|  |  |
| --- | --- |
| (v1.0) July 31, 2019 | S. Leven, Debiotech S.A. |

 LASSO

A Lightweight ASynchronous State Observer for Embedded Systems

# Introduction

*Lasso* enables embedded systems’ developers to observe firmware state in embedded targets such as microprocessors, microcontrollers (MCU) or digital signal processors (DSP) by displaying state information in real-time on a host system such as a PC.

*Lasso* is divided in two parts. *Lasso Host* is a piece of firmware that runs on the embedded target. It retrieves *state information* and communicates through a serial communication channel with *Lasso Client. Lasso Client* runs on a PC and consumes/displays the *state information*. *State information* is any kind of accessible memory on the embedded target. While *Lasso* mainly aims at transferring *state information* from the host to the client, the client can also write *data/commands* to the host.

|  |  |  |
| --- | --- | --- |
| Image result for pc icon | *state information (downlink)*  *data/commands*  *(uplink)* | Image result for embedded programmable system icon |
| *PC running*  *Lasso Client* |  | *Embedded target running*  *Lasso Host* |

Before presenting the design goal and features of *Lasso*, let us have a look at existing alternatives for observing state information in embedded systems.

## Existing Solutions for State Observation

The following tools are typically used to accomplish state observation:

* *in-circuit debuggers* (ICD), typically connecting to a dedicated debug circuit inside the embedded target through a JTAG (“Joint Test Action Group”), SWD (“serial wire debug”) or SWIM (“single wire interface”) port. The debug circuit inside the target is also called *on-chip debugger* (OCD), and when combined with an external ICD, we call the process *on-chip debugging*. Modern *on-chip debugging* typically allows inspecting CPU registers, memory and peripherals to variable extent depending on whether the CPU is halted or running. Modifying CPU registers usually requires the CPU to be halted. A so-called *watch* can be set for memory locations of interest, and the integrated development environment (IDE) will display those memory values in formatted numerical or graphical form at a user-specified refresh rate during runtime. Communication is typically asynchronous to firmware execution.
* *debug monitors* (DM), small pieces of code residing in the target’s non-volatile memory that provide interactive communication during runtime with a host system (“semihosting”). Communication typically uses one of the target’s serial ports and is synchronous (downlink) and asynchronous (uplink) to firmware execution. Instead of a serial port, one possible implementation of a DM reroutes the embedded *printf* function to the dedicated uni-directional SWO (“serial wire output”) of the SWV (“Serial Wire Viewer”) debug interface. However, this interface is only available on higher-end ARM Cortex-M parts and only provides an uplink to the host. Another evolution of a DM is Segger’s RTT (“Real-Time Transfer”), which enables bi-directional communication with the target by using the existing JTAG or SWD interface (no extra serial port pins required). RTT supports multiple logical channels and claims to not affect real-time performance. Implementations of DMs are mainly suitable for transmitting human-readable output (ASCII strings), printed as is in the host’s debug console, and receiving keyboard input.
* *debug instrumentation* (DI), measurement hardware such as logic analyzers and oscilloscopes, which require state information to be available on I/Os of the target CPU. While this debug approach allows very good time resolution (closest to real-time), state information is essentially limited to binary or analog signals. More complex state information can only be conveyed through serial data streams and data decoders.

## Design of Lasso

*Lasso* is intended as an alternative method for state observation in embedded systems. In contrast to ICD/OCD and DI, it is a pure software product. *Lasso Host* runs alongside the target’s main application and requires some (but little) processor time as well as access to a few select peripherals (the minimum is a serial communication port and a periodic timer). In that sense, *Lasso Host* is a custom implementation of a DM with bi-directional communication. *Lasso Client* is a PC application.

The design goals of *Lasso* are the following:

|  |  |
| --- | --- |
| *Lasso Host* | *Lasso Client* |
| * Small memory footprint (little impact on target’s ROM and RAM budget) * Portable implementation (little target-specific code), written in C * Highly efficient and non-intrusive (running alongside main application with almost no impact on execution speed) * Real-time trace   + of user-selectable memory addresses   + at fixed, user-selectable refresh rates (usually < 100 Hz, sufficient for most real-world electromechanical systems) * Focus on data inspection, not text-based debug messages * Downlink to client is typically asynchronous from main application * Option for synchronous downlink * Extremely simple API | * Installable on any PC platform, installer distributed as a single executable file * User-friendly, e.g. simple, intuitive, yet interesting feature set * By default, state information is displayed in hierarchical list * Graphical display of numeric state information similar to oscilloscope * Numerical/graphical inspection of numeric arrays * No provision for a text-based debug console * Full logging facility with bindings to established spreadsheet software * Option for asynchronous uplink to host * Transparent operation over serial port * Support for user plugins * Support for 3rd party remote control |

# Distribution

*Lasso host*

*Lasso Host* is open source.

Download source files from [http://srvrepos/svn/ELEC/Tools/Lasso host](http://srvrepos/svn/ELEC/Tools/Lasso%20host) or

Download source files from <https://github.com/sleven79/lasso-host>

Interactive code generator: <https://sleven79.github.io>

*Lasso Client*

*Lasso Client* is distributed as a Windows executable (~50Mbytes), with sources being private so far.

Download installer from [http://srvrepos/svn/ELEC/Tools/Lasso client](http://srvrepos/svn/ELEC/Tools/Lasso%20client) or

Download installer from <https://github.com/sleven79/sleven79.github.io>

# Setup

## Lasso Host

In order to use *Lasso Host*, its source files must be included in your target application. For most modern IDEs, it does not matter whether you place the files in your application’s source folder or anywhere else. However, they must be linked into your project’s source file tree in order to be compiled with your application.

A typical *Lasso Host* source file tree looks as follows:

* Lasso host
  + config
    - config\_host.h
  + target
    - lasso\_host\_targetXX.c
  + lasso\_errno.h
  + lasso\_host.c
  + lasso\_host.h
  + lasso\_version.h

*Lasso Host* needs to be configured for each specific target and application. All configuration settings can be found in the “config\_host.h” file in the “config” folder. For each setting, a nominal value is provided with comments on the admissible range, refer to Section §3.1.1. This is only a header file, so writing custom code is not required here.

The second source file that requires adaptation is in the “target” folder and must be adapted to the target hardware. Target code templates exist for Cypress PSoC4, PSoC5, Renesas RX, Texas Instruments TivaTM4C and AM335x (BeagleBone). Note that *Lasso* has only been tested on 32-bit targets. For details, refer to Section §3.1.2.

When using floating point numbers, make sure that the linker pulls in floating point support (for *printf* and *scanf*). For ARM GCC linkers, this can be achieved by setting the linker command line flags “-u \_printf\_float” and “-u \_scanf\_float”. Sometimes, instead of command line flags, the IDE offers checkboxes for these settings.

*Lasso Host* uses a certain amount of heap memory (dynamic allocation memory) through calls to *malloc()*. Make sure that sufficient heap memory is available. The minimum heap allowance should be 1 kByte. Depending on the number of variables declared for transfer on the downlink, a few hundred Bytes of heap will be sufficient. However, it is good practice to allow for at least 1-4 kBytes of heap to be on the safe side. Required heap size can be estimated using the following formula:

Heap [Bytes] = 32 Bytes \* #Vars + sizeof(Vars) + 256 Bytes

E.g. from #Vars = 4 and sizeof(Vars) = 128 Bytes follows Heap = 512 Bytes.

### Lasso Host Configuration

All *Lasso Host* configuration is in the “config\_host.h” template file, which provides a default set of settings suitable for most projects.

