www.WesAudio.com

GCon Framework Manual Rev 1

EN

GCON FRAMEWORK

Manual Rev 1

Table of Contents

1. Preface

This document covers usage of software development frameworks which are used to integrate with GCon controller (e.g. _TITAN). The main focus in this document is to provide mandatory information to create remotely recallable hardware unit, keeping all terms and knowledge provided in this document easy to understand. To make it even simpler we decided to divide this document into simple start up steps ("Software – Quick Start Guide") and advanced section which would require some more work and programing knowledge ("Software – Advanced").

Please note that this documentation as well as software frameworks can't be used to create any own implementation of GCon controller (e.g. _TITAN), which is closed system owned by WesAudio. All information in this document as well as source code in form of attached software frameworks can be used to create own implementations (both hardware and software):

- **GCon client (e.g. Standalone application which communicate with GCon controller, DAW plugin like AAX/VST3/VST2/AU…).**
- **GCon hardware module (e.g. Hardware unit which connects with GCon controller).**

More information about proper definitions can be found in this documentation.

This entire document is description of technology owned by WesAudio. To integrate with WesAudio GCon system, it is required to sign GCon framework NDA and get a source code. If you are interested please send an inquiry to info@wesuaiod.com .

2. Abbreviations and terms

WesAudio – Pro Audio company which created GCon and this documentation.

GCon – high speed communication protocol which allows full management and recall of analog devices. Please note that this is just management protocol, audio signal transfer is not in scope of its capabilities.

I.A.C. – Internal Audio Connector, special _TITAN connector which is integrated with internal audio routing, and allows to connect audio signal directly without any additional wires (please check "I.A.C – Internal Audio Connector" chapter).

NG500 – Next generation 500 series.

NG500 connector – special connector which extends standardized 500 series connector with additional pins.

UDP - The User Datagram Protocol.

GCon controller – Main object in GCon topology, mediator element with basic management capabilities. GCon controller is closed implementation owned by WesAudio company, it is available in two forms:

- Commercial product available worldwide (e.g. _TITAN).
- PCB board which can be ordered from WesAudio to integrate with own hardware.

GCon module – Any object which works under controller.

GCon client – Any Application which uses GCon protocol to communicate with GCon controller – e.g. standalone application, DAW plugins (AAX/VST3/VST2/AU…).

GCon BlackBox plugin – Generic plugin which works with all GCon compatible devices allowing total recall functionality.

GConManager – Standalone application created to manage configuration of GCon devices.

Custom/Vendor specific plugin – Dedicated plugin which will work with specific hardware type.

Persistent memory – Any type of memory which can keep saved data during power off mode – e.g. EEPROM, FLASH.

3. Overview

This chapter describes basic concepts behind GCon protocol and its frameworks.

3.1. What is provided?

The main question is what is provided along with GCon documentation and frameworks, and what still has to be done to create own recallable units. Please find below short summary:

What is provided:

- GConPeriphery – bunch of drivers which covers following aspects:
  - Communication with GCon controller(e.g. _TITAN) – communication driver based on SPI, data queues, utilities.
  - GCon protocol code/decode implementation – easy to integrate, callback based interface to implement all plugin related functionalities.
- GConPeriphery example – Atmel Studio 7.0 based project which literally implements all above concepts and could be a great start up point to create own firmware implementation.
- GConBlackBox plugin – BlackBox plugin is literally plugin without any controls. Its main purpose is to connect to all GCon compatible devices and save all the runtime settings (e.g. parameters states). Automation is not possible at the moment, however most demanded market feature – total recall will work out of the box. Plugin is delivered in following formats:
  - AAX/AAX DSP
  - AU
  - VST2/VST3
- GPE – GCon protocol environment – high level C++ based API which can be used to create any application which will remotely manage hardware units. This framework can be used to create DAW plugins, or standalone applications.

What has to be done:

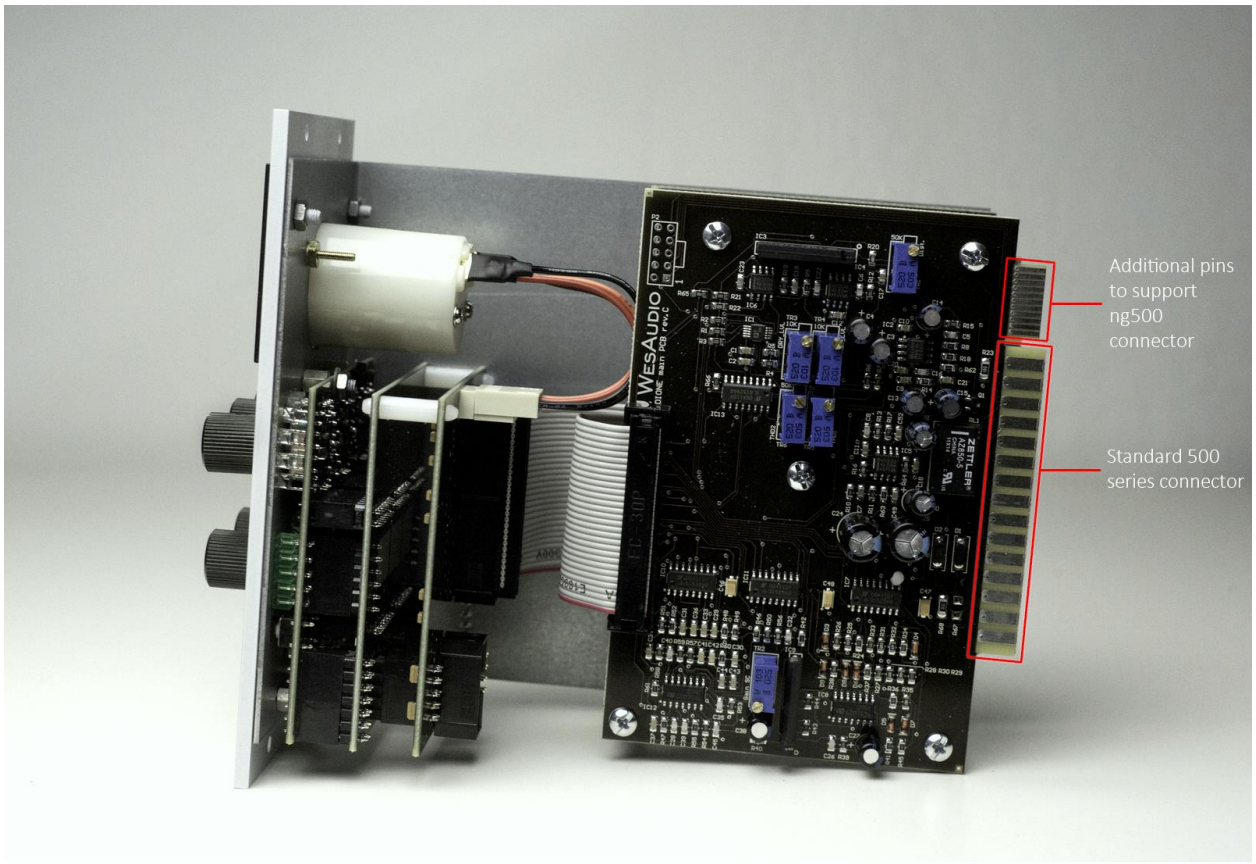
- Front panel management – buttons, knobs, encoders, LEDs implementation, etc... The reason for this approach is that in general all those kind of solutions are device specific, and it is hard to create a “C” framework which will actually work in generic way. There are so many possibilities, that at the end, such framework would be very complex.
- Analog circuit management – management of analog components responsible for audio processing (e.g. compressor ratio or release circuits, input level, etc.).
- OPTIONAL: dedicated plugin – GConBlackBox covers most of the functionalities which are in fact demanded by audio industry. That is why creation of dedicated plugin which will control device remotely can be either postponed for a later time, or abandoned entirely.

3.2. GCon

GCon is high speed protocol independent from physical medium, it defines own address space and proper logic for easy and secure communication. Currently it is implemented by _TITAN which supports both USB 2.0 and Ethernet UDP based connection**. To simplify whole area, this document will focus on integration with GCon rather then GCon itself.**

3.2.1. NG500

Ng500 series stand for "Next generation 500 series, and it implements special connector which allows to communication inside _TITAN chassis.



