Diolan

DLN-Series Interface Adapters
C/C++ API Manual

Supported Products:

DLN-1, DLN-2, DL-4M, DLN-4S

Version 2.0

30 November 2015

C/C++ API

This manual describes DLN-series adapters and how they can be used to communicate with third-party client applications.

This manual includes API documentation. The APIs are implemented in C and C++.

You are not required to use C or C++ programming language to interface the DLN series adapters. Most of the programming languages can use C-style shared libraries (DLL in Windows), while the function-calling conventions may differ. That should not be a problem, because most programming languages can call functions supplied by the operation system. These functions are mostly implemented in C-style shared libraries. If you experience problems with interfacing the *dln.dll* library from your application, feel free to submit a question to the dlnware.com forum.

We provide the specialized libraries, examples and API documentation for .Net languages and LabView.

The documentation is constantly growing with new content. If you can't find the required information, please post a question to the dlnware.com forum and our engineers will assist you.

1. Communication with Adapter

Internally, the communication with the adapter is implemented by sending data blocks to and from the adapter. These blocks of data represent commands, responses and events.

You do not need to understand all these data structures to write your application. Most functionality is encapsulated in the regular C-style functions (for example, **`DlnOpenDevice()`**, **`DlnI2cMasterTransfer()`**, etc.). All you need to do is to call these functions and the rest of the work will be done by the *dln.dll* shared library and the device driver.

If you want to benefit from the event-driven interface, or to optimize the speed of the data transfer between your application and the adapter, you can return and read the corresponding chapter later.

1.1 Device Opening & Identification

DLN API is designed for a maximum flexibility. It allows to build simple applications that interface a single device with only a few lines of code. It also provides a lot of functions to build more complex applications, that can interface several adapters simultaneously, even if they are connected to different computers. There is a number of approaches to distinguish between different adapters, different types of adapters, and check what subset of the functionality supports the adapter in use.

We will cover all the details in the following subsections. Some information may seem to be complex in the beginning. Feel free to skip it and move to the next chapter. Most of the applications need only two functions to establish the connection with the DLN series adapter - **`DlnConnect()`** and **`DlnOpenDevice()`**. You can even omit the **`DlnConnect()`** function in the Direct Interface mode.

The resources allocated by these two functions are automatically cleared up when the application terminates. Nevertheless, it is a good practice to explicitly close the device handle and connection to the DLN server by calling the **`DlnCloseHandle()`** and **`DlnDisconnect()`** functions.

1.2 Connecting to the DLN Server

You can interface the DLN series adapters either directly or through the DLN server application. The DLN server is a Windows Service or Linux / Mac OS X Daemon. The applications communicate with the DLN server through TCP/IP. The most prominent difference between the Direct and the Server Based interfaces is that the Direct Interface can provide you with the higher bandwidth when you transfer a large amount of data, while the Server Based Interface allows several applications to communicate with the same adapter simultaneously. For additional details please refer to the Interface Types chapter in the User Manual.

Server Based Interface

Without going into details (the details are described in the Server Base Interface implementation chapter), you need to connect to the DLN server if you use the Server Based Interface. The connection is established by calling the **`DlnConnect()`** function. The DLN Server IP address and TCP/IP port number are passed to this function as parameters.

If the DLN series adapter is connected to the computer where you launch your application, and you don't change the default port configuration, you can use the **`DlnConnectDefault()`** function to connect to the DLN server.

As with most of the functions that allocate resources, the established connection can be closed by calling the **`DlnDisconnect()`** function with the same parameters. Use the **`DlnDisconnectDefault()`** to close the connection established with the **`DlnConnectDefault()`** function. If you want to close all currently opened connections, call the **`DlnDisconnectAll()`** function.

Direct Interface

The DLN shared library for the Direct Interface does not require the connection to the DLN server. It is designed to directly communicate with DLN-series adapters connected to the computer where the application is running. You are not required to call the **`DlnConnect()`**, **`DlnDisconnect()`** and **`DlnDisconnectAll()`** functions in the Direct Interface, but you can do this to make your code compatible with the Server Based Interface. These functions do nothing in the Direct Interface, they simply return the successful result code

1.3 Device Opening

There is a number of functions that you can use to open the DLN-series adapter (to establish the connection with this adapter). We will review one function here. The others will be described in the next chapter - Device Identification.

When the device is opened by a process, the DLN library allocates resources required to establish the connection and associates a device handle with this device. The device handle is used to identify the adapter in many function calls. The device handle is valid until either the process terminates, or the handle is closed using the **`DlnCloseHandle()`** or **`DlnCloseAllHandles()`** function.

You can open the same adapter several times by calling one of the **`DlnOpenXXX()`** functions. We do not recommend this approach and in most cases changing the application architecture eliminates the necessity to open the same hardware repeatedly. Each time you call one of those functions, a new handle is associated with the same hardware and additional resources to manage this handle are allocated. If application design requires you to open the same device multiple times, it is important to close all allocated handles when you don't need them. The C++ programmers can use our C++11-like unique_hdln class to manage the device handle.

The device handle is represented by the HDLN type which is defined in the dln.h file as:

```
typedef uint16_t HDLN;
```

The simplest way to establish the connection to the DLN-series adapter is to call the **`DlnOpenDevice()`** function. This function accepts two parameters - an index number of the adapter to open and the pointer to the variable that receives the device handle.

The **`DlnOpenDevice()`** function is useful when your application is designed to work with a single DLN-series adapter. You can call this function, specifying 0 as a **deviceNumber** parameter as in the following example:

```
HDLN handle;
DLN_RESULT result = DlnOpenDevice(0, &handle);
```

You can also use the **`DlnOpenDevice()`** function to open all available adapters in a loop, but you should not make any assumption about the relation between the **deviceNumber** parameter and the specific hardware. You can call the **`DlnGetDeviceSn()`** or **`DlnGetDeviceId()`** function to identify the device after it is open, or use one of the functions listed in the next chapter to open the specific hardware.

1.4 Device Identification

Every DLN-series adapter has a unique serial number, allocated during the manufacturing. You can obtain the serial number of the device by calling the **`DlnGetDeviceSn()`** function. Use the **`DlnOpenDeviceBySn()`** function to open the device with the specific serial number.

The device serial number can't be changed. We do not recommend to use it to distinguish between the devices that are expected to perform different actions. Using the serial number tightly couples your application to the specific hardware.

There is a much more scalable approach. You can assign an ID number to any DLN-series adapter and then use it to identify the specific hardware. To assign the ID number use our DeviceId.exe application or call the **`DlnSetDeviceId()`** function. The ID number is stored in the internal non-volatile memory. It remains the same even when the adapter is connected to another computer.

When you know the ID number of the specific adapter, you can open it with **`DlnOpenDeviceById()`** function.

1.5 Hardware Type and Supported Functionality

DLN series include a number of different adapters. All these adapters support API described in the current manual, but their available functionality may differ slightly. For example, they can support different SPI and I2C bus frequencies, not all of the adapters implement I2C/SPI slave interfaces, etc.

If you know that your application needs a specific DLN-series adapter, you can check the hardware type of the adapter by calling the **`DlnGetHardwareType()`** function. The hardware type constants are defined in the *dln_generic.h* file as follows:

```
#define DLN_HW_TYPE uint32_t
#define DLN_HW_TYPE_DLN5   ((DLN_HW_TYPE)0x0500)
#define DLN_HW_TYPE_DLN4M  ((DLN_HW_TYPE)0x0401)
#define DLN_HW_TYPE_DLN4S  ((DLN_HW_TYPE)0x0402)
#define DLN_HW_TYPE_DLN3   ((DLN_HW_TYPE)0x0300)
#define DLN_HW_TYPE_DLN2   ((DLN_HW_TYPE)0x0200)
#define DLN_HW_TYPE_DLN1   ((DLN_HW_TYPE)0x0100)
```

You can use the **`DlnOpenDeviceByHwType()`** function to open the adapter with the predefined hardware type.