One of the main parameters of *Lasso Host* is LASSO\_HOST\_TICK\_PERIOD\_MS. Note that a tick in *Lasso* stands for the period at which state information is sent from the client to the host, and this period is configured in milliseconds. Timing related configuration parameters in *Lasso* are always specified either in ticks, or in milliseconds.

Another important parameter if LASSO\_HOST\_TIMESTAMP. If 1, *Lasso Host* creates a timestamp based on its internal tick counter and provides this timestamp as state information to the client.

For more info on *Lasso Host* configuration parameters, please refer to §6 or directly to the comments in the configuration template file. More details will also follow in future versions of this document.

### Target-specific Code

All target-specific code is defined in the “lasso\_host\_targetXX.c” template file. Code covers

1. setting up a serial communication port through the *lasso\_comSetup\_targetXX()* function
2. sending serial (downlink) data through the *lasso\_comCallback\_targetXX()* function
3. a CRC generator function, if required, through the *lasso\_crcCallback\_targetXX()* function

Have a look at the existing code examples to start your own implementation.

### Other Code Requirements

#### For the Downlink (Host to Client)

Lasso requires a periodic call to the communication function *lasso\_hostHandleCOM()* that sends state information to the client. Usually, this call would be located in an interrupt service routine (ISR). The following code example shows an implementation on Cypress PSoC:

CY\_ISR(LASSO\_ISR) {

#if (LASSO\_HOST\_ISR\_PERIOD\_DIVIDER > 1)

static volatile uint8\_t lasso\_divider = 0;

if (lasso\_divider == 0) {

lasso\_divider = LASSO\_HOST\_ISR\_PERIOD\_DIVIDER;

#endif

lasso\_hostHandleCOM();

#if (LASSO\_HOST\_ISR\_PERIOD\_DIVIDER > 1)

}

lasso\_divider--;

#endif

}

It is not mandatory to have a dedicated ISR for *Lasso*. Instead, an existing, periodic ISR of the main application may be used. However, the frequency of this ISR might not match the desired tick frequency of *Lasso*. Therefore, the example above also demonstrates the use of configuration parameter LASSO\_HOST\_ISR\_PERIOD\_DIVIDER. This parameter can be used to scale down the frequency of calls to *lasso\_hostHandleCOM()*.

#### For the Uplink (Client to Host)

Lasso requires frequent, but not necessarily periodic, calls to the communication function *lasso\_hostReceiveByte()* that reads incoming commands send by the client. Usually, these calls would be done from the user-application’s main loop. The following code example shows an implementation on Cypress PSoC:

for(;;) {

// forward all received characters to lasso host

if (LASSO\_UART\_GetRxBufferSize()) {

lasso\_hostReceiveByte(LASSO\_UART\_GetChar());

}

/\* Place your application code here. \*/

}

Note that above implementation shows an example, where incoming data is read Byte per Byte from within the main thread. Usually, incoming serial data is stored in a FIFO buffer on the target hardware. If the FIFO buffer fills up faster than one main loop cycle, a buffer overrun will occur and incoming data will be lost. To counteract this problem, do one of the following:

* Implement a fast main loop
* Ensure that the FIFO buffer is large enough to hold the longest frame of incoming data (see LASSO\_HOST\_COMMAND\_BUFFER\_SIZE defined in the host’s config file, usually 16 Bytes)
* Implement a different strategy of reading from the FIFO buffer (e.g. by background DMA transfer, or an ISR with priority over the main thread)

In case you opt for the solution with a large FIFO buffer, where data is read Byte per Byte from the FIFO in the main loop, also make sure that LASSO\_HOST\_COMMAND\_TIMEOUT\_TICKS is defined large enough in the host’s config file. Consider the following equation:

mainloop cycle duration <

LASSO\_HOST\_COMMAND\_TIMEOUT\_TICKS \* LASSO\_HOST\_TICK\_PERIOD\_MS

Each time a new Byte is read from the FIFO and handed over to *Lasso Host* with lasso\_hostReceiveByte(), *Lasso Host* resets its internal timeout counter.

### Application Programmer’s Interface (API)

The API of Lasso Host is extremely simple. Apart from the calls to both communication functions seen in Sections §3.1.3.1 and §3.1.3.2, the user-application only needs to register the target specific functions of Section §3.1.2, register the state information to be downloaded to the client and call the function that finalizes the setup of *Lasso*’s memory spaces.

The API is presented in the order of logical calls to the API function from a user-application:

#### lasso\_hostRegisterCOM()

Signature:

int32\_t lasso\_hostRegisterCOM (

lasso\_comSetup cS,

lasso\_comCallback cC,

lasso\_actCallback aC,

#if (LASSO\_HOST\_STROBE\_CRC\_ENABLE == 1) || (LASSO\_HOST\_COMMAND\_CRC\_ENABLE == 1)

lasso\_perCallback pC,

lasso\_crcCallback rC

#else

lasso\_perCallback pC

#endif

);

This function registers the target-specific callbacks required by *Lasso Host*.

cS is the target-specific callback that sets up the serial communication link on the host side.

cC is the target-specific callback that actually performs downlink transmissions on the host.

aC is an optional application-specific callback that gets called each time the client activates or deactivates state information transfer. This parameter can be NULL.

pC is an optional application-specific callback that gets called each time the client changes the download period. This parameter can be NULL.

rC is a target-specific CRC generator function. This parameter is only required if CRC generation has been configured at least for one of strobe or command packets.

This function immediately calls cS and returns any error code generated by cS. If cS returns successfully, this function either also returns 0 or the error code EINVAL, if one of the required callbacks is NULL.

#### lasso\_hostRegisterCTRLS()

Signature:

int32\_t lasso\_hostRegisterCTRLS (lasso\_ctlCallback cC);

This function registers yet another user-specified callback, which is called when the client sends a specific controls (CTRLS) command to the host. Also refer to Section §4.5.2. Returns either 0 on success or EINVAL when cC is NULL.

#### lasso\_hostRegisterDataCell()

Signature:

int32\_t lasso\_hostRegisterDataCell (

uint16\_t type,

uint16\_t count,

const void\* ptr,

const char\* const name,

const char\* const unit,

#if (LASSO\_HOST\_STROBE\_DYNAMICS != LASSO\_STROBE\_DYNAMIC)

const lasso\_chgCallback onChange

#else

const lasso\_chgCallback onChange,

uint16\_t update\_rate

#endif

);

*Lasso Host* registers state information for download to the client. Each piece of state information is registered in the form of so-called “DataCells”. The main purpose of each DataCell is to point to the associated state information in host memory space. Also, DataCells inform about the data type and array size of the state information and associate a name, unit and update rate with it. Note that each pieces of state information can only have a single type.

For type, specify:

|  |  |
| --- | --- |
| LASSO\_BOOL | True / False |
| LASSO\_CHAR | Single ASCII character or ASCII array |
| LASSO\_UINT8 | 0 … 255 |
| LASSO\_INT8 | -128 … 127 |
| LASSO\_UINT16 | 0 … 65535 |
| LASSO\_INT16 | -32768 … 32767 |
| LASSO\_UINT32 | 0 … 4294967295 |
| LASSO\_INT32 | -2147483648 … 2147483647 |
| LASSO\_UINT64\* | 0 … 264-1 |
| LASSO\_INT64\* | 263 … 263-1 |
| LASSO\_FLOAT | IEEE 754 single precision floats |
| LASSO\_DOUBLE\* | IEEE 754 double precision floats |

\*if available on target

You can combine the type specification with other DataCell settings:

|  |  |
| --- | --- |
| LASSO\_DATACELL\_ENABLE | Indicates that, by default, underlying state information is selected for download in *Lasso Client* |
| LASSO\_DATACELL\_WRITEABLE | Indicates that underlying state information can be written to by *Lasso Client* |
| LASSO\_DATACELL\_PERMANENT | Indicates that underlying state information cannot be deselected from download by *Lasso Client* |

Example combination: type = LASSO\_UINT32 | LASSO\_DATACELL\_ENABLE

For count, specify the number of memory cells of type type to download.

