



DGILib

USER GUIDE

Table of Contents

| | |
|---|----|
| 1. Description..... | 3 |
| 2. API..... | 4 |
| 2.1. Discovery..... | 4 |
| 2.1.1. initialize_status_change_notification..... | 4 |
| 2.1.2. uninitialize_status_change_notification..... | 4 |
| 2.1.3. register_for_device_status_change_notifications..... | 4 |
| 2.1.4. unregister_for_device_status_change_notifications..... | 5 |
| 2.1.5. discover..... | 5 |
| 2.1.6. get_device_count..... | 5 |
| 2.1.7. get_device_name..... | 5 |
| 2.1.8. get_device_serial..... | 5 |
| 2.1.9. is_msd_mode..... | 6 |
| 2.1.10. set_mode..... | 6 |
| 2.2. Housekeeping..... | 6 |
| 2.2.1. connect..... | 6 |
| 2.2.2. disconnect..... | 6 |
| 2.2.3. connection_status..... | 7 |
| 2.2.4. get_major_version..... | 7 |
| 2.2.5. get_minor_version..... | 7 |
| 2.2.6. get_build_number..... | 7 |
| 2.2.7. get_fw_version..... | 7 |
| 2.2.8. start_polling..... | 8 |
| 2.2.9. stop_polling..... | 8 |
| 2.2.10. target_reset..... | 8 |
| 2.3. Interface Communication..... | 8 |
| 2.3.1. interface_list..... | 8 |
| 2.3.2. interface_enable..... | 9 |
| 2.3.3. interface_disable..... | 9 |
| 2.3.4. interface_get_configuration..... | 9 |
| 2.3.5. interface_set_configuration..... | 10 |
| 2.3.6. interface_clear_buffer..... | 10 |
| 2.3.7. interface_read_data..... | 10 |
| 2.3.8. interface_write_data..... | 11 |
| 2.4. Auxiliary..... | 11 |
| 2.4.1. Power..... | 11 |
| 3. Revision History..... | 16 |

1. Description

DGILib is a Dynamic-Link Library (DLL) to help software applications communicate with Data Gateway Interface (DGI) devices. See the [Data Gateway Interface user guide](#) for further details. DGILib handles the low-level USB communication and adds a level of buffering for minimizing the chance of overflows.

The library helps parse data streams of high complexity. The timestamp interface is parsed and split into separate buffers for each data source. The power interface is optionally parsed and calibrated using an auxiliary API.

2. API

The API functions are separated into four groups:

| | |
|--------------------------------|--|
| Discovery | Used to discover available devices and get information about them. |
| Housekeeping | Provides version information, connection, and session control. |
| Interface Communication | Handles communication with the various interfaces of DGI. |
| Auxiliary | Extended functionality with interface-specific usage. |

2.1. Discovery

2.1.1. initialize_status_change_notification

Initializes the system necessary for using the status change notification callback mechanisms. A handle will be created to keep track of the registered callbacks. This function must always be called before registering and unregistering notification callbacks.

Function definition

```
void initialize_status_change_notification(uint32_t* handlep)
```

Parameters

handlep Pointer to a variable that will hold the handle

2.1.2. uninitialized_status_change_notification

Uninitializes the status change notification callback mechanisms. This function must be called when shutting down to clean up memory allocations.

Function definition

```
void uninitialized_status_change_notification(uint32_t handle)
```

Parameters

handle Handle to uninitialized

2.1.3. register_for_device_status_change_notifications

Registers provided function pointer with the device status change mechanism. Whenever there is a change (device connected or disconnected) the callback will be executed. Note that it is not allowed to connect to a device in the context of the callback function. The callback function has the following definition: typedef void (*DeviceStatusChangedCallback)(char* device_name, char* device_serial, BOOL connected)

Function definition

```
void register_for_device_status_change_notifications(uint32_t handle, DeviceStatusChangedCallback deviceStatusChangedCallback)
```

Parameters

handle Handle to change notification mechanisms

deviceStatusChangedCallback Function pointer that will be called when the devices change

2.1.4. **unregister_for_device_status_change_notifications**

Unregisters previously registered function pointer from the device status change mechanism.

Function definition

```
void unregister_for_device_status_change_notifications(uint32_t handle, DeviceStatusChangedCallBack deviceStatusChangedCallBack)
```

Parameters

| | |
|------------------------------------|--|
| handle | Handle to change notification mechanisms |
| deviceStatusChangedCallBack | Function pointer that will be removed |

2.1.5. **discover**

Triggers a scan to find available devices in the system. The result will be immediately available through the *get_device_count*, *get_device_name* and *get_device_serial* functions.