Instead of checking the specific device type, you can check if the adapter implements the required functionality. DLN API provides you with a number of ways to do this. We will discuss all these ways in the following subsections.

1.5.1 Ports and Pins Count

The DLN API is logically divided into modules. Each module contains a set of functions and declarations for the specific interface (for example, GPIO, I2C Master interface, I2C Slave interface, etc.).

You can check if the specific interface is supported by the connected adapter by calling the corresponding **`DlnXXXGetPortCount()`** function.

All these functions return the **`DLN_RESULT`** value. If the interface is supported, the corresponding function returns the **`DLN_RES_SUCCESS`** value. Otherwise the return code is equal to **`DLN_RES_NOT_IMPLEMENTED`**.

If these functions succeed, they fill in the count parameter with the number of available ports.

The example below prints the number of I2C master ports supported by the connected DLN-series adapter:

```
DlnConnect("localhost", DLN_DEFAULT_SERVER_PORT);
 HDLN device;
 DlnOpenDevice(0, &device);

 uint8_t count;
 DLN_RESULT result = DlnI2cMasterGetPortCount(device, &count);
 if (DLN_SUCCEEDED(result))
  printf("The adapter has %d I2C master ports.\n", count);
 else
  printf("The adapter does not support an I2C master interface\n");
```

The GPIO ports consist of 8 pins. If the number of available GPIO pins is not multiple of 8, the last port (with the highest index) can have less that 8 pins. Use the **`DlnGpioGetPinCount()`** function to obtain the number of GPIO pins for the current DLN-series adapter.

1.5.2 Function Return Value

In the previous chapter we saw how you can use the return value of the **`DlnXXXGetPortCount()`** function to check if the corresponding interface is implemented by the DLN-series adapter. The same approach can be applied for all functions from the DLN API.

For example, some of the adapters have internal pull-up resistors on GPIO pins, while others do not. Moreover, the pull-up resistors may be available only for a subset of GPIO pins. You can enable these pull-ups with the **`DlnGpioPinPullupEnable()`** function. This function, as well as all other DLN API functions, returns the **`DLN_RESULT`** value. If the return value is equal to **`DLN_RES_NOT_IMPLEMENTED`**, you can't enable the internal pull-up resistor for current pin.

This approach of checking for the available functionality sometimes has an undesirable side effect. If the function succeeds, it performs an action (for example, **`DlnGpioPinPullupEnable()`** function enables the pull-up resistor if it can do so). If you need to check the supported functionality, but not to perform an action, use the **`DlnGetCommandRestriction()`** function. As you will see in the following section, this function may also provide some additional information about the requested functionality.

1.5.3 Specific Command Restrictions

1.6 Event-Driven Interface

As described in the corresponding section of the User Manual, the event driven interface can save your computational and USB resources and increase the application responsibility.

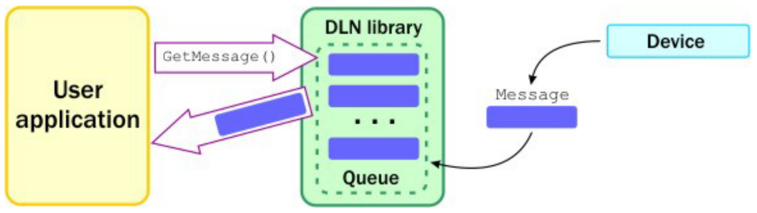
Each module has its own set of available events. The examples of how to handle and configure these events are provided in the corresponding parts of the current manual.

In the following sections we examine the general approach of DLN event processing and review two examples. The first example handles device connection/disconnection events and prints the corresponding messages at the standard output (stdout). The second example can be used as a generic events monitor. In addition to handling device connection/disconnection, it also processes all other event types. It dumps the event data at the standard output (stdout).

1.6.1 Events Queue

1.6.2 Notifications

The DLN library may notify the user application when new messages arrive from the device.



There are 4 different types of notifications:

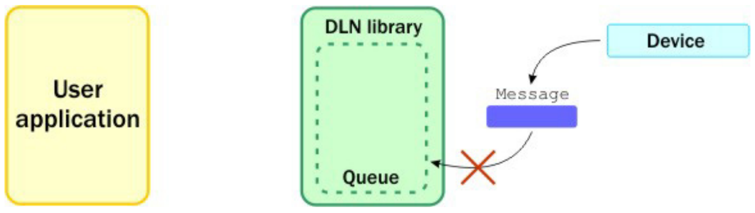
- Callback function.
- Event object.
- Window message.
- Thread message.

You can configure the same notification settings for all the messages. To do so, call the **`DlnRegisterNotification()`** function and specify **`HDLN_ALL_DEVICES(0)`** value as a handle. In this case the DLN library will notify the user application about messages from all devices.

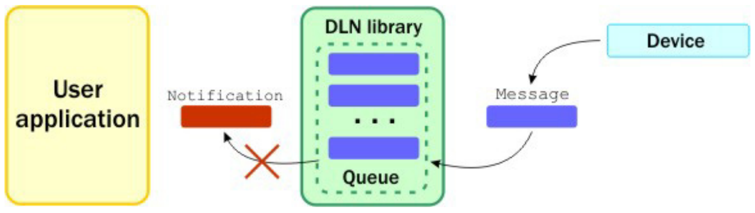
The DLN library may notify the application about messages from a specific device. To configure such notification settings, call the **`DlnRegisterNotification()`** function and specify the handle of the device. Streams (like devices) have their own handles. So, you may configure the notification settings for a specific stream as well.

You can use the **`DlnRegisterNotification()`** function several times, specifying various notification settings for different devices (streams). For example, if you have 4 devices, you may register certain notification settings for one device and different settings for other devices. When a device sends a message, the library checks the notification settings for current device. If the library finds such settings, the notification is generated. If there are no settings for current device, the library checks the notification settings for all devices. If there are no such settings either, the notification isn't generated and the message isn't pushed into the queue.

Sometimes it is useful when messages aren't pushed into the queue. It is most convenient for those who use only synchronous communication. During the synchronous communication the application doesn't call the **`DlnGetMessage()`** function. Thus the messages aren't removed from the queue. It leads to memory leak and eventually to memory overflow. If you don't want messages to be enqueued, you shouldn't register any notification settings.



If you want the messages to be enqueued without notification, do the following. Call the **`DlnRegisterNotification()`** function and specify **`DLN_NOTIFICATION_TYPE_NO_NOTIFICATION`** value as the notification type. In this case the messages will be pushed into the queue without notification to the user application. The messages can be obtained with the help of the **`DlnGetMessage()`** function.



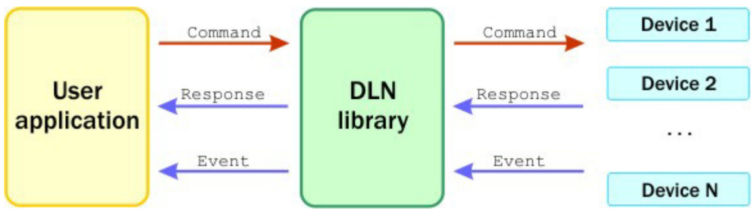
To unregister the notification settings call the **`DlnUnregisterNotification()`** function.