For ptr, specify the address at which the memory cell (or first memory cell of an array) is located.

For name, specify a name string (ROM).

For unit, specify a unit string (ROM).

For onChange, specify your own callback or NULL. This callback will be called when the client initiates a write to the underlying memory cell, but before executing the write (the callback can actually validate the new value to be written).

For update\_rate, specify the period at which the underlying state information is transmitted to the client. This parameters is only available under certain conditions, in particular when Lasso Host is configured for dynamic strobing (see §x.x). As an example, update\_rate = 3 means that state information is transmitted every third *Lasso Host* tick.

The function returns 0 on success or one of the error codes ENOMEM or EFAULT.

#### lasso\_hostRegisterMEM()

Signature:

int32\_t lasso\_hostRegisterMEM (void);

Finalizes the setup of *Lasso Host*’s memory spaces. Returns 0 on success or error code ENOMEM.

#### lasso\_hostReceiveByte()

Signature:

int32\_t lasso\_hostReceiveByte (uint8\_t b);

Transfers a Byte b received on the target-specific serial uplink to *Lasso Host*. It is up to the user to retrieve Byte b from the uplink. Returns 0 on success or one of the error codes ENODATA, EILSEQ or ENOSPC.

#### Lasso\_hostSetBuffer()

Todo

#### Lasso\_hostCountdown()

Todo

#### lasso\_hostTickPeriod()

Todo

#### lasso\_hostHandleCOM()

Signature:

void lasso\_hostHandleCOM (void);

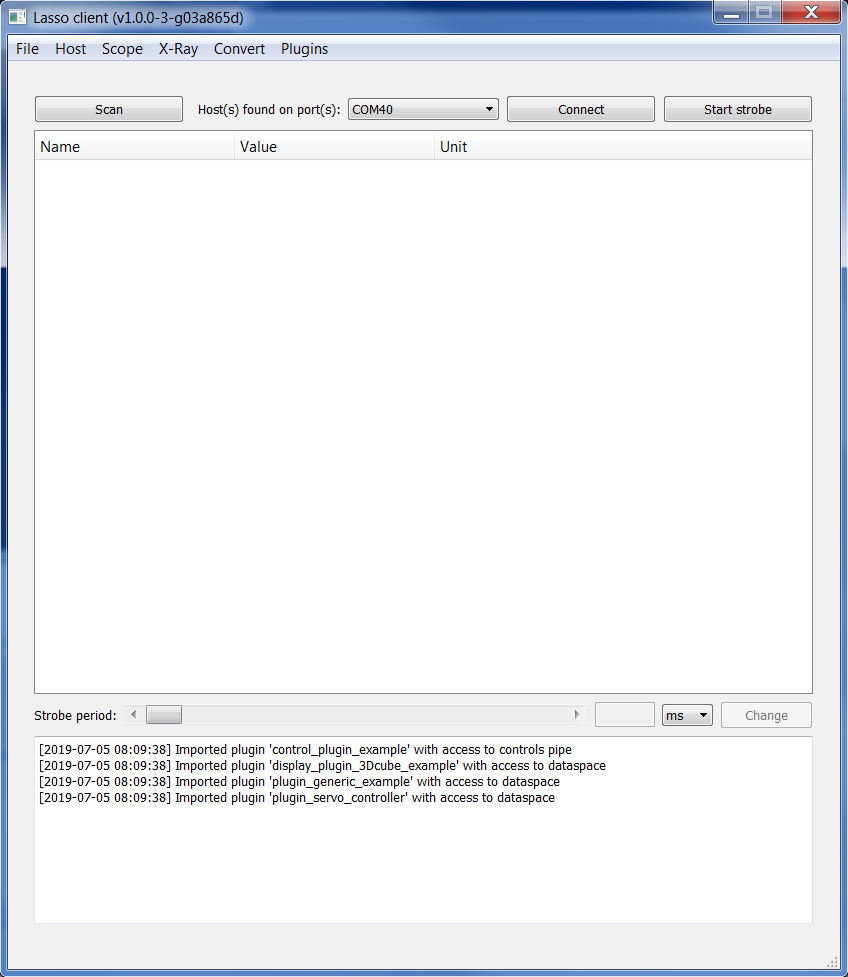
The central *Lasso Host* function for sending state information to the client across the serial downlink.

### Callbacks

Todo

## Lasso Client

Once *Lasso Client* is installed on a PC (so far, only Windows installers are available), it can immediately be used to scan for available hosts, connect to them and download state information in real-time. A first scan is automatically launched at startup, *and Lasso Client* opens as follows:



Menu bar

Console

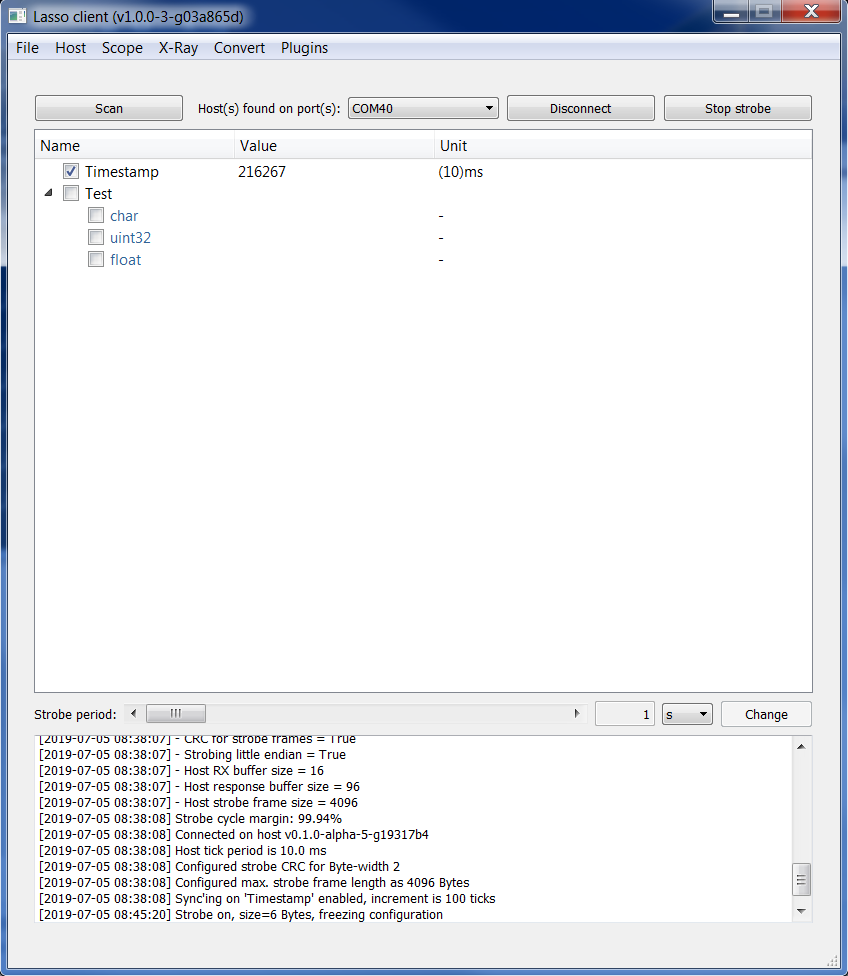
Main window for hierarchical tree view

of state information

After hosts are found, it is possible to connect to them. Note that each *Lasso Client* only permits one host-connection at a time. However, after disconnecting, it is possible to select another host and connect to it. Also, it is possible to use multiple *Lasso Clients* at the same time.