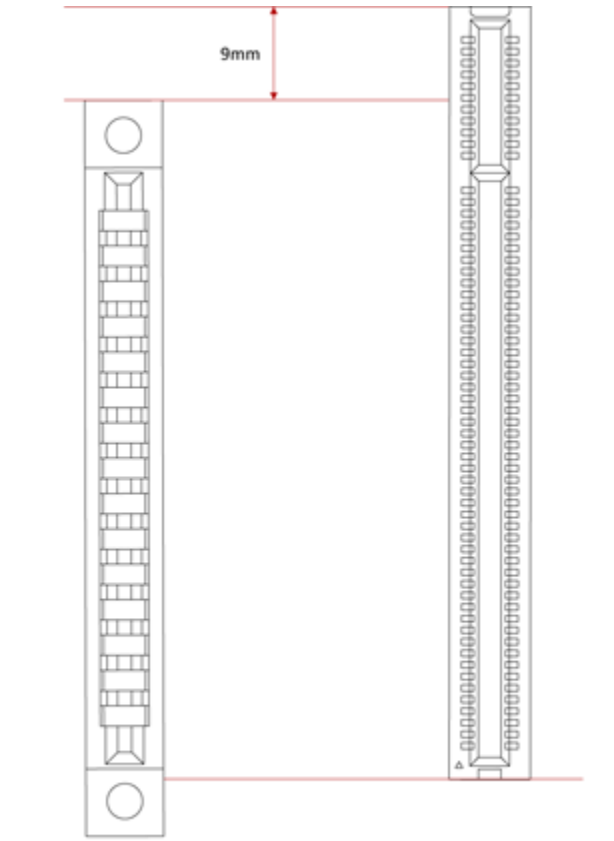
Those additional pins allows internal connection through SPI protocol, which is handled by special driver GConPeriphery described in next chapters. Placement of this connector was designed to keep best possible compatibility with standard 500 series frames. It is confirmed that all 500 series connectors used on the market are 100% compatible with this extension, however there is one exception:

- Some manufacturers apply enormous screws to the plug itself, what prevents the device to fit in. Based on official research more then 90% of available 500 series racks will work just fine. Currently known exceptions:
    - Rupert Neve Designs 500 series Rack (confirmed, removing screw will fix the issue)

- Aphex 500 series rack (not confirmed)

If this is the case, removing upper screw in the 500 series connector will fix the issue.

_TITAN socket which supports ng500 connector is based on PCI standardized socket, and was chosen to be fully compatible with 500 series standard socket. Nevertheless ng500 series connector is higher than standard 500 series plug and some standard 500 series modules won't fit into _TITAN. The reason for that is module's chassis design at the back, which can prevent the module to fit in correctly.

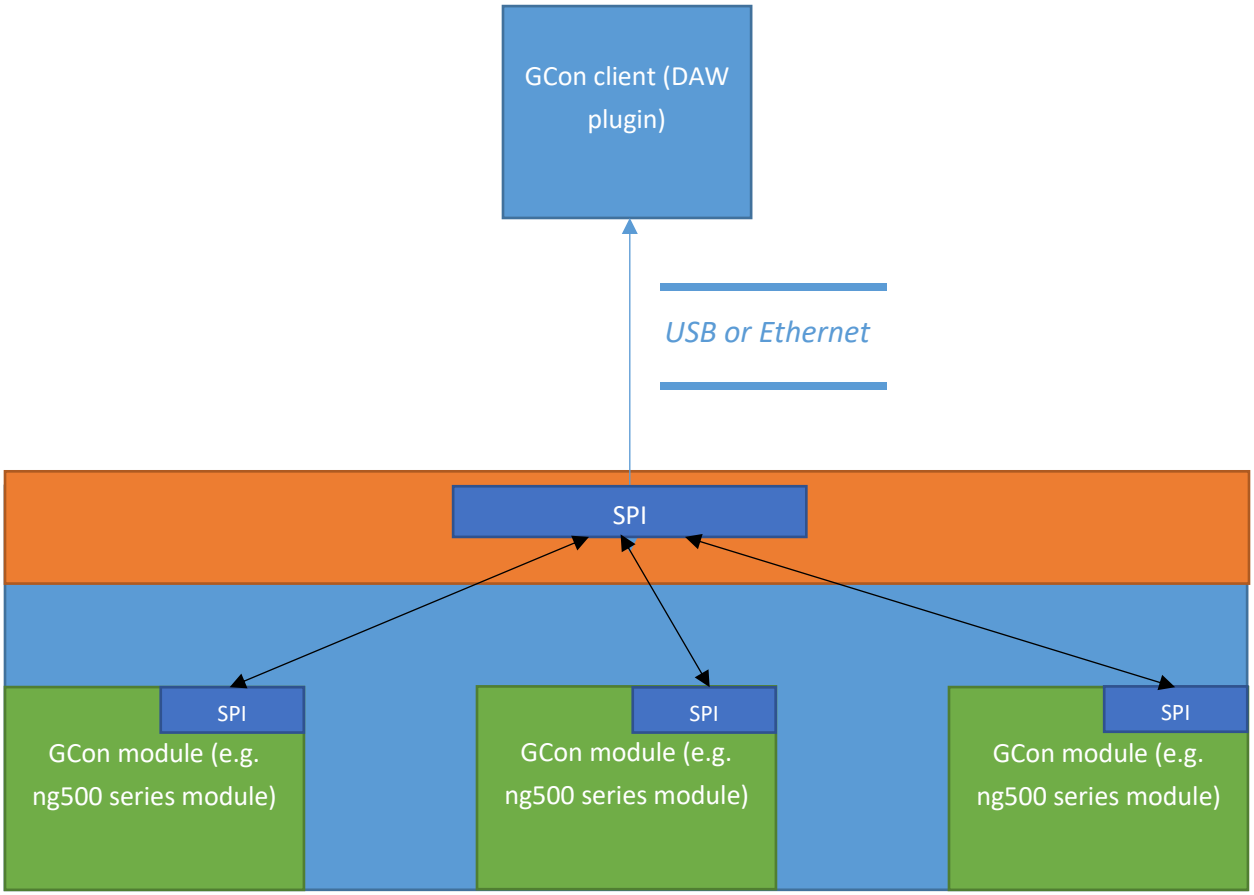


3.2.2. Rack units, consoles, and others.

Rack units or any other studio equipment can also be implemented as standalone version of GCon device, by encapsulating _TITAN PCB board, and creating separate MCU which will communicate with it through SPI. Such PCB board can be ordered from WesAudio. To do so, please send us a query to info@wesaudio.com.

3.3. Architecture

This chapter describes basic high level architecture from communication point of view. Understanding of this chapter is not essential to create own recallable modules, as most of it is already implemented by GCon frameworks and drivers. This short overview has been created to describe basic concepts selected during design phase. If you would like to start as soon as possible, go straight to “Quick Start Guide”.



GCon protocol defines three main roles in protocol translation:

- GCon client,
- GCon controller,
- GCon module.

Controller is master of modules, and it can communicate with northbound objects (GCon clients) using GCon plain protocol through USB or Ethernet (UDP) protocols on physical level. On southbound side, communication goes through SPI. It severs as mediation & management layer between GCon client and GCon module, allowing those to communicate with each other, and provides many special functionalities:

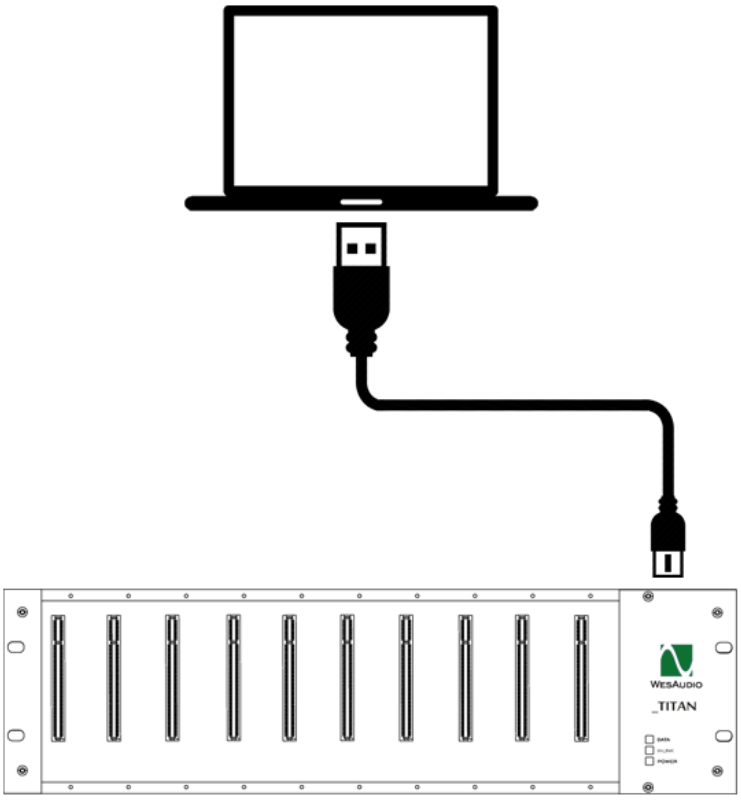
- Keep track of currently connected objects,

- Deliver to GCon client information about any topology changes (disconnections, connections),
- Mediate messages between modules and hosts,
- Provide information about currently connected modules,
- Keep track of currently paired units to prevent multiple GCon clients to access one GCon module.