1.7 Messages

The communication with a device is performed by the use of messages. A message is a packet of data that is sent from the library to a device and vice versa. The DLN adapters utilize three types of messages:

- commands;
- responses;
- events.

Here is a short comparison to make things more logical and simple.



Types of messages

	Name	Sender	Recepient	Description
Command	DLN_DO_SOME_ACTION_CMD	User application	Device	Contains an instruction to a device.
Response	DLN_DO_SOME_ACTION_RSP	Device	User application	Contains information about some changes that took place.
Event	DLN_SOMETHING_CHANGED_EV	Device	User application	Contains information about some changes that took place.

Commands are sent from the user application to a device. They contain some instructions to the device. You may instruct the device to perform an action (for example, to change voltage on a pin) or to configure the settings of the device. Each command has corresponding response.

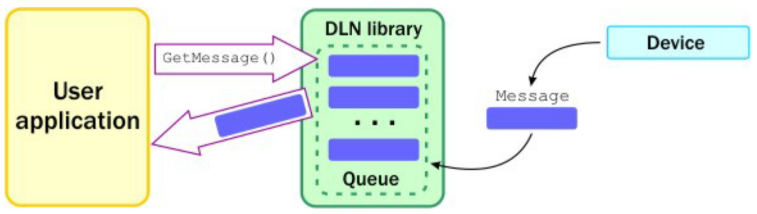
A response is sent from the device to the user application after a command execution. A response always returns the result of the command execution. If the command was successfully executed, the response informs the user application about this. If it is impossible to complete the command, the response returns the error code. Some commands request specific data (for example, the serial number of a device or the total number of connected adapters). In this case the response returns the requested data in addition to the result of the command execution.

Events are sent from the device to the user application. They contain information about some changes that have taken place. A user can predefine the condition of an event generation. For example, an event may be generated when a new device is connected or when voltage changes on an input pin.

All the messages are transferred through the DLN library.

The messages (commands, responses and events) are delivered with the help of three functions:

- `DlnSendMessage()` - sends a specified message (an asynchronous command) to the device.
- `DlnGetMessage()` - retrieves messages (responses and events) sent by the device.
- `DlnTransaction()` - sends a synchronous command, waits for a response and returns the response details. Messages sent by the device (responses and events) are pushed into the DLN library message queue. The user may call the DlnGetMessage() function to get the message from the queue.



The DlnGetMessage() function removes the message from the queue and passes the message details to the user application.

1.8 Simple Generic Module Example

The following example shows how to open DLN device, get its main parameters (hardware version, identifier and serial number), print them to console and close device. You can find the complete example in the "*..\Program Files\Diolan\DLN\examples\c_cpp\examples\simple*" folder after DLN setup package installation.

```
1   #include "..\..\..\common\dln_generic.h"
2   #pragma comment(lib, "..\\..\\..\\bin\\dln.lib")
3
4   int _tmain(int argc, _TCHAR* argv[])
5   {
6   // Open device
7   HDLN device;
8   DlnOpenUsbDevice(&device);
9
10  // Get device parameters
11  DLN_VERSION version;
12  uint32_t sn, id;
13  DlnGetVersion(device, &version);
14  DlnGetDeviceSn(device, &sn);
15  DlnGetDeviceId(device, &id);
16
17  // Print it
18  printf("Device HwType = 0x%x, SN = 0x%x, ID = 0x%x\n",
version.hardwareType, sn, id);
19
20  // Close device
21  DlnCloseHandle(device);
22  return 0;
23  }
```

Line 1: `#include "..\..\..\common\dln_generic.h"`

The *dln_generic..h* header file declares functions and data structures for the generic interface.

Line 2: `#pragma comment(lib, "..\\..\\..\\bin\\dln.lib")`

Use *dln.lib* library while project linking.

Line 8: `DlnOpenUsbDevice(&device);`

The function establishes the connection with the DLN adapter. This application uses the USB connectivity of the adapter. For additional options, refer to the Device Opening & Identification section.

Line 13: `DlnGetVersion(device, &version);`

This function assigns device parameters to the **`DLN_VERSION`** type structure variable.

Line 14: `DlnGetDeviceSn(device, &sn);`

This function assigns device serial number to the provided pointer to 32-bit integer type variable. Serial number is unchangeable for each device and assigned once during device production.

Line 15: `DlnGetDeviceId(device, &id);`

This function assigns device id to the provided pointer to 32-bit integer type variable. Id number can be assigned by user by calling **`DlnSetDeviceId()`** function.

Line 18: `printf("Device HwType = 0x%x, SN = 0x%x, ID = 0x%x\n", version.hardwareType, sn, id);`

Printing the results. In the console you will see hardware type, serial number and id of the connected device.

Line 21: `DlnCloseHandle(device);`

Closing handle to the previously opened DLN-series adapter.

1.9 Generic Functions

You can use the following functions to establish communication with DLN series adapters, identify them using ID and serial numbers, and obtain version information for all components involved in the communication.

DlnConnect() Function

The **`DlnConnect()`** function establishes the connection to the DLN server.

Syntax

```
DLN_RESULT DlnConnect(
   const char* host,
   uint16_t port
);
```

Parameters

host

A server to which the function establishes connection. This value can be a URL (e.g., example.com), or an IP address (e.g., 127.0.0.1). To connect to a server launched on the same computer, you can use the predefined "localhost" value as a host name.

port

A port number of the DLN server. If you use DLN-series device on the same computer use "9656" port value or the predefined **`DLN_DEFAULT_SERVER_PORT`** constant.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully established connection to the DLN server.

Remarks

The **`DlnConnect()`** function is the *dln_generic.h* file.

DlnDisconnect() Function

The **`DlnDisconnect()`** function closes the connection to the specified DLN server.

Syntax

```
DLN_RESULT DlnDisconnect(
   const char* host,
   uint16_t port
);
```

Parameters

host

A server to which the function closes connection. This value can be a URL (e.g., example.com), or an IP address (e.g., 127.0.0.1). To close the connection to a server, launched on the same computer, you can use the predefined "localhost" value as a host name.

port

A port of the DLN server.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully closed connection to the DLN server.

DLN_RES_NOT_CONNECTED (0xA1)

Connection with the specified server is not established.

Remarks

The `DlnDisconnect()` function is defined in the *dln_generic.h* file.

DlnDisconnectAll() Function

The `DlnDisconnectAll()` function closes connections to all servers at once.

Syntax

```
DLN_RESULT DlnDisconnectAll(
);
```

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully closed connection to all DLN servers.

DLN_RES_NOT_CONNECTED (0xA1)

Connections with no servers are established.

Remarks

The `DlnDisconnectAll()` function is defined in the *dln_generic.h* file.

DlnGetDeviceCount() Function

The `DlnGetDeviceCount()` function retrieves the total number of DLN devices available. If connection is established with several DLN servers, this function will return the total number of DLN adapters, connected to all servers.

Syntax

```
DLN_RESULT DlnGetDeviceCount(
   uint32_t* deviceCount
);
```

Parameters

deviceCount

A pointer to an unsigned 32-bit integer that receives the total number of available DLN-series adapters.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the number of available DLN devices.

Remarks

The **`DlnGetDeviceCount()`** function is defined in the *dln_generic.h* file.

DlnOpenDevice() Function

The **`DlnOpenDevice()`** function opens the specified device. This function uses an index number of the device. This number is randomly system-assigned to each connected device. It cannot be used to identify the device. If you need to open a specific device, use the **`DlnOpenDeviceBySn()`** or **`DlnOpenDeviceById()`** functions.

