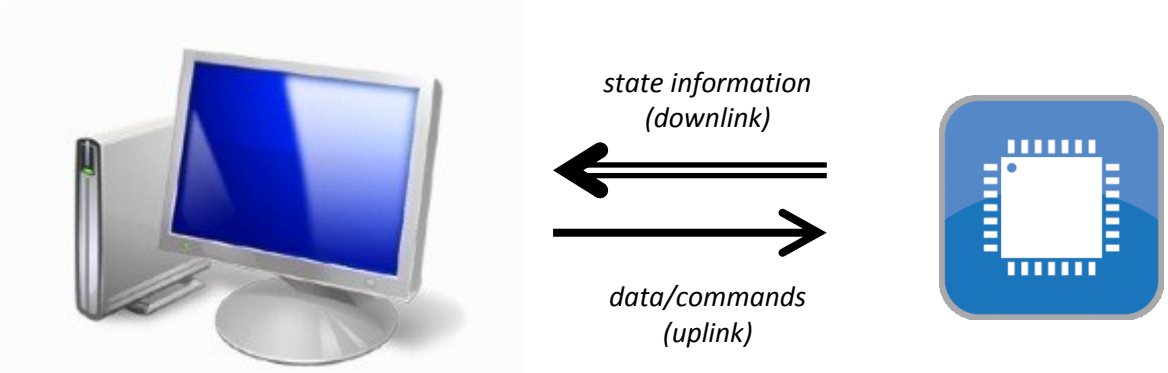
LASSO

A Lightweight ASynchronous State Observer for Embedded Systems

1 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.



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.

1.1 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.

1.2 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	• Installable on any PC platform, installer

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
- 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

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

2 Distribution

Lasso host

Lasso Host is open source.

Download source files from http://srvrepos/svn/ELEC/Tools/Lasso host 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 or

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

3 Setup

3.1 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.

3.1.1 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.

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

3.1.2 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.

3.1.3 Other Code Requirements

3.1.3.1 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()*.

3.1.3.2 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. */
}
```

3.1.4 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:

3.1.4.1 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`.

3.1.4.2 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`.

3.1.4.3 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	ASCII character
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 … 2^{64}-1
LASSO_INT64*	2^{63} … 2^{63}-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`.

3.1.4.4 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`.

3.1.4.5 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`.

3.1.4.6 Lasso_hostSetBuffer()

Todo

3.1.4.7 Lasso_hostCountdown()

Todo

3.1.4.8 lasso_hostTickPeriod()

Todo

3.1.4.9 lasso_hostHandleCOM()

Signature:

