taken from the IRImager Direct SDK.
- `'UpdateOptris'`: with parameters `Serial`, `Emissivity`, `Transmissivity`, `ColoringPalette` and `ScalingMethod`, for changing the settings of an existing Optris camera.
- `'RemoveOptris'`: with the serialnumber as parameter `Serial`.

`CfgRadServe_Extension(<option>, <parameters>)`, with the following options/parameters:

- `'General'`: with parameters `Path`, `Selection`, `Parameters`: for setting the shared library of the extension, the selection value and providing the required parameters.

10 Visualization

When data is streamed via RadServe it is possible to visualize the different computational results. Fig. 20 displays examples of the different visualized data types.



(a) Raw data.

(b) Range profile.

(c) Range-Doppler map.

(d) Detections.

(e) Target tracks.

Figure 20: Visualized data.

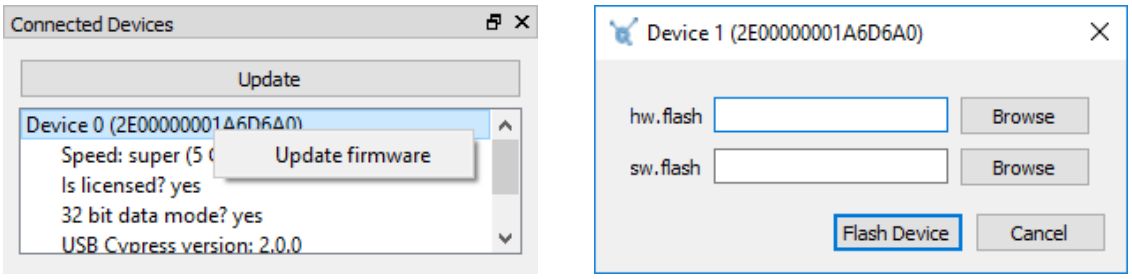
Further information on the features of the RadServe GUI is available in the **RadServe GUI Guide**.

11 Updating the Firmware of the RadarLog

The provided `hw.flash` and `sw.flash` files are used for updating the firmware of RadarLog.

11.1 GUI Version

In the GUI version of RadServe, the dialog for selecting the required files can be opened in the Connected Devices View (see Fig. 21a) by right-clicking on the desired device. This opens the *Flash Device* dialog seen in Fig. 21b.



(a) Connected Devices. (b) Flash Device.

Figure 21: Dialogs for updating the firmware.

After selecting the required files and pressing *Flash Device* a progress bar opens to display the status of the update-process. During this time no other changes can be made via the RadServe GUI.

11.2 Console Version

For updating the firmware via the Console version of RadServe the command `flash_device` can be used.

12 Troubleshooting

In this section some common problems and their solutions are described.

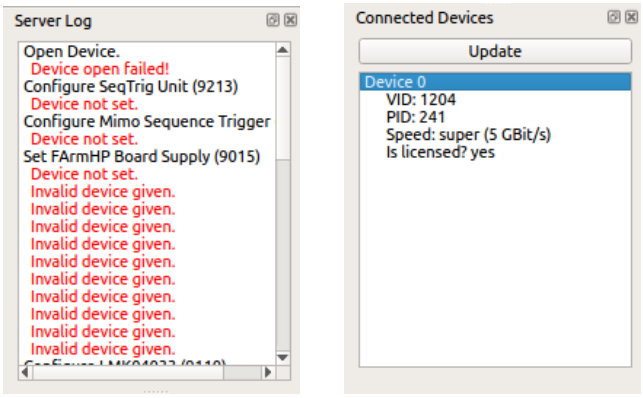
12.1 Insufficient USB Permissions (Ubuntu)

Problem: RadServe is listing the device without its id, accessing the board leads to the following error messages: `'Device open failed!'` and `'Device not set.'`, as seen in Fig. 22.

Cause: User has no permissions to access the USB device.

Solution: Run RadServe as superuser or create a udev-rule. For creating a udev-rule a file, containing the following, must be created in `/etc/udev/rules.d` (f.e. inras.rules):

```
SUBSYSTEM=="usb", ATTRS{idVendor}=="04b4",
                        ATTRS{idProduct}=="00f1", MODE:="0666"
```



(a) Server Log with errors. (b) Connected Devices.

Figure 22: Insufficient USB permissions.

Afterwards, either reboot, or reload the rules of udev with

```
$ sudo udevadm control --reload-rules && sudo udevadm trigger
```

12.2 RadServe crash on large USB Buffers (Ubuntu)

Problem: RadServe crashes after a fixed number of used packages when a large buffer is used.

Cause: On Unix memory is allocated without checking for its existence (Memory Overcommit). An exception happens only when the non-existent memory is actually used. The buffering mechanism of RadServe uses all its requested memory, therefore very large buffers can lead to an application crash.
Solution: The buffer size of RadServe (`N` × `NrChn` × `Mult` × `2`) should be limited to an available size or memory overcommit must be deactivated for the linux system.

13 Application Notes

This section gives an overview of the available RadServe-specific application notes.

AN-20 Range Profile Extension

Application note 20 is an introduction to using the RadServe extensions. It does explain the provided range profile extension example.

AN-22 Range Profile Computation

Application note 22 shows how the internal computation of RadServe can be configured to collect the range profiles of the measurement data.

AN-23 Range-Doppler Computation

Application note 23 shows how the internal computation of RadServe can be configured to collect the range-Doppler maps of the measurement data.

AN-24 Detection List Computation

Similar to the application notes 22 and 23, application note 24 shows how the internal computation of RadServe can be configured to collect a list of detections from the measurement data.

AN-25 Target Tracker Computation

Similar to the application notes 22, 23, and 24, application note 25 shows how the internal computation of RadServe can be configured to collect a list of targets tracker from the measurement data.

AN-30 Range Profile Extension File Creation, and AN-30-Replay

Application note 30 demonstrates how the range pofile extension results can be written to an HDF file, the replay application note shows how the stored data can be collected from RadServe.

AN-32 Range Profile File Creation, and AN-32-Replay

Application note 32 shows how the internal computation of the range profile is stored into an HDF file, from which it can be returned via the replay application note.

AN-33 Range-Doppler File Creation, and AN-33-Replay

Application note 33 shows how the internal computation of the range-Doppler map is stored into an HDF file, from which it can be returned via the replay application note.

AN-34 Detection List File Creation, and AN-34-Replay

Application note 34 shows how the detections are stored into an HDF file, from which it can be returned via the replay application note.

AN-35 Target Tracker File Creation, and AN-35-Replay

Application note 35 shows how the generated target tracks are stored into an HDF file, from which it can be returned via the replay application note.

AN-39 Raw File Creation for Replaying as Computation, and AN-39-ReplayAs

Application note 39 shows how an HDF file containing raw measurement data can be created. The replay application note demonstrates how an HDF file can be read, while running the measurement data through the internal computation of RadServe before returning the results to the application.