Syntax

```
DLN_RESULT DlnOpenDevice(
   uint32_t deviceNumber,
   HDLN* deviceHandle
);
```

Parameters

deviceNumber

A number of the device.

deviceHandle

A pointer to the HDLN type variable that receives the device handle.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully opened the DLN device.

DLN_RES_NOT_CONNECTED (0xA1)

The library is not connected to any server.

DLN_RES_MEMORY_ERROR (0x86)

Not enough memory to process this command.

DLN_RES_HARDWARE_NOT_FOUND (0x81)

Invalid device number. To check the number of available devices, call the **`DlnGetDeviceCount()`** function.

DLN_RES_DEVICE_REMOVED (0x8E)

The device was disconnected during the function execution.

Remarks

The **`DlnOpenDevice()`** function is defined in the *dln_generic.h* file.

DlnOpenDeviceById() Function

The **`DlnOpenDeviceById()`** function opens the device defined by its ID number.

Syntax

```
DLN_RESULT DlnOpenDeviceById(
   uint32_t id,
   HDLN* deviceHandle
);
```

Parameters

id

An ID of the DLN-series adapter.

deviceHandle

A pointer to the HDLN type variable that receives the device handle.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully opened the DLN device defined by ID.

Remarks

You can change the device ID by using the **`DlnSetDeviceId()`** function.

The **`DlnOpenDeviceById()`** function is defined in the *dln_generic.h* file.

DlnOpenDeviceBySn() Function

The **`DlnOpenDeviceBySn()`** function opens the device defined by its serial number. A device serial number is factory-assigned and cannot be changed.

Syntax

```
DLN_RESULT DlnOpenDeviceBySn(
   uint32_t sn,
   HDLN* deviceHandle
);
```

Parameters

sn

The serial number of the DLN-series adapter.

deviceHandle

A pointer to the HDLN type variable that receives the device handle.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully opened the DLN device defined by serial number.

Remarks

The **`DlnOpenDeviceBySn()`** function is defined in the *dln_generic.h* file.

DlnOpenDeviceByHwType() Function

The **`DlnOpenDeviceByHwType()`** function opens the DLN-series device defined by its type.

Syntax

```
DLN_RESULT DlnOpenDeviceByHwType(
    DLN_HW_TYPE hwType,
    HDLN *deviceHandle
);
```

Parameters

hwType

The `DLN_HW_TYPE` variable that represents the hardware type value. The following values are available:

- DLN_HW_TYPE_DLN5
- DLN_HW_TYPE_DLN4M
- DLN_HW_TYPE_DLN4S
- DLN_HW_TYPE_DLN3
- DLN_HW_TYPE_DLN2
- DLN_HW_TYPE_DLN1

deviceHandle

Pointer to the HDLN type variable that receives the device handle.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully opened the DLN device defined by its hardware type.

Remarks

The **`DlnOpenDeviceByHwType()`** function is defined in the *dln_generic.h* file.

DlnCloseHandle() Function

The **`DlnCloseHandle()`** function closes the handle to an opened DLN-series adapter (stream).

Syntax

```
DLN_RESULT DlnCloseHandle(
   HDLN handle
);
```

Parameters

handle

A handle to the DLN-series adapter.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully closed the handle to an opened DLN adapter.

Remarks

The **`DlnCloseHandle()`** function is defined in the *dln_generic.h* file.

DlnCloseAllHandles() Function

The **`DlnCloseAllHandles()`** function closes handles to all opened DLN-series adapters and streams.

Syntax

```
DLN_RESULT DlnCloseAllHandles(
 );
```

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully closed all opened DLN adapters.

Remarks

The **`DlnCloseAllHandles()`** function is defined in the *dln_generic.h* file.

DlnGetVersion() Function

The **`DlnGetVersion()`** function retrieves the following data about the DLN-series adapter:

- Hardware type - the type of the device (for example, DLN-4M).
- Hardware version - the version of the hardware, used in the device.
- Firmware version - the version of the firmware, installed in the device.
- Server version - the version of the server.
- Library version - the version of the DLN-library.

Syntax

```
DLN_RESULT DlnGetVersion(
    HDLN handle,
    DLN_VERSION* version
 );
```

Parameters

handle

A handle to the DLN-series device.

version

A pointer to a `DLN_VERSION` structure that receives information about the DLN adapter version.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the version information of the DLN adapter.

Remarks

The **`DlnGetVersion()`** function is defined in the *dln_generic.h* file.

DlnGetHardwareType() Function

The **`DlnGetHardwareType()`** function determines the type of the connected DLN device.

Syntax

```
DLN_RESULT DlnGetHardwareType (
    HDLN handle,
    DLN_HW_TYPE *type
);
```

Parameters

handle

A handle to the DLN-series adapter.

type

The pointer to the `DLN_HW_TYPE` type variable that receives the device type.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the type of the DLN adapter.

Remarks

The **`DlnGetHardwareType()`** function is defined in the *dln_generic.h* file.

DlnGetDeviceId() Function

The **`DlnGetDeviceId()`** function retrieves the device ID number.

Syntax

```
DLN_RESULT DlnGetDeviceId(
    HDLN handle,
    uint32_t* id
 );
```

Parameters

handle

A handle to the DLN-series device.

id

A pointer to an unsigned 32-bit integer that receives the ID number.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the DLN device ID.

Remarks

You can change the device ID number using the **`DlnSetDeviceId()`** function.

The **`DlnGetDeviceId()`** function is defined in the *dln_generic.h* file.

DlnSetDeviceId() Function

The **`DlnSetDeviceId()`** function sets a new ID number to the DLN-series adapter.

Syntax

```
DLN_RESULT DlnSetDeviceId(
   HDLN handle,
   uint32_t id
);
```

Parameters

handle

HDLN type variable. A handle to the DLN-series adapter.

id

Unsigned 32-bit integer type variable. An id number to be set.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully specified the DLN device ID.

Remarks

The **`DlnSetDeviceId()`** function is defined in the *dln_generic.h* file.

DlnGetDeviceSn() Function

The **`DlnGetDeviceSn()`** function retrieves the device serial number. A serial number is factory-assigned and cannot be changed.

Syntax

```
DLN_RESULT DlnGetDeviceSn(
    HDLN handle,
    uint32_t* sn
);
```

Parameters

handle

A handle to the DLN-series device.

sn

A pointer to the unsigned 32-bit integer that receives the device serial number.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the serial number of the DLN adapter.

Remarks

The **`DlnGetDeviceSn()`** function is defined in the *dln_generic.h* file.

DlnGetPinCfg() Function

The **`DlnGetPinCfg()`** function retrieves the current configuration of the specified pin of the DLN adapter.

Syntax

```
DLN_RESULT DlnGetPinCfg(
    HDLN handle,
    uint16_t pin,
    DLN_PIN_CFG *cfg
);
```

Parameters

handle

A handle to the DLN-series device.

pin

A pin number.

cfg

A pointer to the `DLN_PIN_CONFIG` structure that receives the pin configuration.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the pin configuration.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid.

Remarks

The **`DlnGetPinCfg()`** function is defined in the *dln_generic.h* file.

DlnRegisterNotification() Function

The **`DlnRegisterNotification()`** function registers notification settings.

Syntax

```
DLN_RESULT DlnRegisterNotification(
    HDLN handle,
    DLN_NOTIFICATION notification
 );
```

Parameters

handle

A handle to the DLN-series adapter. You may specify either the handle to a specific device (stream) or the **`HDLN_ALL_DEVICES`** value to apply the notification setting for all the devices (streams).

notification

Defines the notification settings. The settings are passed as the `DLN_NOTIFICATION` structure.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully registered notification settings.