Function definition

```
void discover(void)
```

Parameters

2.1.6. **get_device_count**

Returns the number of devices detected.

Function definition

```
int get_device_count(void)
```

Parameters

2.1.7. **get_device_name**

Gets the name of a detected device.

A non-zero return value indicates an error.

Function definition

```
int get_device_name(int index, char* name)
```

Parameters

| | |
|--------------|---|
| index | Index of device ranges from 0 to <i>get_device_count</i> - 1 |
| name | Pointer to buffer where name of device can be stored. 100 or more bytes must be allocated |

2.1.8. **get_device_serial**

Gets the serial number of a detected device.

A non-zero return value indicates an error.

Function definition

```
int get_device_serial(int index, char* sn)
```

Parameters

| | |
|--------------|--|
| index | Index of device ranges from 0 to <i>get_device_count</i> - 1 |
|--------------|--|

sn Pointer to buffer where the serial number of the device can be stored. 100 or more bytes must be allocated. This is used when connecting to a device.

2.1.9. **is_msd_mode**

EDBG devices can be set to a mass storage mode where the DGI is unavailable. In such cases the device is still detected by DGILib, but it won't be possible to directly connect to it. This command is used to check if the device is in such a mode.

A non-zero return value indicates that the mode must be changed by *set_mode* before proceeding.

Function definition

int is_msd_mode(*char** sn)

Parameters

sn Serial number of the device to check

2.1.10. **set_mode**

This function is used to temporarily set the EDBG to a specified mode.

A non-zero return value indicates an error.

Function definition

int set_mode(*char** sn, *int* nmbed)

Parameters

sn Serial number of the device to set

nmbed 0 - Set to mbed mode. 1 - Set to DGI mode.

2.2. **Housekeeping**

2.2.1. **connect**

Opens a connection to the specified device. This function must be called prior to any function requiring the connection handle.

A non-zero return value indicates an error.

Function definition

int connect(*char** sn, *uint32_t** dgi_hdl_p)

Parameters

sn Buffer holding the serial number of the device to open a connection to

dgi_hdl_p Pointer to a variable that will hold the handle of the connection

2.2.2. **disconnect**

Closes the specified connection.

A non-zero return value indicates an error.

Function definition

int disconnect(*uint32_t* dgi_hdl)

Parameters

dgi_hdl Handle of the connection

2.2.3. connection_status

Verifies that the specified connection is still open.

A non-zero return value indicates an error.

Function definition

int connection_status(*uint32_t** dgi_hdl)

Parameters

dgi_hdl Handle of the connection

2.2.4. get_major_version

A non-zero return value indicates an error.

Function definition

int get_major_version(*void*)

Parameters

2.2.5. get_minor_version

A non-zero return value indicates an error.

Function definition

int get_minor_version(*void*)

Parameters

2.2.6. get_build_number

Returns the build number of DGILib. If not supported, returns 0.

Function definition

int get_build_number(*void*)

Parameters

2.2.7. get_fw_version

Gets the firmware version of the DGI device connected. Note that this is the version of the DGI device, and not the tool.

A non-zero return value indicates an error.

Function definition

int get_fw_version(*uint32_t* dgi_hdl, *unsigned char** major, *unsigned char** minor)

Parameters

dgi_hdl Handle of the connection

major Pointer to a variable where the major version will be stored
minor Pointer to a variable where the minor version will be stored

2.2.8. start_polling

This function will start the polling system and start acquisition on enabled interfaces. It is possible to enable/disable interfaces both before and after the polling has been started. However, no data will be transferred until the polling is started.

A non-zero return value indicates an error.

Function definition

int start_polling(*uint32_t* dgi_hdl)

Parameters

dgi_hdl Handle of the connection

2.2.9. stop_polling

This function will stop the polling system and stop acquisition on all interfaces.

A non-zero return value indicates an error.

Function definition

int stop_polling(*uint32_t* dgi_hdl)

Parameters

dgi_hdl Handle of the connection

2.2.10. target_reset

This function is used to control the state of the reset line connected to the target, if available.

A non-zero return value indicates an error.

Function definition

int target_reset(*uint32_t* dgi_hdl, *bool* hold_reset)

Parameters

dgi_hdl Handle of the connection

hold_reset True will assert reset, false will release it

2.3. Interface Communication

2.3.1. interface_list

Queries the connected DGI device for available interfaces. Refer to the DGI documentation to resolve the ID.

A non-zero return value indicates an error.