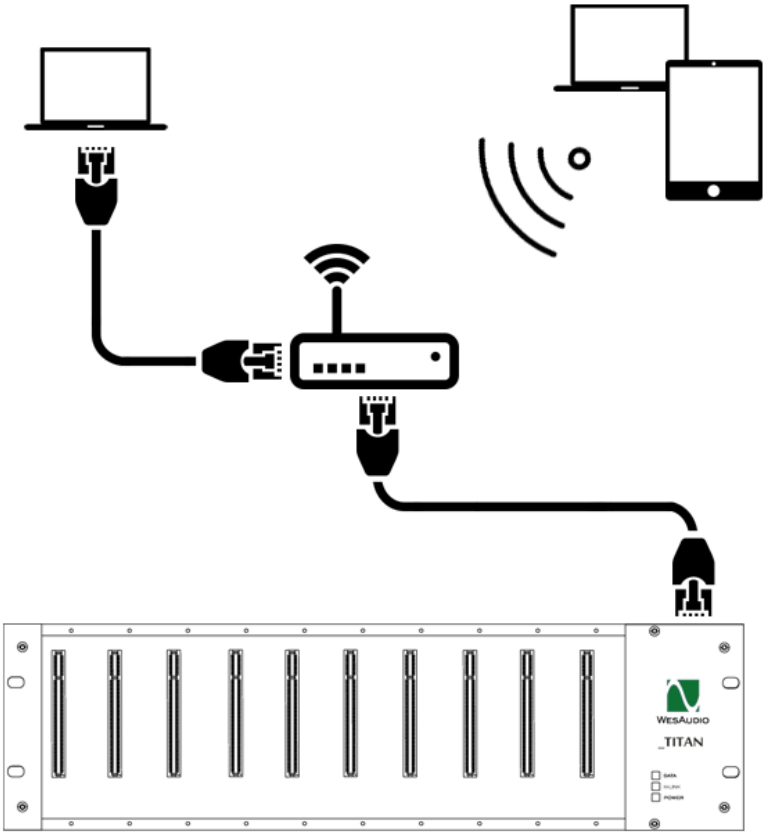
3.4. Setups

Below chapter will show possible setups in studio environment:

3.4.1. Direct USB connection.



3.4.2. Ethernet connection.



4. Hardware

This chapter describes basic schematics of hardware and mechanical components.

4.1. ng500 connector.

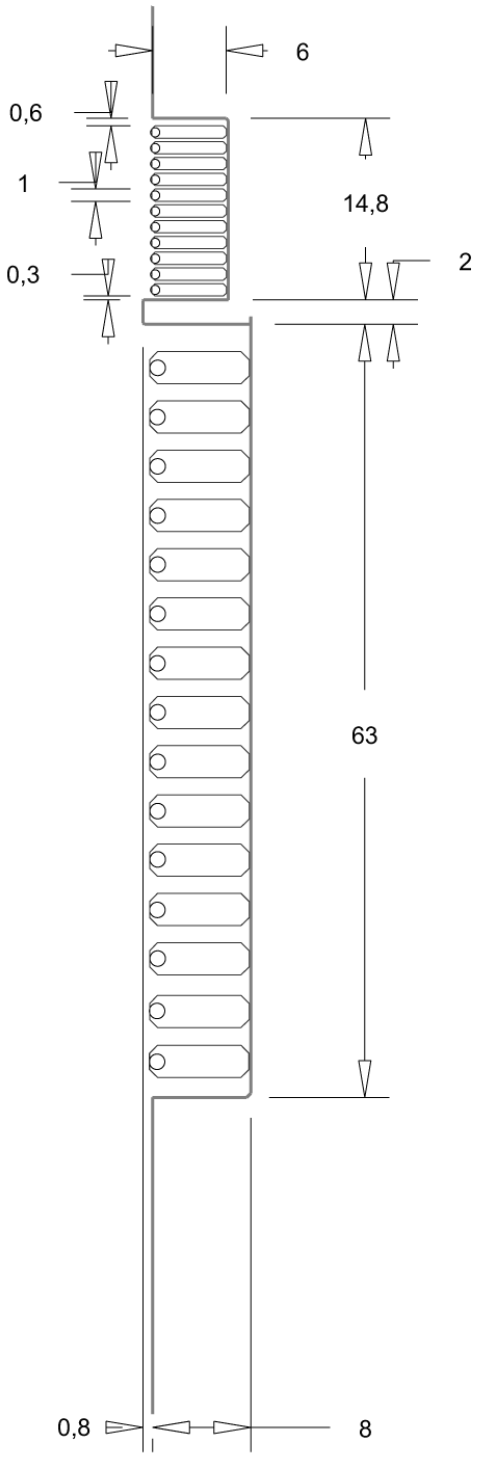
NG500 (Next Generation 500 series) connector implements additional pins to allow digital communication with GCon controller. Standard 500 series connector part is not modified so pin-wise it is 100% compatible with _TITAN pin layout.

4.1.1. PINs layout.

ng500 connector specific part:
D1 – SCLK (SPI clock)
D2 - MISO (Master Input Slave Output)
D3 – MOSI (Master Output Slave Input)
D4 – !SS (Slave Select)
D5 – SCL (I2C Serial Clock)
D6 – SDA (I2C Serial Data)
D7 – A0 (Slot Address LSB)
D8 – A1 (Slot Address)
D9 – A2 (Slot Address)
D10 – A3 (Slot Address MSB)
D11 – DGND (Digital GND)

Standard 500 series connector:
1- Chassis Ground
2- Output +
3- Output +
4- Output –
5- Common
6- Stereo Link
7- Input –
8- Input –
9- Input +
10- Input +
11- not used
12- +16 VDC
13- Power Supply Common
14- -16 VDC
15- +48 VDC

4.1.2. Dimensions.



4.2. MCU.

In general integration with GCon controller can be done on many different microcontrollers from many different vendors. The main requirement is to have enough processing power to allow SPI communication, which is very dependent on SPI slave timing. Even small delay in the response may cause communication to fail.

In general hardware requirements can be summarized like this:

- SPI.
- Enough processing power to handle SPI communication and other background tasks.
- Persistent memory implementation – e.g. EEPROM.

At current state we can recommend Atmel MCU family – XMEGA, as we can confirm that it integrates properly and without any issues with _TITAN.

5. Software - Quick Start Guide.

This chapter describes mandatory start up activities, namely:

- Basic concept,
- Firmware implementation – example and description,
- Integration with GConBlackBox – basic recall plugin implementation.

Custom Plugin implementation framework (GPE – GCon plugin environment) is described in the next chapter.

5.1. Testing environment.

It is quite obvious that some real hardware is needed to properly use WesAudio drivers and software frameworks. There are two ways how to proceed:

- Prepare own evaluation board based on this documentation, and connect it directly to _TITAN.
- Evaluation board – most elegant way as you will have clear understanding about connections between MCU and _TITAN, and will allow you to prepare your own testing additions. This board can be ordered from us directly. If you would like to get more details, please send a query to info@wesaudio.com.

To acquire own _TITAN unit rack for testing purposes please send a query to info@wesaudio.com.

5.2. Example.

Please note that each step described in next chapter, is in fact taken from an example which can be found in GConPeriphery/examples/GConModule. It is Atmel Studio 7 compatible project which is configured to run on Atmel ATXmega256A3U. It contains major functionalities and some dummy implementation parts to simulate front panel activities (like pressing button which triggers parameter change). This dummy implementation is compatible with GConBlackBox plugin, so it is enough to burn this program into device memory, and start GConBlackBox plugin, this should trigger parameter updates which can be noticed in the plugin (plugin will blink). Please note that in this example you will find some parts which are device dependent, like:

- Simple logger implementation through USART.
- SPI configuration.
- Interrupt handler – Atomic blocks.

Nevertheless most of the implementation is generic, and can be used as-is on different devices.

5.3. GConBlackBox plugin

GConBlackBox is generic plugin, which will work with any GCon device if proper handler object is implemented in the firmware. Please find below summary:

- GConBlackBox query all current parameters configuration from the device upon connection.

- GConBlackBox query all current parameters state (including separate configurations e.g. A/B) form the device upon connection.
- Each parameter state is sent to the plugin, and there it is saved into the plugin settings, so each time session is opened, hardware unit will be recalled properly.
- Plugin doesn't have any controls.
- Dedicated plugin can still be used on top of it, and all parameters states can be synchronized.

As stated above, attached example is fully compatible with GConBlackBox.

In the future it is also possible to create automation features or simple remote management for GConBlackBox, however it is not currently planned to be implemented.