Remarks

The **`DlnRegisterNotification()`** function is defined in the *dln_generic.h* file.

DlnUnregisterNotification() Function

The **`DlnUnregisterNotification()`** function unregisters notification settings.

Syntax

```
DLN_RESULT DlnUnregisterNotification(
   HDLN handle
);
```

Parameters

handle

A HDLN type handle to the DLN-series adapter (stream). You may specify either the handle to a specific device (stream) or the **`HDLN_ALL_DEVICES`** value to unregister notification setting for all the devices (streams).

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully unregistered notification settings.

Remarks

The **`DlnUnregisterNotification()`** function is defined in the *dln_generic.h* file.

DlnGetMessage() Function

The **`DlnGetMessage()`** function retrieves a message (response or event) sent by the device.

Syntax

```
DLN_RESULT DlnGetMessage(
    HDLN handle,
    void* messageBuffer,
    uint16_t messageSize
 );
```

Parameters

handle

A handle to the DLN-series adapter.

messageBuffer

A pointer to the buffer that receives the message information.

messageSize

The maximum number of bytes that can be retrieved.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the message.

Remarks

The **`DlnGetMessage()`** function is defined in the *dln.h* file.

DlnSendMessage() Function

The **`DlnSendMessage()`** function sends a specified message (an asynchronous command) to the device.

Syntax

```
DLN_RESULT DlnSendMessage(
    void* message
 );
```

Parameters

message

A pointer to a variable that contains a message to be sent.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully sent the message.

Remarks

The **`DlnSendMessage()`** function is defined in the *dln.h* file.

DlnTransaction() Function

The **`DlnTransaction()`** function sends a synchronous command, waits for a response and returns the response details.

Syntax

```
DLN_RESULT DlnTransaction(
   void* command,
   void* responseBuffer,
   uint16_t responseBufferSize
);
```

Parameters

command

A pointer to a variable that contains the command that should be sent.

responseBuffer

A pointer to the buffer that receives the response information.

responseBufferSize

The maximum number of bytes that can be retrieved.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully performed the transaction.

Remarks

The **`DlnTransaction()`** function is defined in the *dln.h* file.

DlnRestart() Function

The **`DlnRestart()`** function restarts the currently opened DLN-series device by its handle.

Syntax

```
DLN_RESULT DlnRestart(
    HDLN handle
);
```

Parameters

handle

A handle to the DLN device.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully reset the specified DLN adapter.

Remarks

The **`DlnRestart()`** function is defined in the *dln_generic.h* file.

DlnGetCommandRestriction() Function

The **`DlnGetCommandRestriction()`** function retrieves the command restrictions under the current conditions. This allows to avoid errors when executing functions. For details, read Specific Command Restrictions.

Syntax

```
DLN_RESULT DlnGetCommandRestriction(
    HDLN handle,
    DLN_MSG_ID msgId,
    uint16_t entity,
    DLN_RESTRICTION *restriction
);
```

Parameters

handle

A handle to the DLN device.

msgId

A code of the command. You can find command codes in header files of the corresponding module.

entity

A number of a pin or a port (depending on the entity used in the command).

restriction

A pointer to the `DLN_RESTRICTION` structure that receives the command restriction.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the command restriction.

Remarks

The `DlnGetCommandRestriction()` function is defined in the *dln_generic.h* file.

2. Linking an Executable to dln.dll

A dynamic-link library (DLL) is an executable file that acts as a shared library of functions.

There are two ways of linking: static and dynamic linking.

Dynamic linking provides a way for a process to call a function that is not part of its executable code. The executable code for the function is located in a DLL, which contains one or more functions that are compiled, linked, and stored separately from the processes that use them. DLLs also facilitate the sharing of data and resources. Multiple applications can simultaneously access the contents of a single copy of a DLL in memory.

Dynamic linking differs from static linking in that it allows an executable module (either a .dll or .exe file) to include only the information needed at run time to locate the executable code for a DLL function.

In static linking, the linker gets all of the referenced functions from the static link library and places it with your code into your executable.

Using dynamic linking instead of static linking offers several advantages. DLLs save memory, reduce swapping, save disk space, upgrade easier, provide after-market support, provide a mechanism to extend the MFC library classes, support multilanguage programs, and ease the creation of international versions.

We will review both methods of linking.

An executable file links to (or loads) a DLL in one of two ways:

- Implicit linking (dynamic linking).
- Explicit linking (static linking).

Implicit Linking

To implicitly link to a DLL, executables must obtain the following from the provider of the DLL:

- A header file (.h file) containing the declarations of the exported functions and/or C++

classes. The classes, functions, and data should all have__declspec(dllimport).

- An import library (.LIB files) to link with. (The linker creates the import library when the DLL is built.)
- The actual DLL (.dll file).

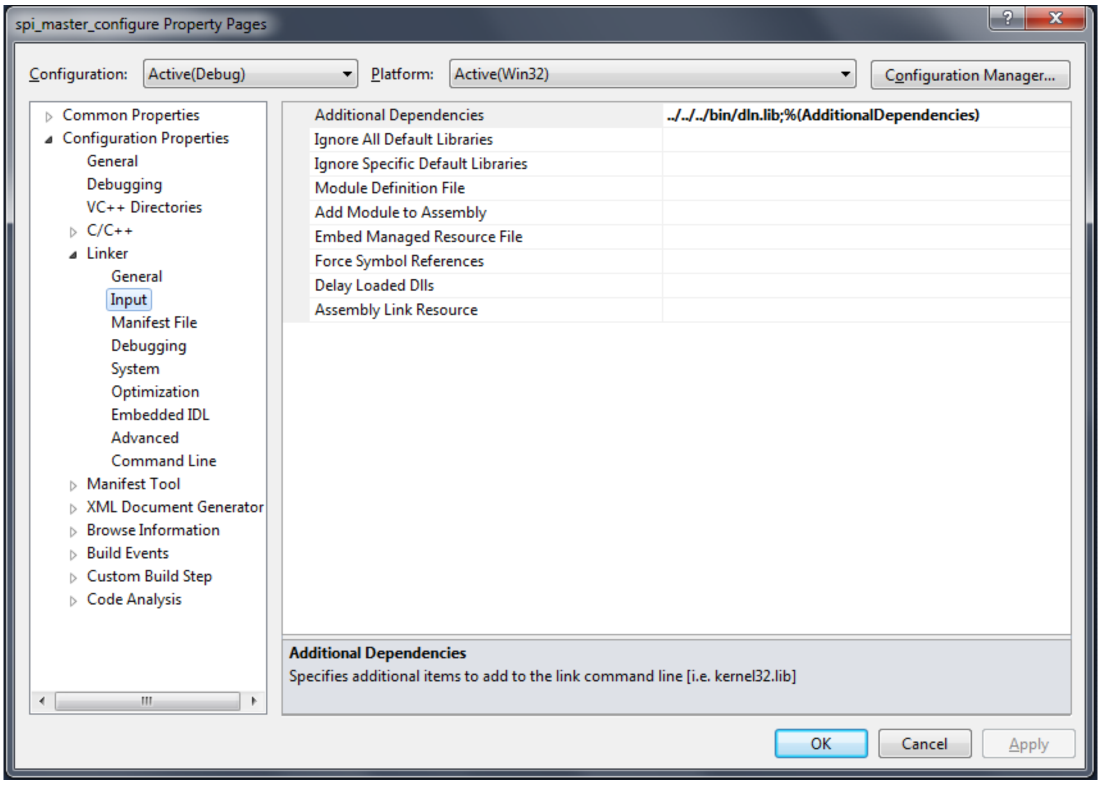
Executables using the DLL must include the header file containing the exported functions (or C++ classes) in each source file that contains calls to the exported functions. From a coding perspective, the function calls to the exported functions are just like any other function call.