Function definition

int interface_list(*uint32_t* dgi_hdl, *unsigned char** interfaces, *unsigned char** count)

Parameters

- dgi_hndl*** Handle to connection
- interfaces*** Buffer to hold the ID of the available interfaces. Should be able to hold minimum 10 elements, but a larger count should be used to be future proof.
- count*** Pointer to a variable that will be set to the number of interfaces registered in buffer.

2.3.2. **interface_enable**

Enables the specified interface. Note that no data acquisition will begin until a session has been started. A non-zero return value indicates an error.

Function definition

int interface_enable(*uint32_t* dgi_hndl, *int* interface_id, *bool* timestamp)

Parameters

- dgi_hndl*** Handle to connection
- interface_id*** The ID of the interface to enable
- timestamp*** Setting this to true will make the interface use timestamping. Consult the DGI documentation for details on the timestamping option.

2.3.3. **interface_disable**

Disables the specified interface. A non-zero return value indicates an error.

Function definition

int interface_disable(*uint32_t* dgi_hndl, *int* interface_id)

Parameters

- dgi_hndl*** Handle to connection
- interface_id*** The ID of the interface to enable

2.3.4. **interface_get_configuration**

Gets the configuration associated with the specified interface. Consult the DGI documentation for details. A non-zero return value indicates an error.

Function definition

int interface_get_configuration(*uint32_t* dgi_hndl, *int* interface_id, *unsigned int** config_id, *unsigned int** config_value, *unsigned int** config_cnt)

Parameters

- dgi_hndl*** Handle to connection
- interface_id*** The ID of the interface
- config_id*** Buffer that will hold the ID field for the configuration item
- config_value*** Buffer that will hold the value field for the configuration item

config_cnt Pointer to variable that will hold the count of stored configuration items

2.3.5. **interface_set_configuration**

Sets the given configuration fields for the specified interface. Consult the DGI documentation for details.
A non-zero return value indicates an error.

Function definition

int interface_set_configuration(*uint32_t* dgi_hdl, *int* interface_id, *unsigned int** config_id, *unsigned int** config_value, *unsigned int* config_cnt)

Parameters

dgi_hdl Handle to connection
interface_id The ID of the interface
config_id Buffer that holds the ID field for the configuration items to set
config_value Buffer that holds the value field for the configuration items to set
config_cnt Number of items to set

2.3.6. **interface_clear_buffer**

Clears the data in the buffers for the specified interface.
A non-zero return value indicates an error.

Function definition

int interface_clear_buffer(*uint32_t* dgi_hdl, *int* interface_id)

Parameters

dgi_hdl Handle to connection
interface_id The ID of the interface

2.3.7. **interface_read_data**

Reads the data received on the specified interface. This should be called regularly to avoid overflows in the system. DGILib can buffer 10Msamples.

A non-zero return value indicates an error.

Function definition

int interface_read_data(*uint32_t* dgi_hdl, *int* interface_id, *unsigned char** buffer, *unsigned long long** timestamp, *int** length, *unsigned int** ovf_index, *unsigned int** ovf_length, *unsigned int** ovf_entry_count)

Parameters

dgi_hdl Handle to connection
interface_id The ID of the interface
buffer Buffer that will hold the received data. The buffer must have allocated 10M elements.
timestamp If timestamp is enabled for the interface, the buffer that will hold the received data. The buffer must have allocated 10M elements. Otherwise send 0.

| | |
|-------------------------------|---|
| <i>length</i> | Pointer to a variable that will hold the count of elements received |
| <i>ovf_index</i> | Reserved. Set to 0. |
| <i>ovf_length</i> | Reserved. Set to 0. |
| <i>ovf_entry_count</i> | Reserved. Set to 0. Could be set to a pointer to a variable that can be used as an indicator of overflows. Overflow would be indicated by non-zero value. |

2.3.8. **interface_write_data**

Writes data to the specified interface. A maximum of 255 elements can be written each time. An error return code will be given if data hasn't been written yet.

A non-zero return value indicates an error. An error will be returned if the interface is still in the process of writing data. Wait a while and try again. The function `get_connection_status` can be used to verify if there is an error condition.

Function definition

```
int interface_write_data(uint32_t dgi_hdl, int interface_id, unsigned char* buffer, int* length)
```

Parameters

| | |
|----------------------------|---|
| <i>dgi_hdl</i> | Handle to connection |
| <i>interface_id</i> | The ID of the interface |
| <i>buffer</i> | Buffer that will hold the received data. The buffer must have allocated 10M elements. |
| <i>length</i> | Pointer to a variable that will hold the count of elements received |

2.4. **Auxiliary**

2.4.1. **Power**

The power interface (as found on some EDBG kits and Power Debugger) uses a protocol stream and calibration scheme that can be tricky to get right. The data rates are also relatively high and the calibration procedure could cause issues if not handled efficiently. Therefore some auxiliary functions to help with this have been made to perform parsing and calibration.