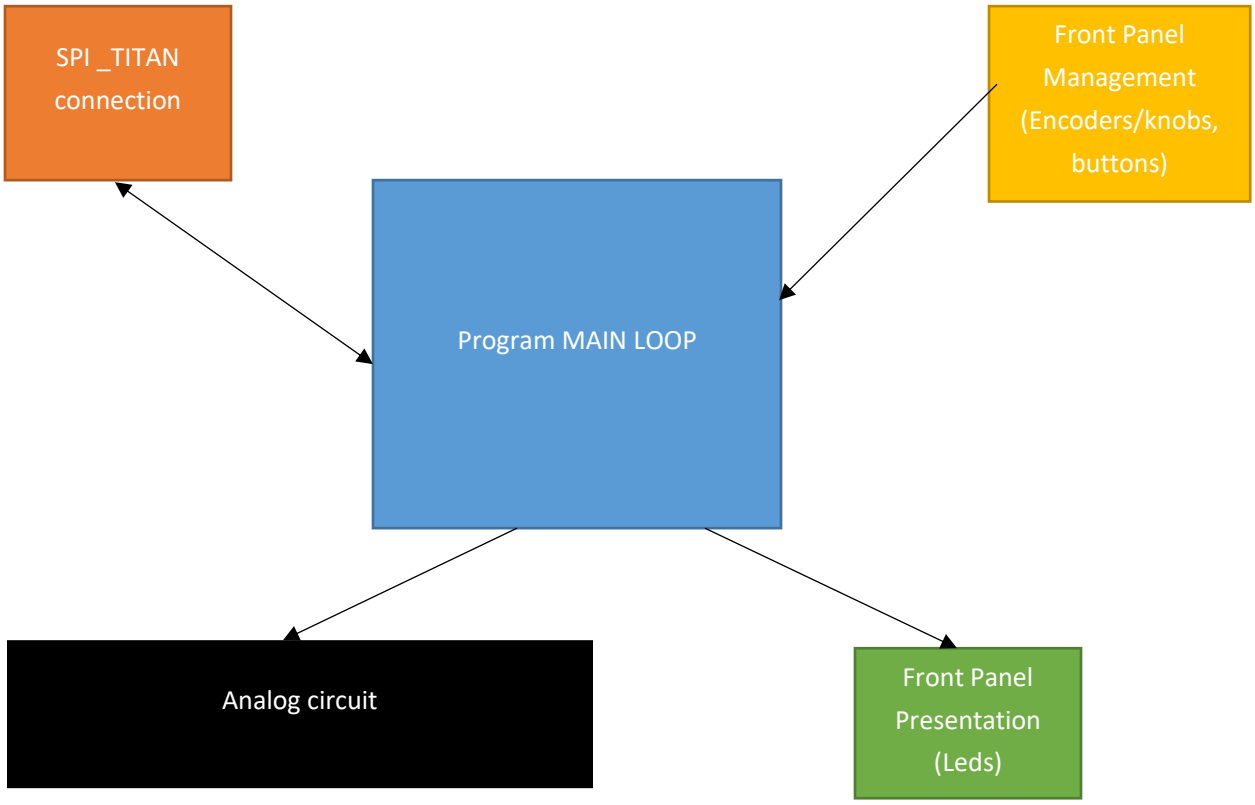
GConBlackBox plugin can be installed using WesAudio installer from www.wesuaiod.com/download . Currently this installer contains all WesAudio plugins, but in the near future there will be separate installer which will just install GCon runtime libraries along with GConBlackBox plugin.

5.4. Firmware – module implementation.

To create digitally controlled analog device, it is mandatory to implement firmware which will be responsible for controlling analog circuit based on remote parameters updates or front panel configuration changes. In this particular case we can simplify, that analog components can be modified from two independent sources:

- Hardware Front panel (In theory this is optional, however mandatory to work with GConBlackBox plugin)
- Via SPI connection.

Based on received data, each parameter should be reflected to analog circuit and front panel.



As you can see on above picture, there are two main input sources (described above), and 3 different output sources:

- Front panel presentation – e.g. Leds, LCD display, etc…
- Analog circuit - e.g. VCA, digital potentiometer, etc…
- SPI connection - In case of internal changes via Front Panel Plugin instance has to be notified through SPI connection.

Any front panel designs and mechanics are not in scope of this document, and this is clearly vendor specific implementation. The same goes for analog circuit management. However if you need further assistance, please send us a query on info@wesaudio.com. GConPeriphery firmware drivers cover whole SPI communication in easy to integrate manner, and attached examples give you simple start up implementation. There are several steps which have to be done, to properly configure firmware:

- Interrupt handler has to be configured – only valid if your implementation demand atomic blocks,
- GCon communication queues have to be initialized with proper size,
- SPI driver has to be enabled on MCU, and properly configured,
- GCon handler has to be implemented with proper actions,
- Proper persistent memory implementation have to be created (e.g. EEPROM).

5.4.1. Interrupt handler.

Interrupt handler defines how atomic blocks will be handled, or rather parts of code which have to be executed without any interference from interrupts. In most cases there are some dependent logical blocks executed in main loop, and some parts are executed in the interrupt routine. In GCon drivers, message queues are accessed from interrupts (SPI) and from main loop (receive/send), thus in most cases it is mandatory to define own interrupt "handler" (or rather atomic block handler) based on device type.

```
gcon_interrupt_handler_init(interrupt_handler_lock, interrupt_handler_unlock);
```

For more details about implementation, please refer to GConPeriphery example.

5.4.2. Data queues.

GConPeriphery driver implements data queues which are used to store messages. Each queue has to be initialized with particular size. It is up to MCU type how much memory can be used for communication purposes. Additionally, each queue can be defined as "interrupt safe" so each call on particular data type guarantees ATOMIC operation. It also depends on MCU architecture, however we strongly suggest to set each queue as "interrupt safe" to avoid some very tricky to investigate problems.

```
gcon_queue_set_recv_msg(10, true); //Setting receive queue with 10 elements.
gcon_queue_set_send_msg(10, true); //Setting send queue with 10 elements.
```

5.4.3. SPI.

To correctly integrate with GConPeriphery driver, there are few steps mandatory. First of all, SPI infrastructure has to be enabled on MCU, then proper GCon communication driver part has to be properly initialized. Because SPI infrastructure is MCU specific part, it is hard to write generic example, nevertheless please find below code based on Atmel framework:

```
ISR(SPIE_INT_vect)
{
	gcon_spi_slave_process();
}
```

```
gcon_spi_slave_init();

//Define callback to read and write to one byte SPI register.
gcon_spi_slave_configure(spi_slave_write, spi_slave_read);

//Atmel specific configuration of SPI
sysclk_enable_peripheral_clock(&SPI_SLAVE);
PORTE.INT1MASK = PIN1_bm;
PORTE.INTCTRL = PORT_INT0LVL_HI_gc;
SPI_SlaveInit(&spiSlaveD,
              &SPI_SLAVE,
              &SPI_SLAVE_PORT,
              false,
              SPI_SLAVE_MODE,
              SPI_INTLVL_HI_gc);
```

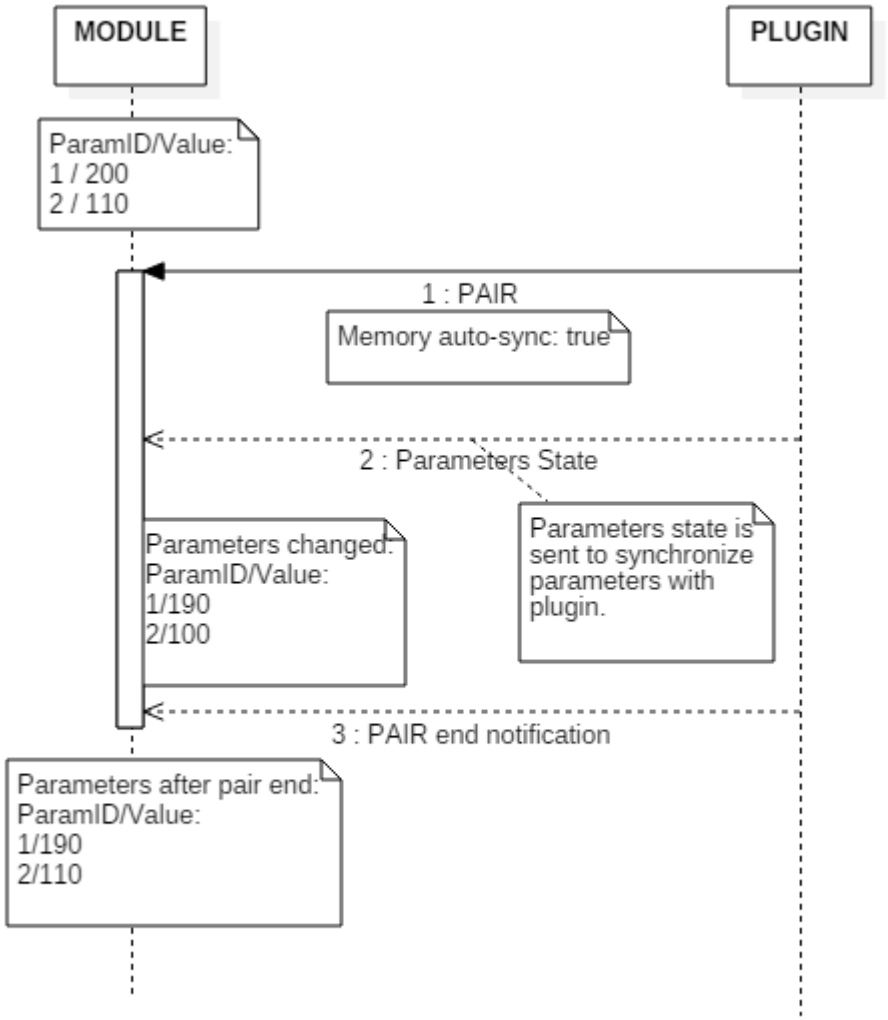
In real shortcut GCon specific configuration is very simple. It is enough to call main SPI callback function in interrupt routine, and configure write & read callbacks to allow access to SPI register. There are two essential functions which has to be called:

- `void gcon_spi_slave_init()` – main initialization function, sets all pointers to NULL, and handles all additional activities.
- `gcon_spi_slave_configure(GCON_SPI_WRITE_TYPE write_func, GCON_SPI_READ_TYPE read_func)` – main SPI callbacks, which are used to write to and read from SPI register. Those will be called each time SPI interrupt is triggered.