On connecting, *Lasso Client* displays the major configuration parameters detected for the selected host, which (must) correspond to the settings in the “config\_host.h” file on *Lasso Host*, as well as the host’s firmware revision number and the current tick period.

Once connected, state information declared for download on *Lasso Host* is displayed in a tree view in the main window. In the example below, state information consists of a timestamp that increments every 10ms and three test variables with different types (char, uint32 and float). By default, the timestamp is enabled for download while the three test variables are not. In *Lasso Client*, the fact to download state information is also called “strobing”. Each periodic snapshot of state information send out by the host is also called a “strobe”. Before strobing is started, only the structure of state information is displayed in *Lasso Client*, but not the values. When clicking on “Start strobe”, the state information in the main window updates at the configured refresh rate (“strobe period”).



Checkboxes for (de)activation of

state information in strobe

Note that the current set of state information that is activated for download cannot be changed while strobing. Stop strobing, and then activate/deactivate individual state information for download by using the associated checkboxes in the tree view.

The following chapters will present the major features of host-client interaction, the different modes available for displaying state information, and finally, advanced features.

# Using Lasso

## Declaring State Information for Download

On *Lasso Host*, use the lasso\_hostRegisterDataCell() function to declare state information for download, refer to Section §3.1.4.3. Note that this allows state information to become available for download, but not necessarily activated for download, if the LASSO\_HOST\_DATACELL\_ENABLE flag is not set.

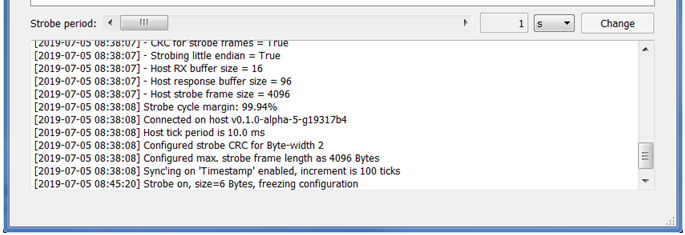
On *Lasso Client*, all state information declared by the host is eligible for download. When state information is declared but not activated for download, such state information can be activated for download by clicking the appropriate checkbox in *Lasso Client*. Note that changing the activation status is only possible when strobing is inactive.

State information may be declared “permanent” by the host by setting the LASSO\_HOST\_DATACELL\_PERMANENT flag. This means that it cannot be deactivated from download in *Lasso Client*. In that case, the checkbox usually associated with each DataCell will not be present.

State information may be declared “writeable” by the host by setting the LASSO\_HOST\_DATACELL\_WRITEABLE flag. This means that the user can modify the content of the underlying memory on the host by entering and transmitting data from *Lasso Client*. By default, state information cannot be written to.

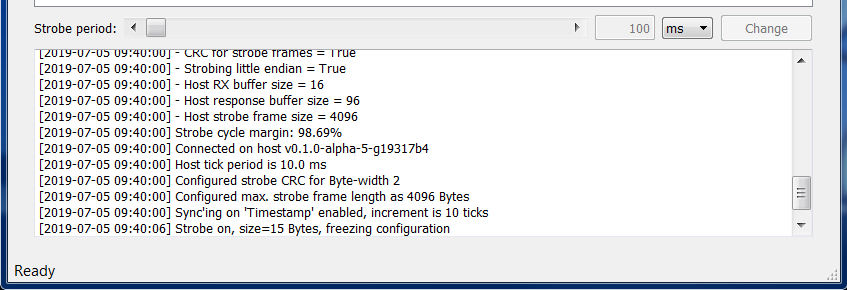
## Inspecting Serial Link Bandwidth Margin

On *Lasso Client*, after connecting to a host, observe the value indicated by “Strobe cycle margin” in the console of the client’s main window.



The typical baudrate that *Lasso* uses for its serial communication channel is 115.2 kBaud. Here, each strobe including payload and CRC has a width of 6 Bytes (this can also be seen in the console window once strobing has been started). With 6 Bytes, each Byte being encoded as 10 bits (start bit, 8 bits payload, stop bit, no parity), 60 bits are transmitted every second (every 100 ticks and a tick period of 10ms). This yields a link occupation of 0.06%, or link margin of 99.94%.

In below example, a strobe of 15 Bytes on the same serial communication link, but with a strobe each 10 ticks (100ms), yields a link occupation of 1.31%, or link margin of 98.69%.



While there is still plenty of margin, note that a baudrate of 115.2 kBaud allows for about 11500 Bytes to be transmitted each second. If your setup comes close to link budget, the baudrate should be increased. However, this feature is currently not available on the *Lasso Client* side.

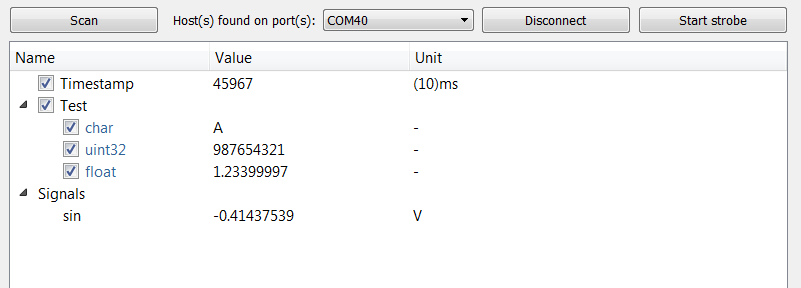
## Displaying State Information

### Hierarchical View

The hierarchical tree view is the most basic way to visualize state information and works with all data types. When a *Lasso Client* connects to a host, it starts by discovering the host’s configuration parameters as well as the state information the host has to offer. In the view below, the *Lasso Host*’s internal timestamp is the first available state information. User-declared test variables follow behind. Note the hierarchy levels. A DataCell declared with name “Test.char” in *Lasso Host* will appear under name “char” in group “Test” in *Lasso Client*. “Signals.sin” translates to name “sin” in group “Signals”. The naming hierarchy can span multiple levels by adding one or more “.” in the DataCell name string.

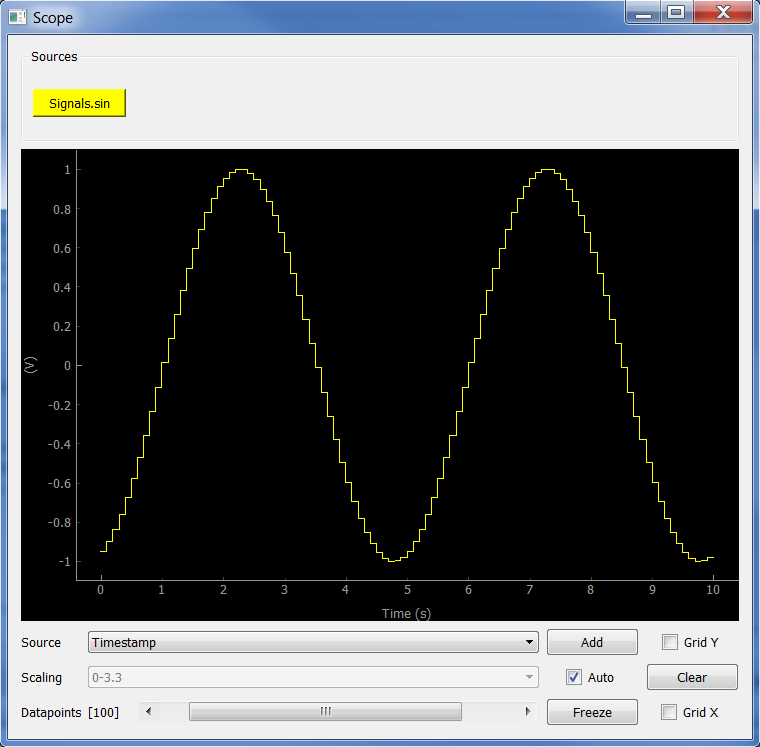
For state information that can be (de)activated for download on the client side, a checkbox is displayed on the left of the name string. Note that in the example below, “Signals.sin” has been declared “permanent” on the host side, so the checkbox is absent (see §4.1).