All required files for building your own application for DLN-series adapters can be found at '..\Program Files\Diolan\DLN\bin' (contains libraries) and 'Program Files\Diolan\DLN\common' (contains header files) after DLN setup package installation.

To build the calling executable file, you must link with the import library. If you are using an external makefile, specify the file name of the import library where you list other object (.obj) files or libraries that you are linking with.

The operating system must be able to locate the DLL file when it loads the calling executable.

You can connect dln.dll library in Visual Studio project by opening Property Pages Dialog, choosing Linker > Input and adding path string to dln.lib to Additional Dependencies field.



Additional Dependencies	../../../bin/dln.lib;%(AdditionalDependencies)
Ignore All Default Libraries	
Ignore Specific Default Libraries	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded Dlls	
Assembly Link Resource	

Additional Dependencies
Specifies additional items to add to the link command line [i.e. kernel32.lib]

Explicit Linking

With explicit linking, applications must make a function call to explicitly load the DLL at run time. To explicitly link to a DLL, an application must:

Call LoadLibrary (or a similar function) to load the DLL and obtain a module handle.

Call GetProcAddress to obtain a function pointer to each exported function that the application wants to call. Because applications are calling the DLL's functions through a pointer, the compiler does not generate external references, so there is no need to link with an import library.

Call FreeLibrary when done with the DLL.

2.1 Configure QT and QT Project with libdln.a for Linux

Configuring QT

To successfully create and compile QT applications and use libdln.a library in Linux, you need to correctly configure and setup QT. This steps should be performed after you successfully performed steps from Software & Hardware Installation in Linux page.

First, download QT sources from QT website (http://download.qt-project.org/official_releases/qt/4.8/4.8.5/qt-everywh...), unpack archive, open terminal in unpacked folder with QT, configure QT with-release and-static flags, then compile QT sources and install QT. For this example QT 4.8.5 version was used.

```
./configure -release -nomake demos -nomake examples
make
make install
```

Configuring QT Project and Connecting libdln.a Library

After QT is compiled you can compile and run any QT application by using properqmake project_name from application sources folder. You can use terminal or QT Creator for creating applications.

device_list_gui project compilation from terminal:

```
path_to_qmake/qmake device_list_gui
make
```

To use libdlb.a library in your application project, you need to add the following to project .pro file:

```
QMAKE_LFLAGS += -static-libgcc
LIBS += /usr/local/lib/libdln.a # path to libdln.a library
```

Also do not forget to include required header .h files to your sources for successful usage API functions from libdln.a. For example:

```
#include "../common/dln.h"
#include "../common/dln_generic.h"
```

3. Return Codes

All DLN API functions use the DLN_RESULT type for return values. The DLN_RESULT type and return codes are defined in dln_result.h file.

The DLN_RESULT values are divided into three types: success values, warnings and error values.

When testing a return value, you could use one of the following macros (they are also defined in dln_result.h file):

```
DLN_SUCCEEDED(Result)
```

Evaluates to TRUE if the Result value is a success type (0 – 0x1F) or a warning type (0x20 – 0x3F).

```
DLN_WARNING(Result)
```

Evaluates to TRUE if the Result value is a warning type (0x20 – 0x3F).

```
DLN_FAILED(Result)
```

Evaluates to TRUE if the Result value is an error type (greater than 0x40).

4. I2C Bus Interface (C/C++ API)

I^{2}C (Inter-Integrated Circuit) is a multi-master, multi-slave, single-ended, serial bus invented by Philips Semiconductor (now NXP Semiconductors). It is typically used for attaching low-speed peripheral ICs to processors and microcontrollers.

I^{2}C can be used to control a wide range of devices: analogue-to-digital and digital-to-analog converters (ADCs and DACs), LCD and OLED displays, keyboards, LED and motor drivers, memory chips and cards (EEPROM, RAM, FERAM, Flash), pressure and temperature sensors and other peripheral devices.

I^{2}C bus specification describes four operating speed categories for bidirectional data transmission:

Standard-mode (Sm)	a bit rate up to 100 kbit/s
Fast-mode (Fm)	a bit rate up to 400 kbit/s
Fast-mode Plus (Fm+)	a bit rate up to 1 Mbit/s
High-speed mode (Hs)	a bit rate up to 3.4 Mbit/s

One more speed category, Ultra-fast mode (UFm), stands for unidirectional data transmission up to 5 Mbit/s.

DLN-series adapters can operate in Standard, Fast and Fast Plus modes.

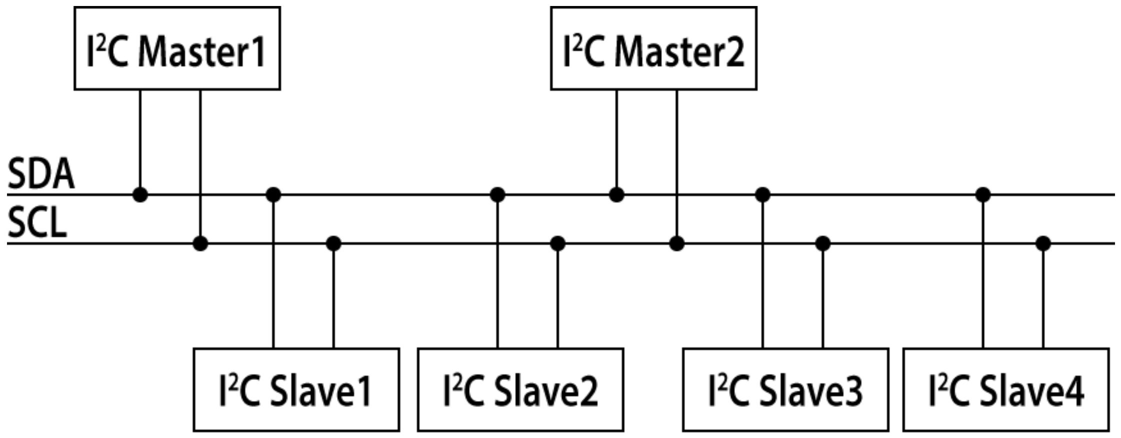
4.1 I2C Bus Protocol

I^{2}C bus uses two lines – SDA (Serial Data line) and SCL (Serial Clock line). Every device connected to the I^{2}C bus can operate as either I^{2}C master (generates the clock and initiates communication with slaves) or I^{2}C slave (responds when addressed by the master).

DLN-series adapters can operate as master devices (read I2C Master Interface). Some DLN adapters can also operate as slave devices (read I2C Slave Interface).

The I^2C bus is a bidirectional bus, but this does not mean that the data is transmitted in both directions simultaneously. At every particular moment, either master or slave sends data over the I^2C bus. The device that sends data to the bus is called **Transmitter**. The device that receives data from the bust is called **Receiver**. Most I^2C devices can both transmit and receive data. However, some I^2C devices are only able to receive data. DLN adapters are capable of transmitting and receiving data.