```
void lasso_hostHandleCOM (void);
```

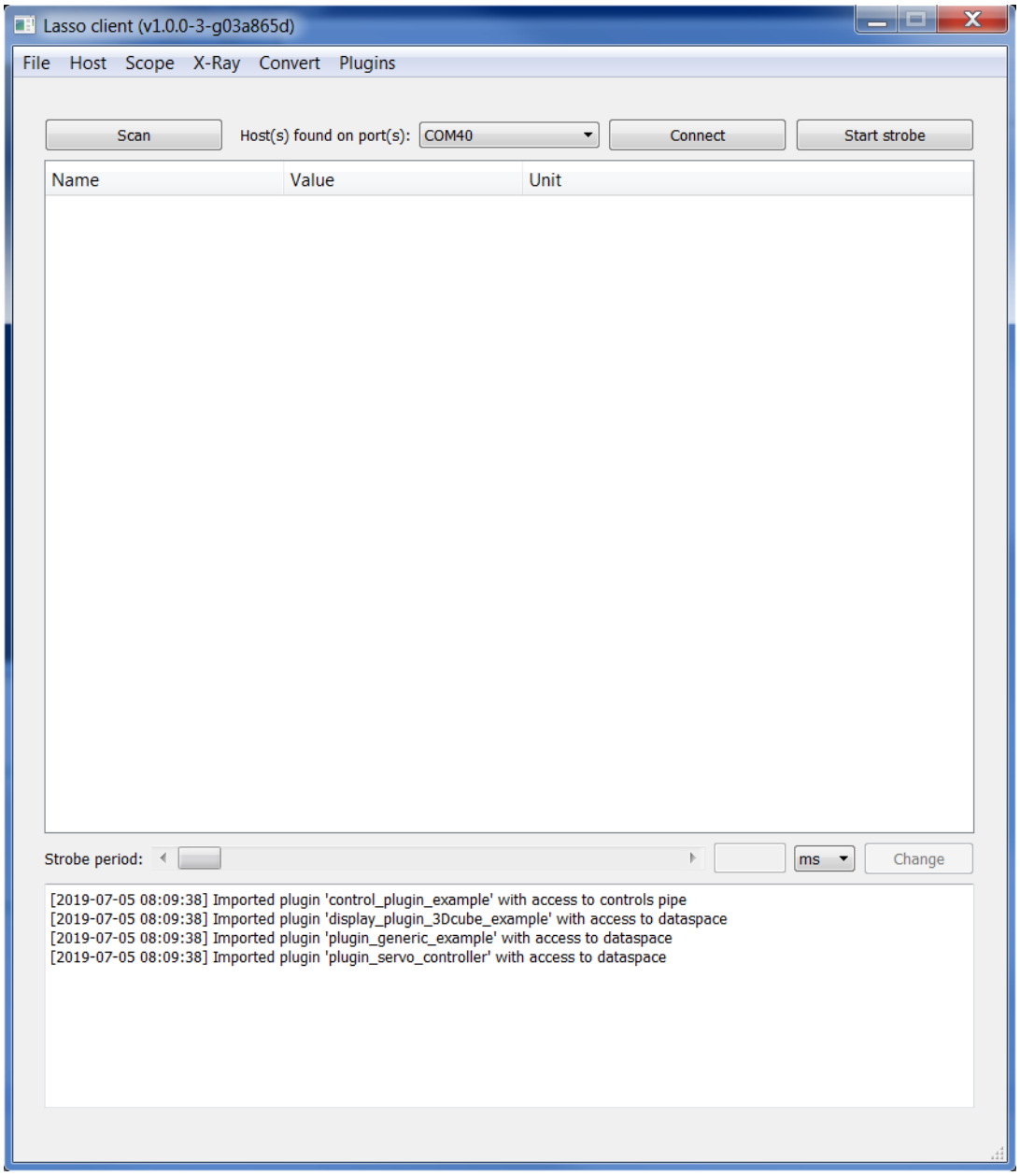
The central *Lasso Host* function for sending state information to the client across the serial downlink.

3.1.5 Callbacks

Todo

3.2 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:

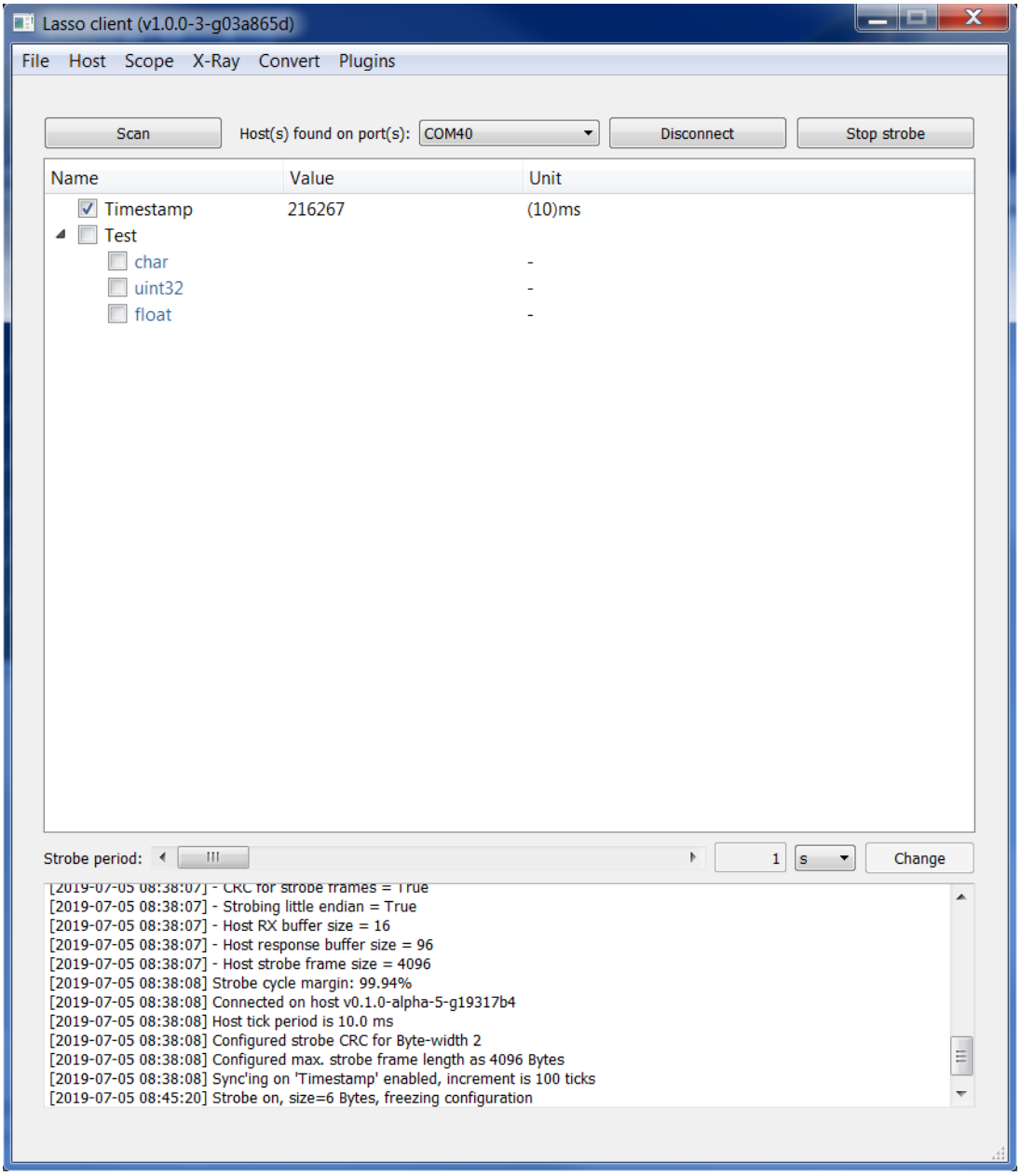


After hosts are found, it is possible to connect to them. Note that each *Lasso Client* only permits one host-connection at a time. However, after disconnecting, it is possible to select another host and connect to it. Also, it is possible to use multiple *Lasso Clients* at the same time.

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”).



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.

4 Using Lasso

4.1 Declaring State Information for Download

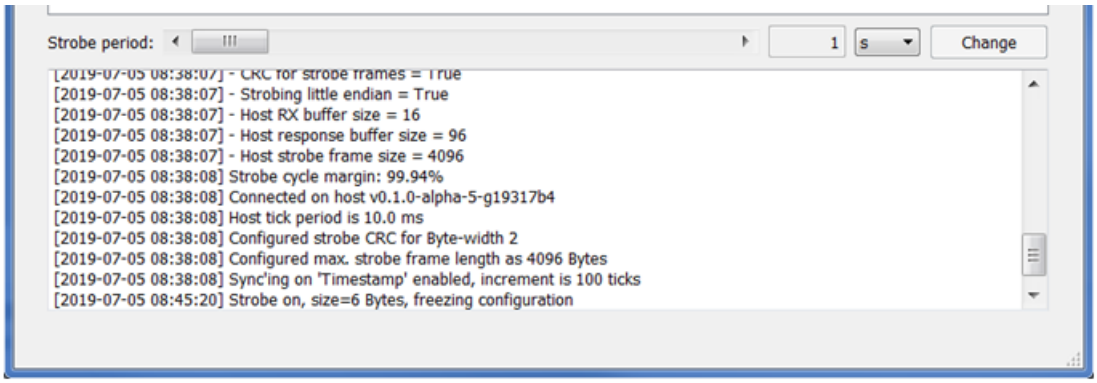
On *Lasso Host*, use the `lasso_hostRegisterDataCell()` function, refer to Section §3.1.4.3. This allows state information to become available for download.

On *Lasso Client*, all state information declared by the host is eligible for download. There is a difference between "being declared for download" and "being activated for download". State information declared for download