State information name strings as well as units will be displayed as soon as there is a connection with the host. The values of activated state information will only be updated when strobing is active.



### Scope View (“Scope”)

For numeric state information, a visualization function similar to an oscilloscope exists. One or more scopes can be opened by clicking “New” under “Scope” in the main menu bar.



In the “Source” dropdown menu below the plot pane, one can select a source from all available numeric state information. By clicking the “Add” button, a new source is added to the graph. For this source, the button name will then change to “Remove”.

In addition, a toggle button appears in the “Sources” bar at the top of the window. Clicking this button hides the trace from view (but keeps it and updates it in internal memory). Clicking the button again makes it reappear. This can be practical when visualizing many traces at the same time (up to 6 traces can be displayed simultaneously right now).

The “Scaling” dropdown menu is disabled by default, since auto-scaling is active. Deselect the “Auto” tickbox to enable manual scaling. The dropdown menu proposes a selection of common ranges for the ordinate (y, vertical axis). The abscissa (x, horizontal axis) scaling can be changed by the “Datapoints” slider. In the above example, with a strobe period of 100ms, 100 datapoints correspond to 10s time. The number of datapoints can be changed between 10 and 10000.

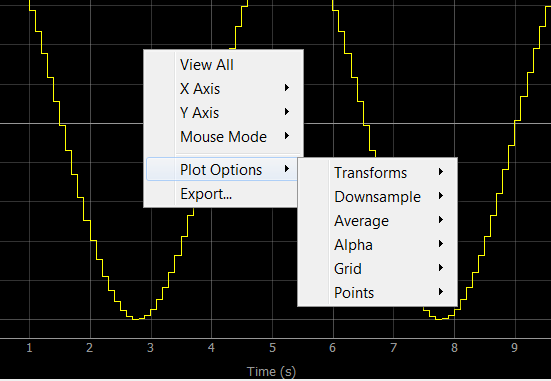
Button “Clear” allows to clear all plot data from the graph.

The “Grid X” and “Grid Y” tickboxes will enable horizontal and vertical grid lines (spacing of grid lines is automatic, and adjusts to the axes ticks).

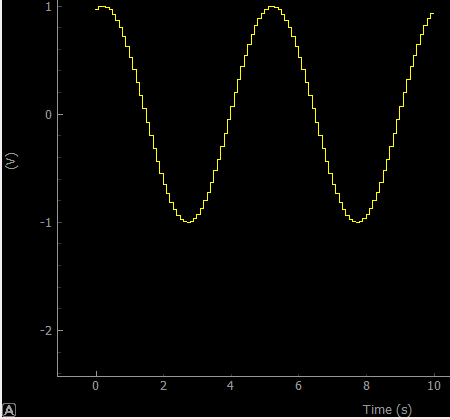
Button “Freeze”, on single click, will freeze the plot pane once the end of the time axis has been reached. During this lapse of time, the button will blink green and a second click will freeze the plot pane immediately, making the button turn red. Another click returns the scope to normal behavior.

Note that the scope’s plot pane is derived from a pyqtgraph object (<http://www.pyqtgraph.org>). This means that one can interact with the graph in real-time. A mouse click “right” opens pyqtgraph’s interactive menu. This enables

* Inverting axes
* Log-scale axes
* Configuring grids, axis ranges and pan-modes
* Export plot data as image files (PNG, TIF, JPG, … ), scalable vector graphics (SVG), or CSV
* And more …



Also noteworthy are the scaling and panning features of pyqtgraph. Use the mouse wheel to scale the plot pane in real-time. Use the mouse’s “left” button to pan the plot pane. This will override the auto-scaling feature and a small box “[A]” appears in the lower left corner of the plot pane:



By clicking on the “[A]” the plot pane returns to auto-scaling and panning is reset to the origin.

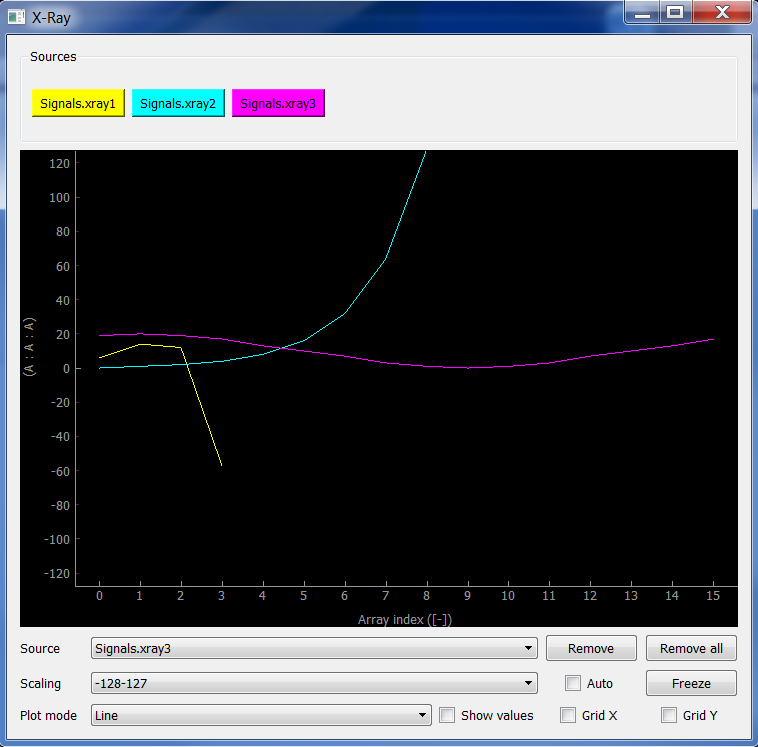
### Array Inspector View (“XRay”)

For numeric, array-like state information, a graphical or tabular visualization function exists. One or more array inspectors can be opened by clicking “New” under “X-Ray” in the main menu bar.

In contrast to the scope view, the array inspector view does not have a timeline as abscissa (x, horizontal axis), but the array index. The range on the abscissa adjusts to the source with the largest array index.

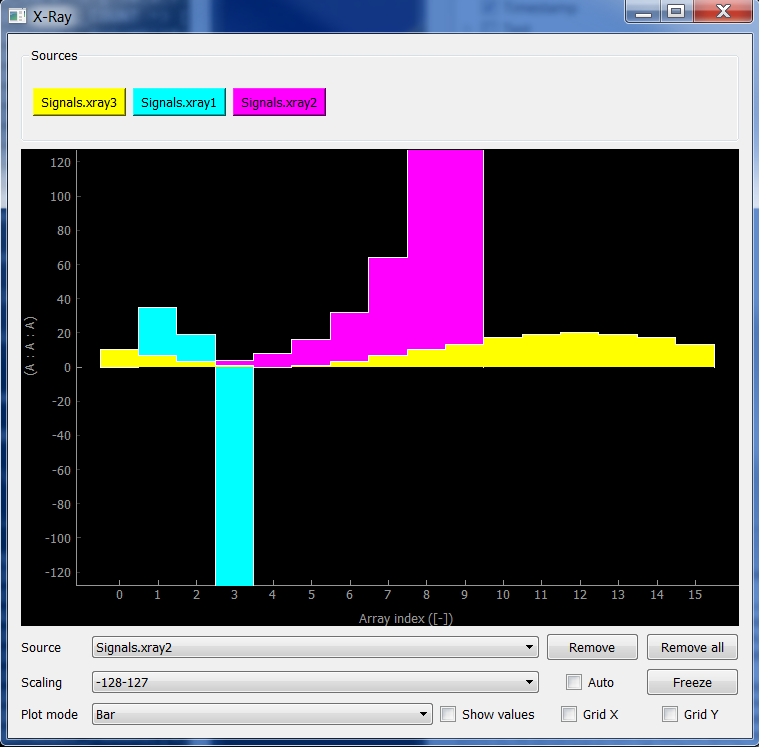
Adding sources to the array inspector view is analogous to the scope view (§4.3.2) by using the “Add” button. Once added, sources can be removed individually (“Remove” button) or all sources can be removed together (“Remove all” button).