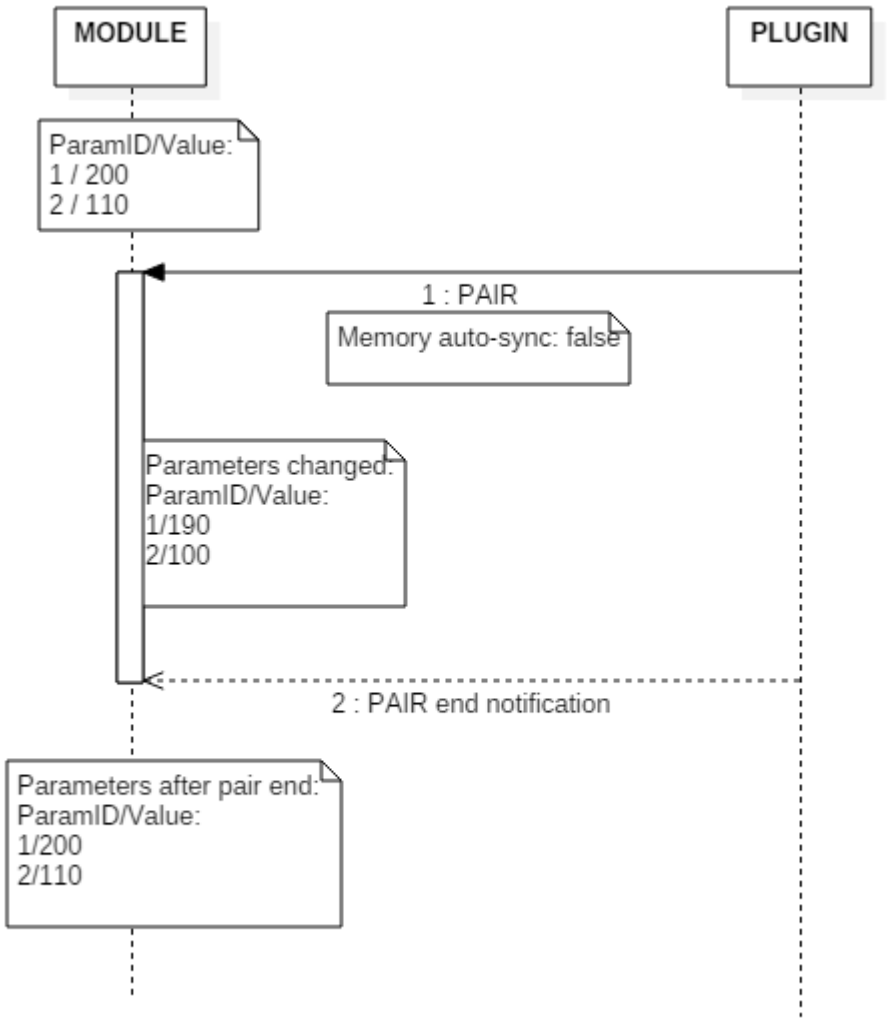
5.4.4. Memories and persistence.

In general GCon protocol allows to isolate internal hardware memory from plugin memory, or rather current state. It is up to GCon client (e.g. DAW plugin) to notify hardware which memory policy should be used (through pair scenario notification):

- Auto Memory Sync enabled (DEFAULT) – means, that GCon client and hardware unit should have memories synchronized. In practice it means, that after disconnection (pair end notification) hardware memory shall not change.



- Auto Memory Sync disabled – Hardware memory and GCon client memory shall work as separate persistence implementation, it means that after disconnection (pair end notification) hardware memory state shall go back to the state from before pair scenario.



Memory policy can be configured using GConManager standalone application, and each plugin should notify proper policy statement through pair scenario (auto_memory_sync flag in GCON_MSG_PAIR, please refer to “Pair scenario” chapter below).

Auto Memory Sync enabled NOTE:

Please note that if memories would have to be synchronized, writing to EEPROM memory (or FLASH) is not efficient, and best option is to implement periodic writes. It could also be considered to write data to persistent memory only on “pair end notification” which informs module that direct connection with the GCon client has ended. Additionally EEPROM memories have defined life cycle based on write operations, in that sense it is better to write parameter state into memory only if change is needed (e.g. parameter value is different in EEPROM memory).

5.4.5. Parameters representation.

It is assumed that each unit implements some sort of parameters representation. On GCon level it is implemented so:

- Each module defines certain number of parameters,
- Parameters can be set up in a setups, for fast “preset” changes.

There is one setup active at a time. If unit doesn’t implement different setups, it is assumed that setup number is equal “1”. In practice such representation is just a map of parameters which can be described with such example:

- SETUP ID 0 (e.g. representation of preset “A”):
    - Parameter ID 0/Value
    - Paramter ID 1/Value
    - …
    - Parameter ID X/Value
- SETUP ID 1 (e.g. representation of preset “B”):
    - Parameter ID 0/Value
    - Parameter ID 1/Value
    - …
    - Paramter ID X/Value
- SETUP ID Y:
    - …

It is up to module how many parameters setups should be implemented.

5.4.6. GCon senarios.

GConHandler is heart of GConPeriphery communication system. It defines all currently implemented GCon scenarios. Those will be described in few categories:

- Mandatory – scenarios which are mandatory to be implemented to allow module and controller communication,
- GConBlackBox specific – same functionalities are mandatory for GConBlackBox integration with the module,
- Optional – some functionalities can be ignored, but it is good to have those implemented.

Please note that each callback definition in gcon_decoder_handler structure accept as first argument handler instance. Handler instance contains “user_data” which can be useful to pass some information. Additionally it has been implemented that way for more complex scenarios, where different callbacks can be executed upon different triggers (e.g. callback calling another callback). Example implementation of gcon_decoder_handler can be found at:

- GConPeriphery\examples\GConModule\src\handler\message_handler.h

- GConPeriphery\examples\GConModule\src\handler\ message_handler.c

5.4.6.1. Mandatory scenarios.

Please note that all of below callbacks are implemented in GConPeriphery example, and you can find working solutions already there. Those callbacks ("receive" functionalities) are defined in gcon_decoder_handler object implementation which can be found in:

- GConPeriphery\src\lib\decode\gcon_message_decoder.h

All callbacks described in this chapter are part of this object definition. On the other hand, all "send" mechanisms are stored in:

- GConPeriphery\src\gcon_sender.h

5.4.6.1.1. Connection establishment scenario.

Each hardware unit have to inform GCon controller about its configuration. There can be two triggers to send this data:

- Module software is initialized (e.g. after restart),
- Following callback is executed on gcon_decoder_handler:

```
bool (*HANDLE_GET_INFO_REQUEST)(struct gcon_decoder_handler_t *self);
```

If any of above triggers is detected, module is supposed to send its information to controller (e.g. _TITAN) using gcon_sender module:

```
uint16_t gcon_sender_push_device_info(gcon_connection_element_v2 device_info);
```

Unit's description is passed using gcon_connection_element_v2 structure:

```
typedef struct  gcon_connection_element_v2_str {
	/*
		Interface version
	*/
	GCON_IF_VER if_ver;
	/*
		Vendor ID
	*/
	GCON_VENDOR_ID vendor_id;
	/*
		Hardware version is vendor specific ID which
		defines element hardware version. This parameter can be
		used to support revision changes of same hardware type.
	*/
	GCON_HW_VERSION hw_version;
	/*
		Hardware type - vendor specific hardware ID.
	*/
	GCON_HW_TYPE hw_type;
	/*
		Element ID - unique element ID.
	*/
	GCON_ELEMENT_ID id;
	/*
		Element Slot - it is used on controller level to inform
		GCon client about module's slot.
	*/
	GCON_SLOT_NUMBER slot;
	/*
		Hardware subtype - ID created to distinguish some specific
		module configuration like "Dual mono", "Stereo" etc. It is
		currently neither used nor documented/defined.
	*/
	GCON_INDEX hw_subtype; //Currently not used.
	/*
		Paired state - it is used on controller level to inform
		GCon client about module's accessibility.
	*/
	GCON_FLAG paired;
	/*
		Current firmware version:
	*/
	GCON_VERSION ver_major;
	GCON_VERSION ver_minor;
	GCON_VERSION ver_build;

	/*
		Software active flag:
		* true  - if software is active on module/unit,
		* false - if software isn't active, for example bootloader
			  is active for upgrade purpose.
	*/
	GCON_FLAG sw_active;
}gcon_connection_element_v2;
```

Above description is copy-pasted from source code, and most of the fields are documented there.