may not be activated for download by default. In that case, while not strobing, such state information can be activated for download by clicking the appropriate checkbox in *Lasso Client*.

State information may be declared "permanent" by the host. 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.

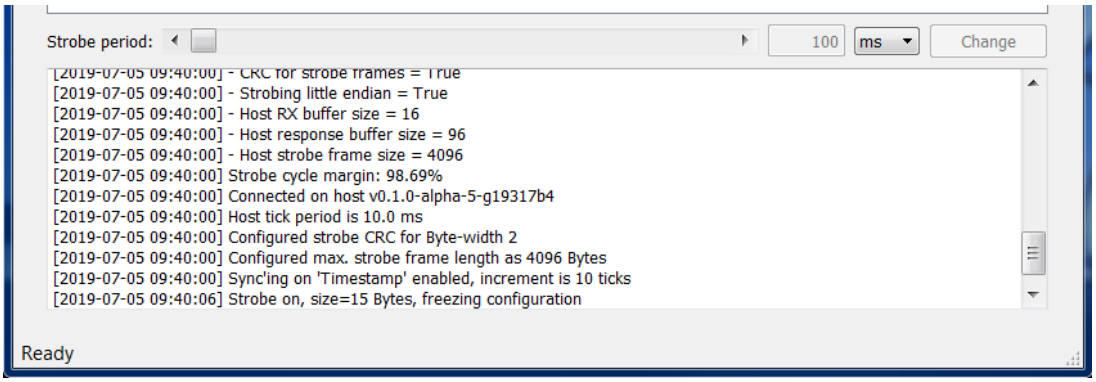
4.2 Inspecting Serial Link Bandwidth Margin

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.

4.3 Displaying State Information

4.3.1 Hierarchical View

4.3.2 Scope View (“Scope”)

4.3.3 Array Inspector View (“XRay”)

4.4 Logging State Information

4.5 Interacting with a Lasso Host at Runtime

4.5.1 Interaction via the Tree View

4.5.2 The Controls (CTRLS) pipe

5 Advanced Features

5.1 Host Side

5.1.1 Encodings

5.1.2 Static and Dynamic Strobing

5.2 Client Side

5.2.1 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

5.2.2 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.

6 Outlook

Lasso still evolves and upgrades are planned in future releases …

Upgrades:

- Configure baudrate on *Lasso Client* side
- Possibility to specify format codes in the unit string of DataCells (e.g. for displaying numeric values as hex, not decimal)