The scaling features of the ordinate (y, vertical axis) with the “Scaling” dropdown menu as well as the grid tickboxes are also identical with the scope view. The freeze function does not have an intermediate state as in the scope view. A click on the “Freeze” button freezes the view immediately.

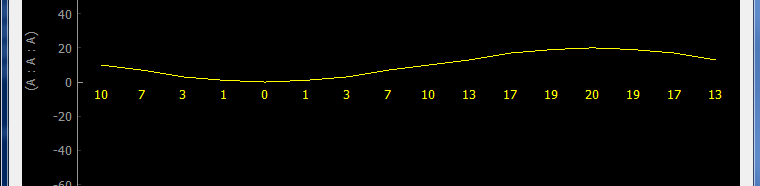


Except for a different use of the abscissa in the graphical visualization function, the major difference between the scope view and the array inspector view is the possibility to select between three plot modes. Above example shows the “Line” plot mode. There are also “Bar” and “Table” plot modes. Switching between plot modes is possible at any time.

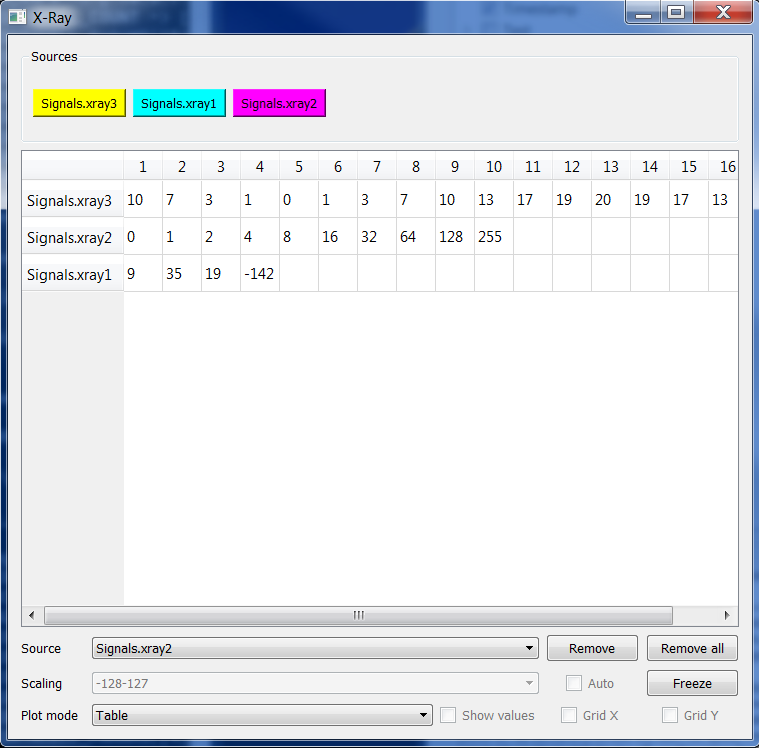
While the “Line” plot mode draws lines connecting two consecutive array values, the “Bar” plot mode draws independent bars for all array values. The indices in this plot mode are centered on the bars.



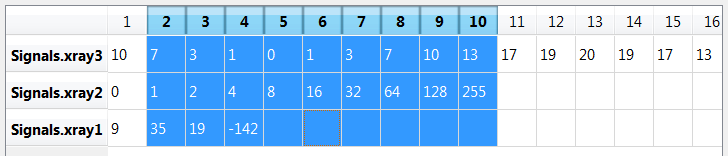
In “Line” and “Bar” plot modes, it is also possible to visualize the numeric value at each array index by clicking on tickbox “Show values”. The numeric values will appear with the same color as the graphical trace and slightly above or below the origin of the y-axis depending on the sign of the trace at each index.



The “Table” plot mode shows arrays in tabular (spreadsheet) form. The rows in the table are organized automatically such as to show the array with the most entries on top and other array in decreasing order on the following rows.



The integrated “selection” and “save” features of pyqtgraph in “Table” plot mode (see below) allow copying of all or part of the array values to the Windows clipboard. Saving to a .tsv file (as tab-separated values) is also possible.



For the “Line” and “Bar” plot modes, the same interactive options as in scope view mode are available, in particular the export as image files (PNG, TIF, JPG, … ), scalable vector graphics (SVG), or CSV.

## Changing the Strobe Period

As soon as a *Lasso Host* is connected, *Lasso Client* offers to adjust the strobe period within the limits determined during the host discovery process (these limits are fixed in the configuration file on the host side). In the main window of *Lasso Client*, the scrollbar “Strobe period” and the time unit dropdown menu can be used to configure a new strobe period. When a new strobe period value is selected, a change request must be sent to the host by clicking the “Change” button within a time laps of 10 seconds (otherwise, the current strobe period is maintained).

The new strobe period remains valid until the host resets (e.g. due to a power cycle), event upon which the strobe period return to its default value. However, a new strobe period configuration is not lost when disconnecting and reconnecting to the same host during the same power cycle.

In the example below, the configuration file on the host side held the following #defines:

LASSO\_HOST\_TICK\_PERIOD\_MS (10)

LASSO\_HOST\_STROBE\_PERIOD\_MIN\_TICKS (10)

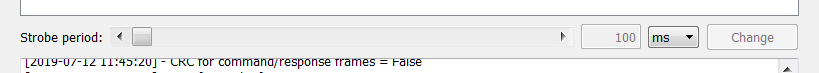
LASSO\_HOST\_STROBE\_PERIOD\_MAX\_TICKS LASSO\_STROBE\_SLOWEST

LASSO\_HOST\_STROBE\_PERIOD\_TICKS LASSO\_HOST\_STROBE\_PERIOD\_MIN\_TICKS

where LASSO\_STROBE\_SLOWEST is the upper limit of a 16-bit uint: 65535 (from lasso\_host.h).

The above settings mean that the minimum strobe period is 100 ms (10x 10 ms), while the maximum period is 655.35s (65535x 10 ms, rounded down to 10 min). This would be the absolute maximum that *Lasso* allows for a tick period of 10 ms. Therefore:

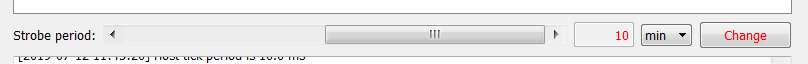
strobe period in the milliseconds range (floor is 100 ms):



strobe period in the seconds range (from 1 to 59 s):



strobe period in the minutes range (ceiling is at 10 mins).



Changing the strobe period can be done at any time, while strobing is activated or not.

## Logging State Information

*Lasso Client* automatically logs state information as soon as strobing is started. Log files can be found in folder “/logs” of *Lasso*’s installation directory. Each log session between clicking on “Start strobe” and “Stop strobe” results in two output files, one with the recorded raw data and one with a data descriptor, whose filenames have a unique timestamp. Example:

* lasso\_data\_D20190712\_T143839.log (raw data, no human readable information)
* lasso\_data\_D20190712\_T143839.txt (data descriptor file, human readable)

The timestamp in the filename has the date-time format Dyyyymmdd\_Thhmmss with resolution down to the second.