However there are few fields that require more explanation:

- **Interface version** – defines current module interface support. Currently supported version is: GCON_IF_VERSION_2, and unit should report this version to controller.
- **Vendor ID** – is defined on GCon level, if you don't have your ID defined already, please send a query to info@wesaudio.com.
- **Versioning** – Versioning will be described in detail in next revision of this document and it will be very dependent on chapter "GConManager ingtegration". At this point it is enough to just pass GCon framework version to avoid GConManager application warning.

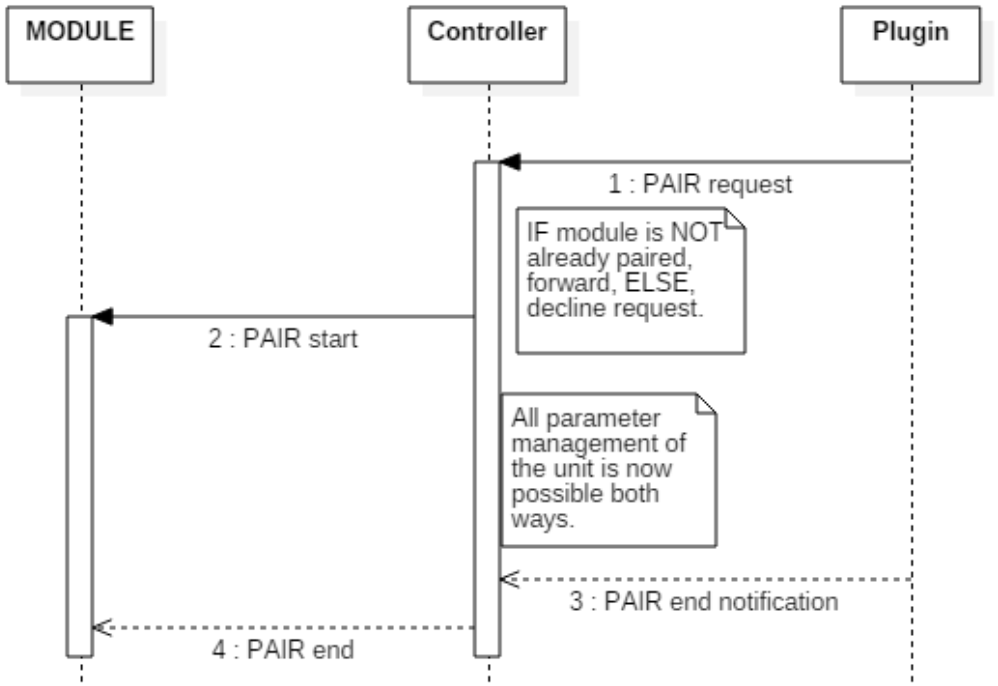
5.4.6.1.2. Pair scenario.

Pair scenario is one of the most important functionalities. If plugin would require exclusive rights upon certain hardware unit, it will send request to controller. Controller based on current state will:

- Decline the request if element is already paired with different GCon client ID,
- Will accept the pair request, and will forward pair message to the unit. In this case, following callback will be executed:

```
void (*HANDLE_PAIR)(struct gcon_decoder_handler_t *self,
                    gcon_pair_v2_info *pair_info);
```

If this callback is executed, it means that unit has to accept the request. In most simple case unit should just visualize this change on its front panel, but it is of course optional. _TITAN won't forward any parameter-like message types if unit is not paired with particular GCon client.



There are following data being forwarded to hardware unit during pair scenario:

```
typedef struct gcon_pair_v2_info_t {
	/*
		Priority of pair scenario, when two endpoints are
		currently paired, and third endpoint will send
		pair request with higher priority, currently
		active pair mode will be disabled, and
		particular endpoints informed. Higher priority
		always wins. This information should be not relevant to module.
	*/
	GCON_INDEX priority;

	/*
		CURRENTLY NOT IMPLEMENTED - when there are
		endpoints paired with certain priority, and
		new request will be sent with same priority
		but with this flag active, currently
		active pair mode will be disabled, and
		particular endpoints informed.
	*/
	GCON_STATUS forced;

	/*
		Defines if modules should synchronize
		internal memory to client state.
	*/
	GCON_STATUS memory_auto_sync;

	/*
		CURRNETLY NOT IMPLEMENTED - this value is ignored
		by controller, and it is kept here for future
		development.
	*/
	GCON_INDEX virtual_id;
} gcon_pair_v2_info;
```

First two fields:

- **GCON_INDEX priority;**
- **GCON_STATUS forced;**

  are relevant only for GCon controller (e.g. _TITAN) and are forwarded to modules just for information purpose. In real shortcut this data can be just ignored.

  Memory auto-sync flag:

- **GCON_STATUS memory_auto_sync;**

  sets proper memory policy (please check “Memories and persistence” chapter). **NOTE: If memory auto-sync flag is enabled, plugin have to synchronize its current parameters state by sending GCON_PARAMETERS_STATE message to the module right after pair request is confirmed – this**

message type contains all parameters information and is described in chapter below ("Update parameters state scenario").

PAIR scenario allows also some more complex possibilities, but those are relevant only to hardware/vendor specific GCon client (e.g. Plugin) implementation and will be described in chapter "Software – Advanced".

5.4.6.1.3. Parameters update – send.

If pair state is active, unit is supposed to notify any parameter updates (e.g. through front panel activities) to GCon client using following function (can be found in gcon_sender.h):

```
uint16_t gcon_sender_push_params(gcon_parameter * params, uint8_t param_size);
```

Parameters are passed as an array of gcon_parameter structures, and only modified parameters should be notified to GCon client. To get more information about helper functions used to send parameter updates, please refer to GConPeriphery example.

5.4.6.1.4. Parameters update – receive.

Parameters update are received through following: gcon_decoder_handler callback:

```
bool (*HANDLE_PARAM_BULK)(struct gcon_decoder_handler_t *self,
                          gcon_parameter *params,
                          unsigned int parameters_no);
```

This callback receives parameters from a GCon client. Such information should be passed to analog handlers to visualize and apply proper parameter changes. Parameters are passed through a callback implementation as an array of gcon_parameter where parameters_no states array size.

5.4.6.1.5. Change element ID scenario.

```
void (*HANDLE_SET_ELEMENT_ID)(struct gcon_decoder_handler_t *self,
                              GCON_ELEMENT_ID new_id);
```

Each time this callback is called, GConManager (GCon management standalone application) requested element ID change. Module should change its own ID by writing it to any persistent memory, and it should re-establish the connection either by restarting itself, or sending again module information (please check chapter "*Connection establishment scenario"*).

5.4.6.1.6. Update parameters state scenario.

If GConClient will send GCON_MSG_PARAMETERS_STATE message type to module, modules has to synchronize its current parameters state, as well as parameters snapshot in persistent memory. State of parameters is called through following callback:

```
void (*HANDLE_PARAMS_STATE)(struct gcon_decoder_handler_t *self,
                            gcon_parameters_state* param_state);
```

5.4.6.2. GConBlackBox specific scenarios.

All callbacks in this chapter are mandatory for hardware module to integrate properly with GConBlackBox plugin.

5.4.6.2.1. GConBlackBox requests.

Module (hardware unit) needs to provide certain data to GConBlackBox in order to integrate properly. Those are very simple synchronous scenarios. There are two types of data which are mandatory:

- gcon_parameters_state – defined in msg_types.h – this structure contains information about current parameters state including different setups (e.g. A/B).
- gcon_parameters_config – defined in msg_types.h – this structure contains information about parameter configuration. At this time only parameter type is important information, where all meter type parameters are ignored by the plugin. In the future however more information can be used, and for example automation features can be also enabled based on given configuration.

Configuration of parameters should be sent to controller right after following callback is executed:

```
void (*HANDLE_PARAMS_CONFIG_REQUEST)(struct gcon_decoder_handler_t *self);
```

To send this information to controller, following function from gcon_sender.h should be executed:

```
uint16_t gcon_sender_push_params_config(gcon_parameters_config *config);
```

Example code which sends such configuration can be found at: GConPeriphery\examples\GConModule\src\handler\message_handler.c, in function: message_handler_params_config.