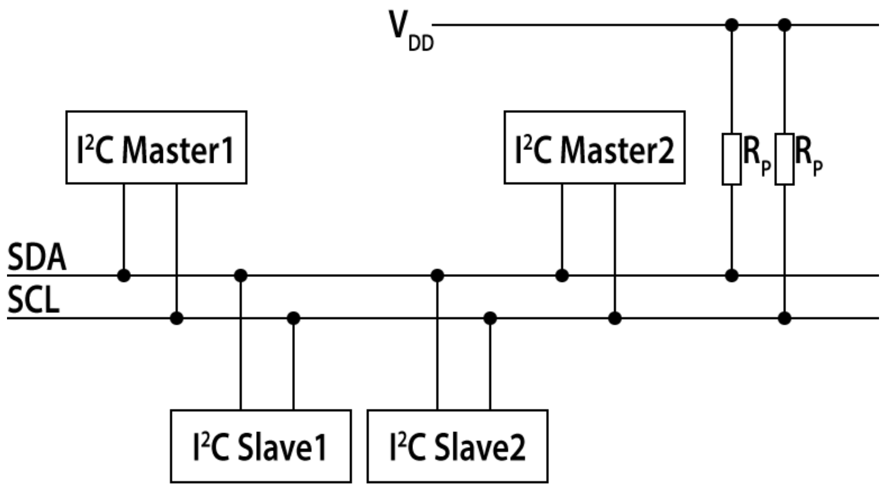
The I^2C bus is a multi-master bus; that means that any number of master devices can be present. The DLN-series adapters support clock synchronization and arbitration to avoid conflicts with other master devices on the same I^2C bus. Read Avoiding Conflicts in a Multi-master I2C Bus for details.



The maximum number of slave devices is limited by the address space. Each slave device has a unique address. The I^2C bus can use 7 or 10-bit addressing. The DLN-series adapters use 7-bit addressing.

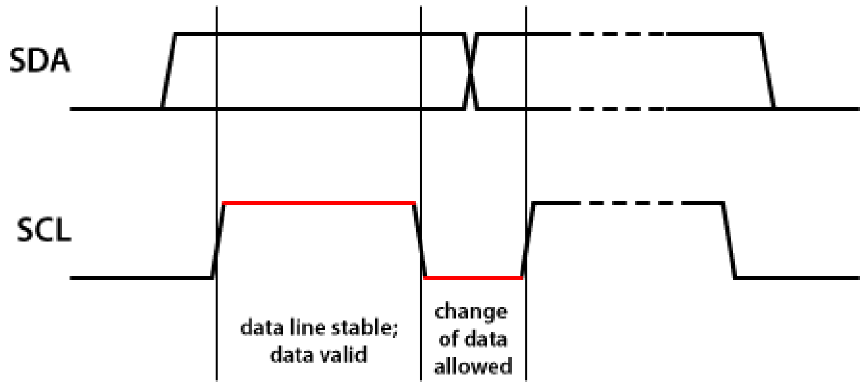
4.1.1 Data Transmission

Both SDA and SCL are bidirectional lines, connected to a positive supply voltage via a pull-up resistor.



All the devices connected to the I^2C bus must have an open-drain (or open-collector) output stages – they can either pull the bus low or be in high-impedance. When there is no data transmission on the I^2C bus, both lines are HIGH. In this case, we say that the I^2C bus is free.

I^2C master generates clock signal on the SCL line. One SCL pulse is generated for each data bit. The data on the SDA line must be stable during the HIGH period of the clock (while SCL line is high). The changes on the SDA line occur when the SCL line is LOW. The only exception from this rule are START, STOP and Repeated START Conditions described later.



4.1.2 Byte Format and Acknowledge Bit

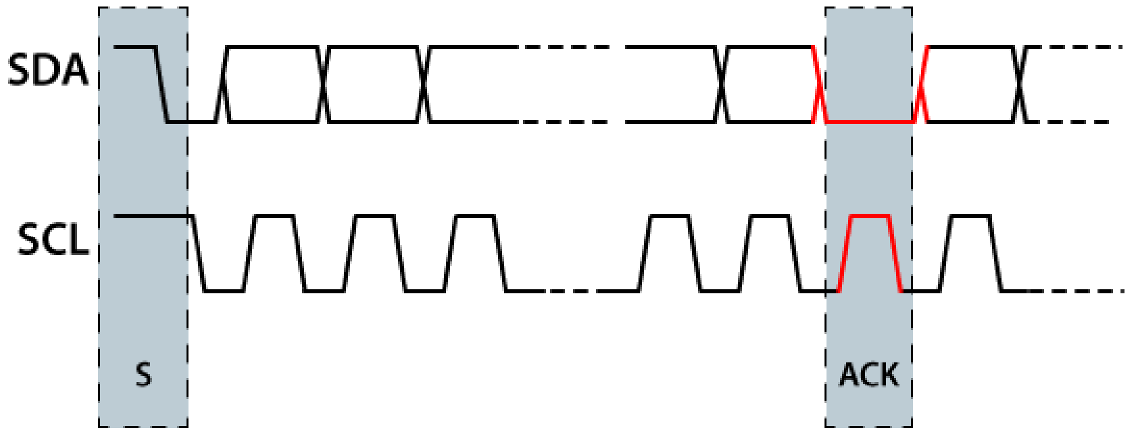
Every byte, sent over the I^2C bus, is eight bits long. Data is transferred with the Most Significant Bit (MSB) first. An Acknowledge bit must follow each byte. The master generates all clock pulses, including the acknowledge clock pulse.

The Acknowledge bit allows:

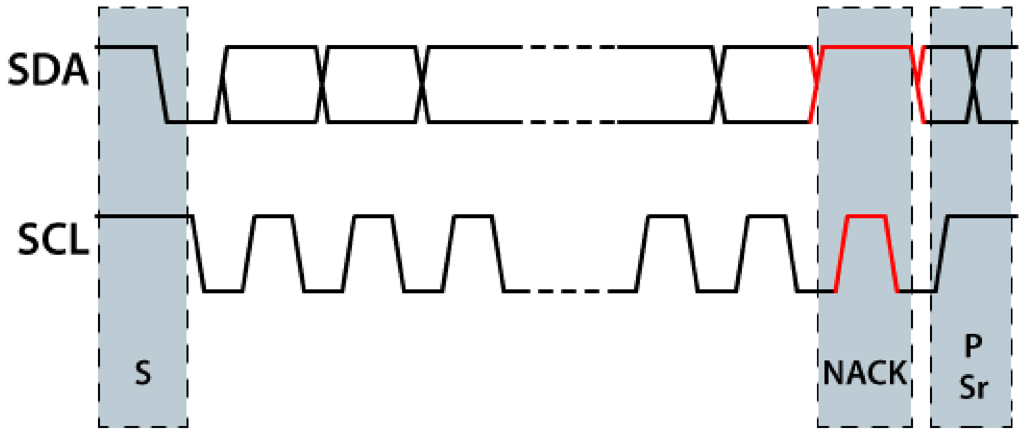
- The receiver to signal the transmitter that the byte was successfully received and another byte may be sent;
- The receiver to signal the transmitter that it received enough data and the transmission should be terminated;
- The slave to signal the master that the specified slave address is present on the bus and transmission can start (see Slave Address and Data Direction);
- The slave to delay the transmission, while it prepares for another byte of data (see Clock Stretching for details).

After transmission of the last eighth bit of data, the transmitter releases the SCL line (during the low phase of clock). This gives an opportunity to receiver to acknowledge (or not acknowledge) the data.

If the receiver pulls the line LOW during the HIGH period of the ninth clock pulse, it acknowledges the byte (the **Acknowledge (ACK)** signal).