The raw data file contains each and every Byte received through strobing from the host. Data is uncompressed, but compact; there are neither line breaks nor other formatting characters. Raw data cannot be imported meaningfully by third-party tools without prior conversion.

The data descriptor file contains a JSON dump of a Python dictionary with the following entries:

* the start time (same as the timestamp in the filename)
* the strobe period in milliseconds
  + **Note: this is the strobe period at the time strobing was started. If the strobe period is changed while strobing, the new strobe period is not recorded!**
* a description of each datacell activated for strobing, including
  + array size
  + Byte width
  + format string
  + name and unit string
  + type
  + **Note: the set of datacells activated for strobing cannot be changed once strobing has started. This is why the information about datacells contained in the data descriptor file is valid for the entire record until strobing stops.**

An example of a data descriptor file is shown below:

{

"starttime": "2019-07-12 14:38:39",

"timestamp": {

"ms": 1000

},

"datacells": {

"0": {

"array size": 1,

"byte width": 4,

"format": "<L",

"name": "Timestamp",

"type": "UINT32",

"unit": "(10)ms"

},

}

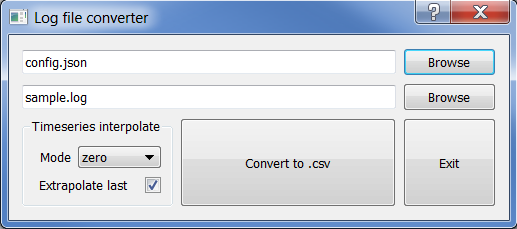
}

### Converting Logs to Spreadsheets

The main purpose of log files is to preserve data for future processing with third party tools, in particular spreadsheet tools such as MS Excel.

As discussed above, the raw log files do not contain any human readable data format nor spreadsheet compatible data. A conversion is necessary. Fortunately, *Lasso Client* provides the necessary tools to perform this conversion.

A click on “Convert” in *Lasso Client*’s main menu bar opens the following dialog:



In order to transform a given .log file into a spreadsheet compatible .csv file, a config file needs to be provided that controls the conversion process (“config.json” in the example above). The config file instructs the conversion tool how to interprete the source data, how to translate source data to output data, and how to format the output data. An example configuration file is presented and explained below:

{

"author": "me",

"date": "2017-11-29",

"constants": {

"\_HSC\_RESOLUTION": 16384,

"\_HSC\_RANGE\_MIN" : "0.1\*\_HSC\_RESOLUTION",

"\_HSC\_RANGE\_MAX" : "0.9\*\_HSC\_RESOLUTION",

"\_MMHG\_RANGE\_MIN" : 0.0,

"\_MMGH\_RANGE\_MAX" : 1551.5,

"\_TIMESCALE" : 1000

},

"columns": {

"0": {

"name": "Timestamp",

"unit": "s",

"source": "Timestamp",

"scale" : "\_TIMESCALE",

"format": "{:.3f}"

},

"1": {

"name": "ADC0",

"unit": "V",

"source": "ADC0",

"range": {

"source min": 0,

"source max": 3.3,

"destination min": 0,

"destination max": 3.3

},

"format": "{:.3f}"

"interpolate": "linear",

}

"2": {

"name": "ArrayData",

"unit": "lsb",

"source": "SampleArray",

"range": {

"source min": -128,

"source max": 127,

"destination min": -256,

"destination max": 255

},

"format": "{:d}",

"timeseries": "True"

}

}

}

Like the data descriptor file, the format of the config file is a JSON dump of a Python dictionary. The entries “author” and “date” are informative only, they are not required for conversion and can contain any kind of data, typically a string, though.

Under “constants”, one can add any constant numeric values that will be used for the conversion of source to output data. Valid entries are numeric constants as well as string expressions that are parsed by *Lasso Client* to yield other numeric constants. Inside string expressions, other numeric constants may be used, in addition to the following symbols:

( ) - + / \*

This allows composing simple mathematical expressions that help map source data to output data. Possible examples are range conversions and the addition/subtraction of an offset.

Under “columns”, one configures the actual output columns. In the above example, column “0” will have title “Timestamp [s]” (a combination of name and unit strings). The source data for column “0” comes from datacell “Timestamp”. In fact, the converter tool of Lasso Client looks up the entry for datacell “Timestamp” in the corresponding data descriptor file in order to determine where to look for the underlying “Timestamp” values in the raw log file.

The next entry for each column is either a “scale” entry or a “range” entry. The “scale” entry has a single numeric value or string expression. Output data is computed from source data by multiplying by this value. If a “scale” entry is provided, it takes priority over the “range” entry.

A “range” entry is slightly more complicated, it usually has four (optional) sub-entries:

source\_min, source\_max, destination\_min, destination\_max

In analogy to the « scale » entry, all four must have a single numeric value or string expression. They can be omitted, in which case default values are used (source\_min=0, source\_max=1, destination\_min=0, destination\_max=1).

The mapping from source to output data, when using the “range” option, is as follows:

output = (source – source\_min) \* (destination\_max – destination\_min) / (source\_max – source\_min) + destination\_min

Note that for datacells where lowerCaseOf(name of datacell) is “timestamp”, a “scale” entry must be present. A “range” entry may be present, but is not taken into account.

Next, each column has a « format » entry that is used to convert numerical output data to human readable data. Standard Python format string notation is used, e.g. “{:.3f}” for floating point output with three digits behind the comma or “{:3d}” for integer output with at least 3 digits (unused digits are replaced by spaces).

Finally, there are two optional fields that can be used for each column specification. The first field is the “timeseries” field, which can have values “True” or “False”. If this field is not present, the default interpretation is “False”. The notion of timeseries only applies to arrays that are recorded by *Lasso*. Usually, arrays would be considered as being flat, that is, all values in the array have the same timestamp (the timestamp of each strobe). However, *Lasso host* might be configured to record timeseries data into arrays, that is, data that has been recorded at a faster period than the strobe period. For example, a pressure sensor is sampled by the host every 10ms, while the strobe period is 100ms. Then, the array that is transmitted with each strobe contains 10 entries with a 10 times faster period than the strobe.

The log file converter of *Lasso client* can deal with this situation if the field “timeseries” for the specific column is set to “True” in the converter’s config file. For each strobe record, Lasso will generate as many log file rows as there are array entries. All strobe data that is not “timeseries” data, or “timeseries” data with fewer rows, will then have to be interpolated. This is where the second optional field becomes valuable.

The “interpolate” field determines how column data is stretched by interpolation in order to align with the longest “timeseries” data. In case there is no “timeseries” data, interpolation does not apply. The “interpolate” field must have one of the following values:

“zero”, “linear”, “quadratic”, “cubic”, “previous”, “next”

The first four options apply 0, 1st, 2nd, 3rd order spline-interpolation. The other two apply step interpolation by maintaining the previous or next known value for each interpolation interval.

If no individual “interpolate” option is specified for each column, the global interpolation mode setting from the converter menu is used (see picture of menu above).