2.4.1.1. **auxiliary_power_initialize**

Initializes the power parser.

A non-zero return value indicates an error.

Function definition

```
int auxiliary_power_initialize(uint32_t* power_hdl_p, uint32_t dgi_hdl)
```

Parameters

| | |
|---------------------------|---|
| <i>power_hdl_p</i> | Pointer to variable that will hold the handle to the power parser |
| <i>dgi_hdl</i> | Handle to connection |

2.4.1.2. **auxiliary_power_uninitialize**

Uninitializes the power parser.

A non-zero return value indicates an error.

Function definition

int auxiliary_power_uninitialize(*uint32_t* power_hdl)

Parameters

power_hdl Handle to the power parser

2.4.1.3. auxiliary_power_register_buffer_pointers

Registers a set of pointers to be used for storing the calibrated power data. The buffers can then be locked by auxiliary_power_lock_data_for_reading, and the data directly read from the specified buffers. Zero-pointers can be specified to get the buffers allocated within DGILib. This requires the data to be fetched using auxiliary_power_copy_data.

A non-zero return value indicates an error.

Function definition

int auxiliary_power_register_buffer_pointers(*uint32_t* power_hdl, *float** buffer, *double** timestamp, *size_t** count, *size_t* max_count, *int* channel, *int* type)

Parameters

power_hdl Handle to the power parser

buffer Buffer that will hold the samples. Set to 0 for automatically allocated.

timestamp Buffer that will hold the timestamp for the samples. Set to 0 for automatically allocated.

count Pointer to a variable that will hold the count of samples. Set to 0 for automatically allocated.

max_count Number of samples that can fit into the specified buffers. Or size of automatically allocated buffers.

channel Power channel for this buffer: A = 0, B = 1 (Power Debugger specific)

type Type of power data: Current = 0, Voltage = 1, Range = 2

2.4.1.4. auxiliary_power_unregister_buffer_pointers

Unregisters the pointers for the specified power channel.

A non-zero return value indicates an error.

Function definition

int auxiliary_power_unregister_buffer_pointers(*uint32_t* power_hdl, *int* channel, *int* type)

Parameters

power_hdl Handle to the power parser

channel Power channel for this buffer: A = 0, B = 1 (Power Debugger specific)

type Type of power data: Current = 0, Voltage = 1, Range = 2

2.4.1.5. auxiliary_power_calibration_is_valid

Checks the status of the stored calibration.

Returns true if the calibration is valid, false otherwise. Unity gain and offset will be used.

Function definition

bool auxiliary_power_calibration_is_valid(*uint32_t* power_hdl)

Parameters

power_hdl Handle to the power parser

2.4.1.6. auxiliary_power_trigger_calibration

Triggers a calibration of the specified type. This can take some time, so use `auxiliary_power_get_status` to check for completion.

A non-zero return value indicates an error.

Function definition

`int auxiliary_power_trigger_calibration(uint32_t power_hdl, int type)`

Parameters

power_hdl Handle to the power parser

type Type of calibration to trigger. See the DGI documentation for details.

2.4.1.7. auxiliary_power_get_calibration

Gets the raw calibration read from the tool.

A non-zero return value indicates an error.

Function definition

`int auxiliary_power_get_calibration(uint32_t power_hdl, uint8_t* data, size_t length)`

Parameters

power_hdl Handle to the power parser

data Buffer that will hold the read raw calibration data

length Number of raw calibration bytes to fetch. See the DGI documentation for number of bytes.

2.4.1.8. auxiliary_power_get_circuit_type

Gets the type of power circuit.

A non-zero return value indicates an error.

Function definition

`int auxiliary_power_get_circuit_type(uint32_t power_hdl, int* circuit)`

Parameters

power_hdl Handle to the power parser

circuit Pointer to a variable that will hold the circuit type: OLD_XAM = 0x00, XAM = 0x10, PAM = 0x11, UNKNOWN = 0xFF

2.4.1.9. auxiliary_power_get_status

Gets the status of the power parser.