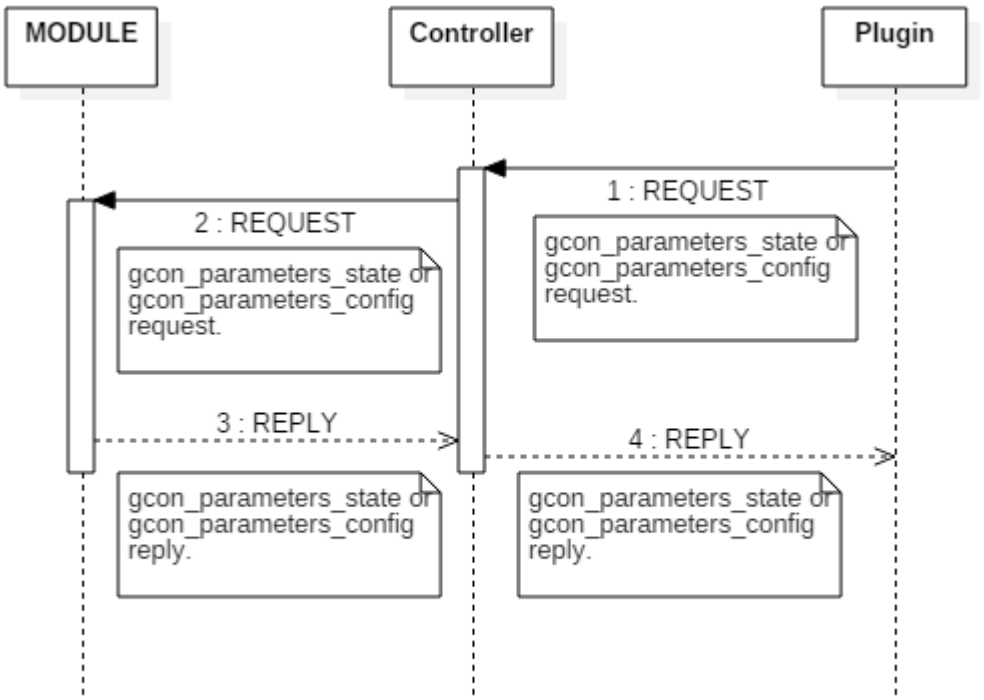
State of parameters should be sent to controller right after following callback is executed:

```
void (*HANDLE_PARAMS_STATE_REQUEST)(struct gcon_decoder_handler_t *self);
```

To send this information to controller, following function from gcon_sender.h should be executed:

```
uint16_t gcon_sender_push_params_state(gcon_parameters_state *state);
```

Example code which sends such configuration can be found at: GConPeriphery\examples\GConModule\src\handler\message_handler.c, in function: message_handler_params_state.



5.4.6.3. Optional scenarios.

All scenarios in this chapter are optional. Most of it are connected to functionalities which are not mandatory for module to work at all but are nice to have.

5.4.6.3.1. Multiple parameter setup.

This chapter is optional, because unit doesn't have to implement multiple setups of parameters.

If module is paired with GCon client, it means that all parameters are managed on GCon client side. During that time any front panel preset (setup) changes shall be only communicated to GCon client, as the plugin acts as a master holding current parameters state. It is up to plugin, to update parameters values and inform module which values have been changed.

STEP 1: To notify GCon client that setup has been changed, following function can be used:

```
uint16_t gcon_sender_push_setup_change(uint8_t setupId);
```

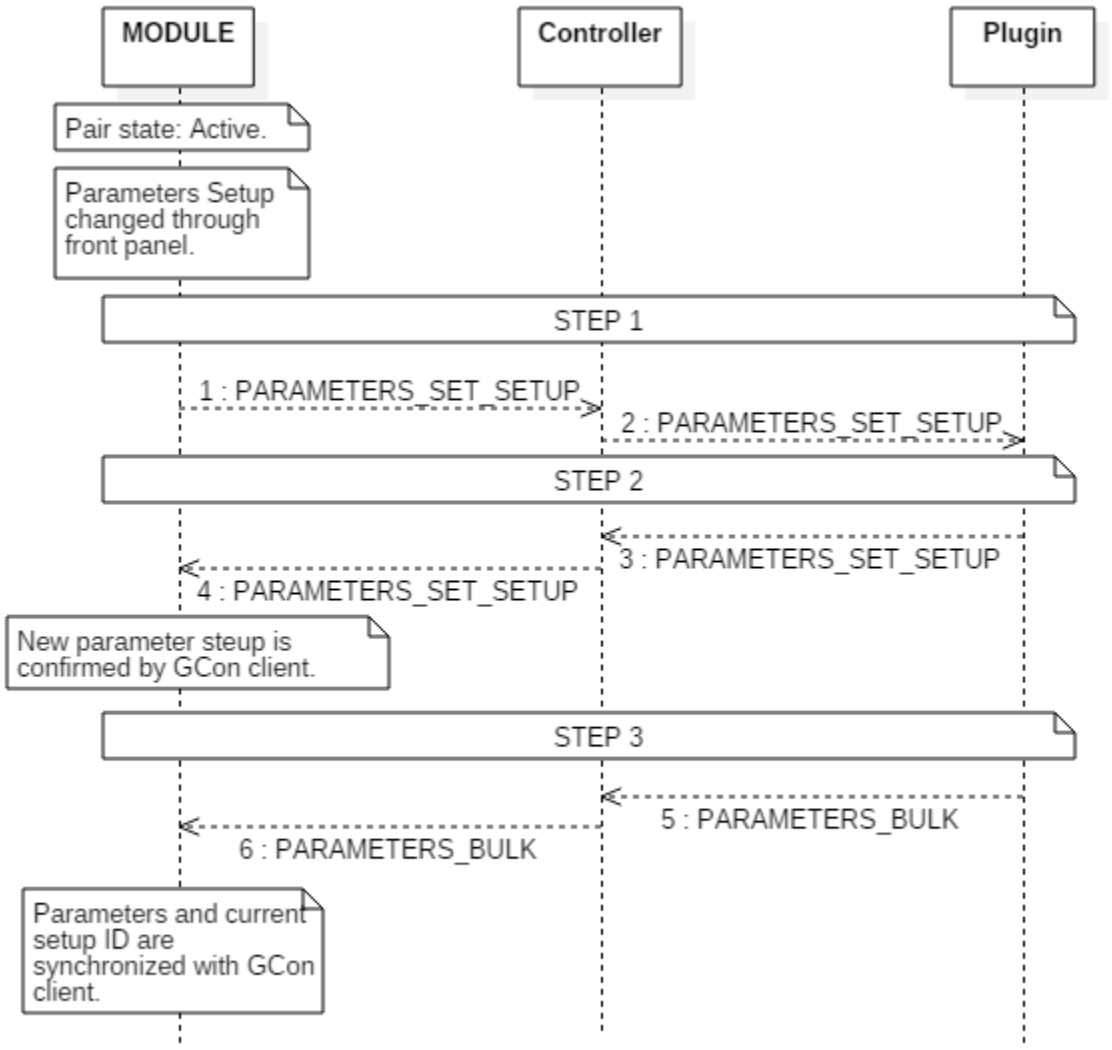
This function will inform GCon client that user requested setup change.

STEP 2: GConBlackBox plugin will send confirmation that setup has been changed, and following callback will be executed:

```
void (*HANDLE_PARAMS_SET_SETUP)(struct gcon_decoder_handler_t *self,
                                GCON_INDEX setup_id);
```

This callback informs module to visualize that parameters setup (e.g. A/B) has been changed. Please note that unit is not supposed to load parameters from persistence memory, it is just a notification that setup has been changed.

STEP 3: GConBlack box will update all parameters using common parameters update scenario (Please check *"Parameters update – receive"* chapter).



5.4.6.3.2. Parameter events.

```
bool (*HANDLE_PARAM_EVENT)(struct gcon_decoder_handler_t *self,
                           GCON_INDEX param_id, GCON_PARAM_EVENT_ID event_id);
```

This callback notifies about incoming parameter events. For example, unit can be notified when plugin parameter is in “touch” state. This is not used by GConBlackBox at all, because this plugin doesn’t have any controls, so this implementation is essential only for dedicated unit’s plugin.

5.4.6.3.3. Unit reboot.

```
bool (*HANDLE_RESET)(struct gcon_decoder_handler_t *self);
```

GConManager can send request to restart the element. After that callback is received, unit should restart itself.

5.4.6.3.4. Software upgrade start-up.

```
bool (*HANDLE_START_BOOT)(struct gcon_decoder_handler_t *self);
```

This callback is called if GConManager or any other application will request remote firmware upgrade. If this is supported, unit shall enter upgrade state. More information about firmware upgrade integration can be found in chapter “Firmware remote upgrade”.

5.4.6.3.5. External switches.

```
bool (*HANDLE_SET_SWITCH)(struct gcon_decoder_handler_t *self,
                          gcon_switch_type type, bool state);
```

This callback notifies module about internal GCon controller (e.g. _TITAN) switches modification. For example when link switch will be engaged on _TITAN frame, module will be notified about switch type and new switch state. For example, this functionality can be used to automatically detect stereo link changes by mono modules and engage linking if necessary.

5.4.6.3.6. Heartbeats.

```
void (*HANDLE_HEARTBEAT)(struct gcon_decoder_handler_t *self);
```

Each upper system heartbeat message is forwarded to modules. This callback can be used to check connection state and current activity to detect possible connection problems.

5.4.6.3.7. Vendor specific scenarios.

Each message type which won’t be successfully handled will be passed to following gcon_decoder_handler callback:

```
bool (*CUSTOM_DECODE)(char* buffer, unsigned int size);
```

I can be used by the vendor to implement more complex scenarios, or some other functionalities supported in GCon protocol. It is not needed for integration with GConBlackBox, but can be useful in Vendor specific GCon client implementation.