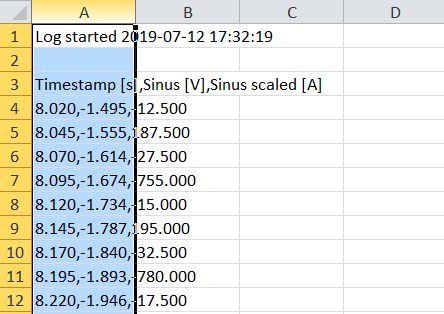
In the converter menu, the “Extrapolate last” checkbox has a special function. When stretching columns to the longest “timeseries”, one can choose between

* Aligning the last known values at the end of each column (checkbox unticked)
  + this makes extrapolation unnecessary, only interpolation is used
  + time step linear interpolation between strobes is (slightly) distorted
* Aligning the last known values at the beginning of the last strobe record (checkbox ticked)
  + this makes extrapolation necessary for the last strobe record
  + time step linear interpolation between strobes is correct

Note that for datacells where lowerCaseOf(name of datacell) is “timestamp”, the “extrapolate last” option is selected by default in order to avoid distortion in the linear time scale. It is generally recommended to tick this option.

The output of the convert operation results in a comma-separated .csv file with line breaks, ready for import into spreadsheet software. In case of conversion error, *Lasso Client*’s console window shall be consulted.

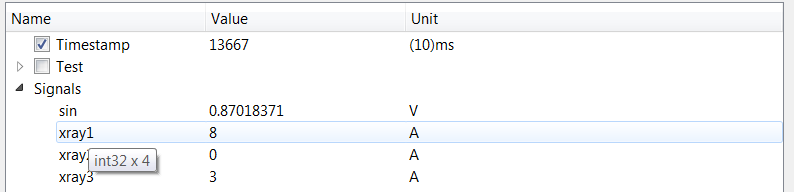
In order to convert the .csv file into a full-fledged spreadsheet, the spreadsheet software’s conversion assistant can be used. The .csv file holds comma-separated data in a single column. This data must be split into a new column at each comma. In MS Excel, select column A and use the import assistant by clicking “Data→Convert”.



## Interacting with a Lasso Host at Runtime

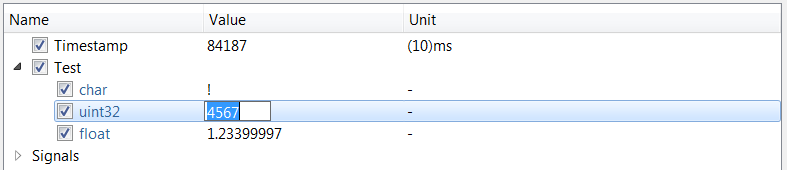
### Interaction via the Tree View

The tree view permits to determine the type and size of each datacell by hovering with the mouse pointer over the name of the datacell. Below is an example for a datacell (“xray1”) that holds an array of 4x int32. Note that the tree view only displays the first value of an array.

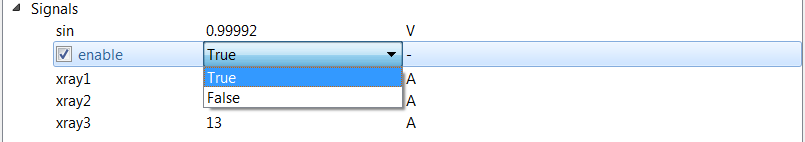


### Changing Datacell Values

The user can change values of datacells declared with the flag LASSO\_HOST\_WRITEABLE. These datacells appear with a blue name tag (otherwise the name tag is black). By double-clicking (left mouse button) in the values column of the tree view, the modification dialog opens. Type in the new value and press enter. The value is transferred to *Lasso host* immediately.



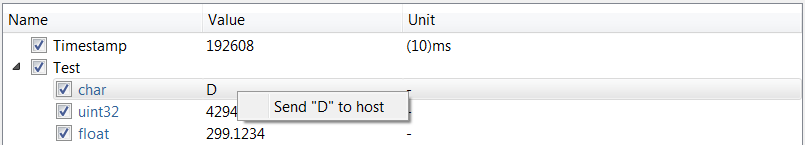
For Boolean values, a dropdown menu opens. The choice is then restricted to “True” and “False”.



For strings (arrays of char), any character that exceeds the length of the array specified on the host side will be truncated.

Changing datacell values is possible while strobing or not strobing. A type check is performed before sending values to the host. A message will appear in *Lasso Client*’s console window if the entered value is incompatible with the target datacell’s type.

It is also possible to re-send the current datacell value, which can be particularly useful for transmitting command characters multiple times. By clicking (right mouse button) in the values column of the tree view, a pop-up opens, and when clicked on, will transmit the current value of this datacell to *Lasso Host*.



## Limitations

Lasso requires sufficient bandwidth on the serial link. Strobing will definitely fail when the bandwidth limit is reached (refer to §4.2).

However, strobing can also fail for other reasons, for example when the latency on the client side is too high. In case of high strobe frequencies (e.g. 100 Hz, one strobe frame every 10 ms), even a modern PC running *Lasso Client* under MS Windows might already struggle. The strobe queue will quickly overrun, strobe frames are then dropped and lost.

# Advanced Features

## Host Side

### Encodings

RN

COBS

ESCS

### Packings

ASCII

msgpack

### Static and Dynamic Strobing

### The Controls (CTRLS) pipe

The controls pipe permits sending an array of Bytes from a *Lasso Client* to a *Lasso Host* without expecting an acknowledgement (response) from the host. The controls pipe is uni-directional and each controls frame has a maximum (Byte) length of LASSO\_HOST\_COMMAND\_BUFFER\_SIZE (see *Lasso* configuration in §6). The *Lasso Client* GUI does not make use of the controls pipe directly, but user can access it through plugins.

On receiving a controls frame, *Lasso Host* will pass the frame immediately to a user-specified callback function instead of interpreting the controls itself. This means that handling the content of the controls pipe is completely up to the user. For example, it is possible to implement a custom command interface.

In order to hook up the callback function, the user application on *Lasso Host* must make a successful call to lasso\_hostRegisterCTRLS().

In order to use the controls pipe on *Lasso Client*, a user’s plugin must first define the plugin for use with the controls pipe by setting

CONTROLS = True

and register the PyQt signal that gives access to the controls pipe

if SIG\_UPDATE\_CONTROLS:

self.SIG\_UPDATE\_CONTROLS = SIG\_UPDATE\_CONTROLS

Assuming that the controls state is represented by the Python list self.state[…], a frame is then placed in the controls pipe by calling

self.SIG\_UPDATE\_CONTROLS.emit(self.state)

It is best to refer to one of the plugin examples in *Lasso*’s “/plugins” subfolder in order to start using the controls pipe.

### Notifications

Todo

### Synchronization to User Application

## Client Side

### Plugins

Users can freely create plugins for *Lasso Client* by using Python and PyQt5 scripting. In fact, plugins may import all Python libraries used by the core of *Lasso Client*. The available libraries are packaged by the *Lasso Client* installer in the “/bin” subfolder of the installation directory and include pyqtgraph, numpy, scipy, moderngl and pyrr.

Some plugin examples are distributed with the *Lasso Client* installer and can be found in the “/plugins” subfolder of the installation directory.

Plugins declare which state information they need to access (read-only). They can also interact with *Lasso Hosts* through the controls pipe (refer to §4.5.2).

All plugins available in the “/plugins” subfolder are loaded automatically into *Lasso Client* on startup. However, plugins only become available if loaded successfully (otherwise, an error message will appear in the console of *Lasso Client*’s main window).

Todo

### Remote Control

Once a connection between *Lasso Client* and *Lasso Host* is established, it is possible for 3rd party software to interact with the *Lasso Host* by using the delegation feature (relay mode) of *Lasso Client*.

Todo

### Notifications

Todo

# Configuration Reference

# Outlook

*Lasso* still evolves and upgrades are planned in future releases …

Upgrades:

* Possibility to specify format codes in the unit string of DataCells (e.g. for displaying numeric values as hex, not decimal)
* Full strobe queue indicator in *Lasso Client*
* Persistent trace in scope window
* Inspection of “structs” (implicitely possible)