Return codes

- IDLE = 0x00
- RUNNING = 0x01
- DONE = 0x02

- CALIBRATING = 0x03
- INIT_FAILED = 0x10
- OVERFLOWED = 0x11
- USB_DISCONNECTED = 0x12
- CALIBRATION_FAILED = 0x20

Function definition

int auxiliary_power_get_status(*uint32_t* power_hdl)

Parameters

power_hdl Handle to the power parser

2.4.1.10. auxiliary_power_start

Starts parsing of power data. The power and power sync interfaces are enabled automatically, but note that it is necessary to start the polling separately. This only starts the parser that consumes data from the DGILib buffer.

A non-zero return value indicates an error.

Function definition

int auxiliary_power_start(*uint32_t* power_hdl, *int* mode, *int* parameter)

Parameters

power_hdl Handle to the power parser

mode Sets the mode of capture.

0 - continuous capturing which requires the user to periodically consume the data.

1 - oneshot capturing that captures data until the buffer has been read once, has been filled or the time from the first received sample in seconds equals the specified parameter.

parameter Mode specific

2.4.1.11. auxiliary_power_stop

Stops parsing of power data.

A non-zero return value indicates an error.

Function definition

int auxiliary_power_stop(*uint32_t* power_hdl)

Parameters

power_hdl Handle to the power parser

2.4.1.12. auxiliary_power_lock_data_for_reading

Blocks the parsing thread from accessing all the buffers. This must be called before the user application code accesses the buffers, or a call to auxiliary_power_copy_data is made. Afterwards auxiliary_power_free_data must be called. Minimize the amount of time between locking and freeing to avoid buffer overflows.

A non-zero return value indicates an error.

Function definition

int auxiliary_power_lock_data_for_reading(*uint32_t* power_hdl)

Parameters

power_hdl Handle to the power parser

2.4.1.13. auxiliary_power_copy_data

Copies parsed power data into the specified buffer. Remember to lock the buffers first. If the count parameter is the same as max_count there is probably more data to be read. Do another read to get the remaining data.

A non-zero return value indicates an error.

Function definition

int auxiliary_power_copy_data(*uint32_t* power_hdl, *float** buffer, *double** timestamp, *size_t** count, *size_t* max_count, *int* channel, *int* type)

Parameters

power_hdl Handle to the power parser

buffer Buffer that will hold the data

timestamp Buffer that will hold the timestamps

count Pointer to a variable that will hold the count of elements copied

max_count Maximum number of elements that the buffer can hold

channel Power channel for this buffer: A = 0, B = 1 (Power Debugger specific)

type Type of power data: Current = 0, Voltage = 1, Range = 2

2.4.1.14. auxiliary_power_free_data

Clears the power data buffers and allows the power parser to continue.

A non-zero return value indicates an error.

Function definition

int auxiliary_power_free_data(*uint32_t* power_hdl)

Parameters

power_hdl Handle to the power parser

3. Revision History

| Doc Rev. | Date | Comments |
|----------|---------|---------------------------|
| 42771A | 09/2016 | Initial document release. |



Atmel® | Enabling Unlimited Possibilities®



Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**

© 2016 Atmel Corporation. / Rev.: Atmel-42771A-DGILib_User Guide-09/2016

Atmel®, Atmel logo and combinations thereof, Enabling Unlimited Possibilities®, and others are registered trademarks or trademarks of Atmel Corporation in U.S. and other countries. ARM®, ARM Connected® logo and others are the registered trademarks or trademarks of ARM Ltd. Other terms and product names may be trademarks of others.

DISCLAIMER: The information in this document is provided in connection with Atmel products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Atmel products. EXCEPT AS SET FORTH IN THE ATMEL TERMS AND CONDITIONS OF SALES LOCATED ON THE ATMEL WEBSITE, ATMEL ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS AND PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF ATMEL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Atmel makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and products descriptions at any time without notice. Atmel does not make any commitment to update the information contained herein. Unless specifically provided otherwise, Atmel products are not suitable for, and shall not be used in, automotive applications. Atmel products are not intended, authorized, or warranted for use as components in applications intended to support or sustain life.

SAFETY-CRITICAL, MILITARY, AND AUTOMOTIVE APPLICATIONS DISCLAIMER: Atmel products are not designed for and will not be used in connection with any applications where the failure of such products would reasonably be expected to result in significant personal injury or death ("Safety-Critical Applications") without an Atmel officer's specific written consent. Safety-Critical Applications include, without limitation, life support devices and systems, equipment or systems for the operation of nuclear facilities and weapons systems. Atmel products are not designed nor intended for use in military or aerospace applications or environments unless specifically designated by Atmel as military-grade. Atmel products are not designed nor intended for use in automotive applications unless specifically designated by Atmel as automotive-grade.