5.4.6.3.8. Data notification.

```
void (*DATA_NOTIF)(void);
```

This callback isn't really part of a scenario, but rather simple internal mechanism. This callback will be executed each time message is available on "receive" queue.

5.4.7. Main loop.

Main loop implementation should cover few aspects, but most important part, is that it has to execute receiver context of the GCon. In general it can be simplified to main loop implemented in GConPeriphery example:

```
void module_run(gcon_decoder_handler *handler)
{
	//Check if connection to controller is still active.
	check_connection_state();

	//Update connection info if needed...
	update_connection_info();

	//Check front panel activities:
	front_panel_dummy_run();

	//Receive messages
	gcon_receiver_process(handler);

	//Update analog circuit about any incoming changes.
	analog_run();

	//Update persistent memory.
	memory_manager_run();
}
```

Above steps can be executed in different order and in fact not all of those function calls have to be implemented as main loop explicit calls. In fact this is just example, and those steps can be implemented in many different ways. However to keep this example clean and easy to read, simplest approach has been chosen. Please find description of each step:

- **STEP 1:** `check_connection_state()` - **Connection state check:** simple heartbeat check to see if connection with GCon controller is still active.
- **STEP 2:** `update_connection_info()` - **Notify element information to GCon controller –** simple check if element information shall be sent to controller. As described in previous chapter this shall be done upon two triggers:
  - **restart of module,**
  - **request from controller**

It is described in detail in chapter ("Connection Establishment Scenario").

- **STEP 3:** `front_panel_dummy_run()` – **Process any front panel activities –** button activities, knob/encoder movement, switch position change – each action shall be processed, and proper parameter update have to be executed.
- **STEP 4:** `gcon_receiver_process(handler)` – **Main receiver context –** specific handler implementation is heart of GCon receive system.
- **STEP 5:** `analog_run()`– **Analog context –** this part should reflect any parameter changes into analog circuit.
- **STEP 6:** `memory_manager_run()` – **Persistent memory writer –** in scope of persistent memory update there are many different ways and implementation approaches, this particular call implements periodic checks if parameter shall be updated in the memory. Advantage of that approach is no significant performance impact, and low memory write ratio which shouldn't affect memory life cycle.

6. Software – advanced.

This chapter describes all advanced aspects of GCon frameworks and how to create own hardware specific GCon client (e.g. DAW plugin).

6.1. GConManager integration.

This chapter will be part of revision 2 of this document. It will cover:

- Custom configuration files deployment,
- Optional firmware upgrade integration.

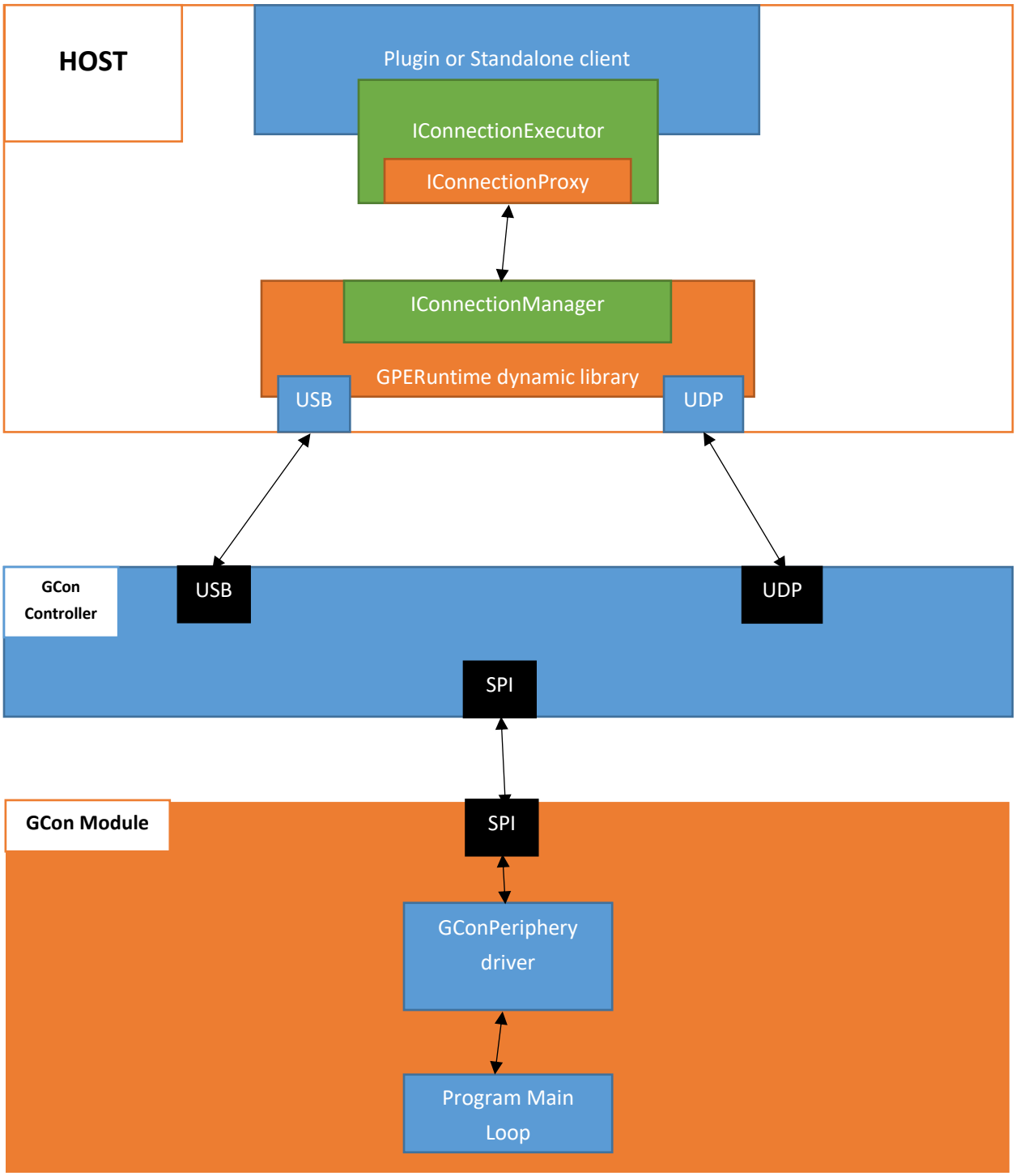
6.2. Software Frameworks.

There are two software frameworks available in GCon system:

- GPE – GCon Protocol Environment – High level C++ API used on HOST machine to communicate with GCon controller (e.g. _TITAN).
- GConPeriphery – GCon embedded driver & utilities used to communicate with controller – most of this framework is described in previous chapter ”Software – Quick Start Guide”, however some advanced parts which haven’t been described there require some additional information.

6.2.1. Software Architecture.

Please find below picture to get basic overview about software architecture:



6.3. GCon protocol/plugin environment (GPE – C++ API)

This chapter will be defined in revision 2 of this documentation, and will cover following aspects:

- How to create vendor/hardware specific plugin,
- How to integrate with GPE utilities.

6.4. GConPeriphery (Hardware drivers & utilities)

In general GConPeriphery wraps all GCon specific coding and decoding, and provides all mechanisms to communicate with GCon Controller. Most of the aspects are described in "Software – Quick Start Guide" chapter, however there are few more things which should be defined.

6.4.1. Configuration

GConPeriphery dependent project has to define some compiler switches:

- GCON_MALLOC_INTERRUPT_SAFE – this flag means that malloc/free native functions will be executed as ATOMIC blocks. It is not mandatory to use this particular configuration switch, however if this is not used, no allocation should take place in the interrupt routines.
- GCON_EMBEDDED – GCon drivers and protocol codecs have been designed to be as portable as possible. This is the reason why so many code is integrated into header files, which make it much easier to use. This switch is mandatory, and configures some optional configuration for embedded software.

6.4.2. Custom configuration

This chapter explains basic GConPeriphery configuration options.

- SPI driver can be configured with following preprocessor instructions:
    - GCON_CUSTOM_LISTENER - if current implementation of queues and buffer handling is not feasible for particular MCU, new implementation can be injected through this definition. To do so it is enough to create data_listener.h along with implementation, of course at the same time implementing same interface as gcon_data_listener.h.
    - GCON_LIGHTWEIGHT_DRIVER - removes some driver code which isn't mandatory. Mainly audio processing is removed and dynamic protocol switching (if this switch is engaged, SPI periphery shouldn't be disabled in runtime, because it could cause SPI line problems).

6.5. Firmware remote upgrade.

This chapter will be described in revision 2 of this document, and will cover some basics about integration with firmware upgrade accessible from GConManager.

Editor	Version	Date	Description

Michal Weglicki	**V1**	**25.06.2016**	**Document created.**
Michal Weglicki	**V2**	**07.01.2018**	**NDA requirements changed**