The **Not Acknowledge (NACK)** signal is defined when SDA remains HIGH during this clock pulse. The master then can generate either a STOP (P) condition to abort the transmission, or a repeated START (Sr) condition to start a new transmission.



The following conditions can lead to the Not Acknowledged (NACK) signal:

1. There is no device to acknowledge the slave address – no slave with the specified address is connected to the I^2C bus.
2. The slave is unable to receive or transmit – it is busy performing another function.
3. The slave does not support the specified data direction (read or write).
4. The receiver gets data that it does not understand.
5. The receiver cannot receive any more data bytes.
6. A master-receiver must signal the end of the transmission to the slave-transmitter.

4.1.3 I2C Transaction

The data transmission includes the following steps:

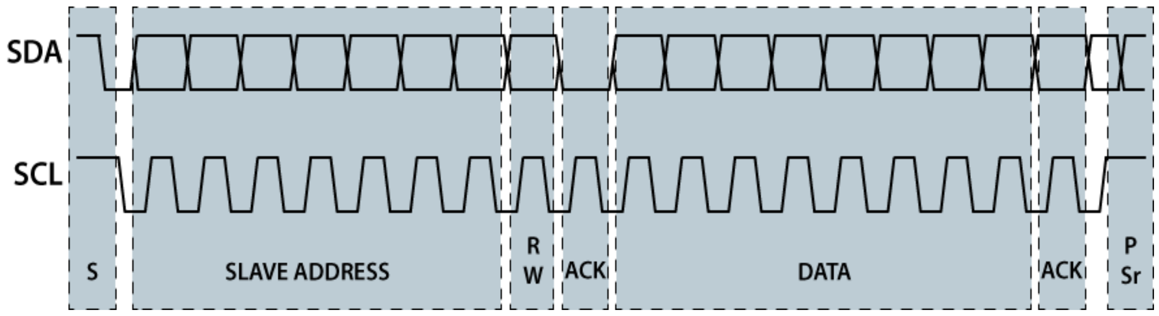
1. The master initiates communication by generating a START (S) Condition;
2. The master sends the first byte that includes a Slave Address and Data Direction;

3. The slave generates the acknowledgement (ACK) signal. If the master receives no acknowledgement signal, it generates the STOP (P) condition to terminate the transmission.

4. The transmitter (master or slave) writes a byte of data to the bus and the receiver (slave or master) reads this byte of data from the bus.

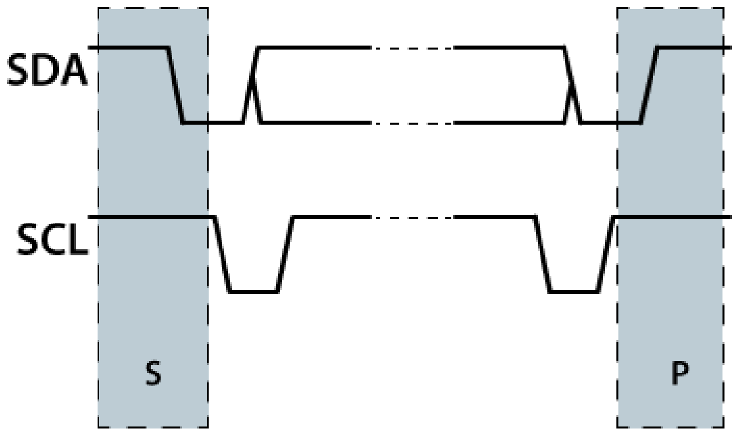
5. After each byte of data, the receiver sends the acknowledgement (ACK) signal and the transmission continues. If the receiver sends no acknowledgement signal, the transmitter stops writing data to the I^2C bus.

6. To terminate transmission, the master generates the STOP (P) Condition. To change transmission parameters, the master generates the Repeated START (Sr) Condition.



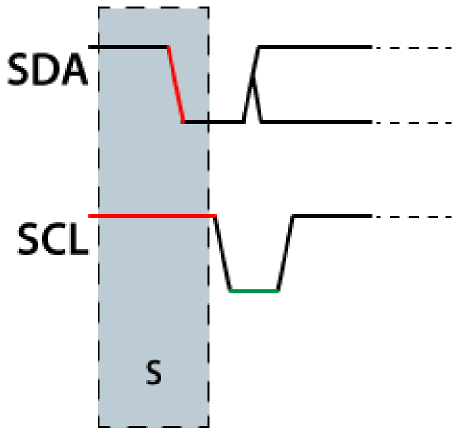
START, STOP and Repeated START Conditions

All transactions begin with a START (S) condition and finish with a STOP (P) condition.



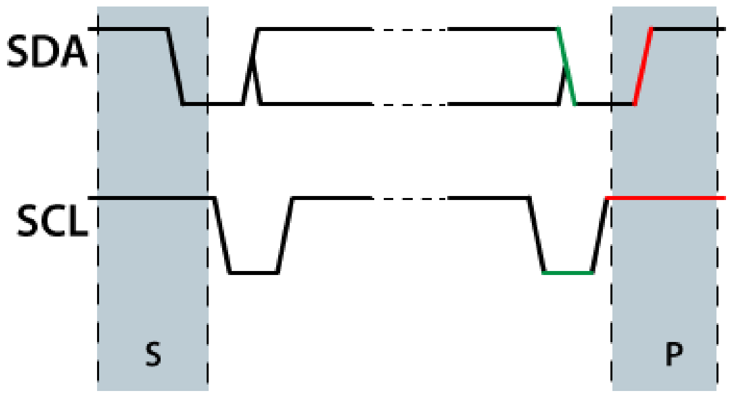
START (S) Condition

To generate a **START condition**, the master changes the SDA line from one to zero while the SCL line is HIGH (marked in red on the following diagram). The I^2C bus is considered busy after the START condition. To prepare the bus for transmission of the first bit, the master outputs zero on the SCL line (marked in green).



STOP (P) Condition

To generate a **STOP condition**, the master changes the SDA line from zero to one while the SCL line is HIGH (marked in red). The I^{2}C bus is considered free after the STOP condition. To prepare for the STOP condition, the master sets the SDA line to zero during the LOW phase of the SCL line (marked in green).

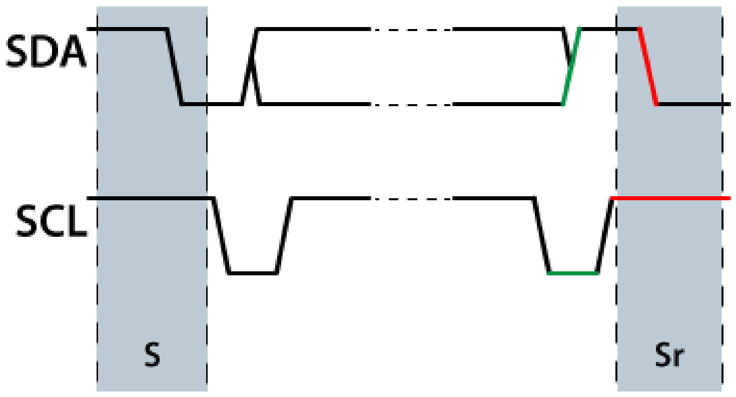


Repeated START (Sr) Condition

Instead of the STOP condition, the master can generate a **repeated START (Sr) condition**. Like a START condition, to generate a repeated START condition, the master changes the SDA line from one to zero while the SCL line is HIGH (marked in red). In this case, the I^{2}C bus remains busy. To prepare for the repeated START condition, the master sets the SDA line to one during the LOW phase of the SCL line (marked in green).

The START (S) and repeated START (Sr) conditions are functionally identical. The repeated start conditions is used in the following situations:

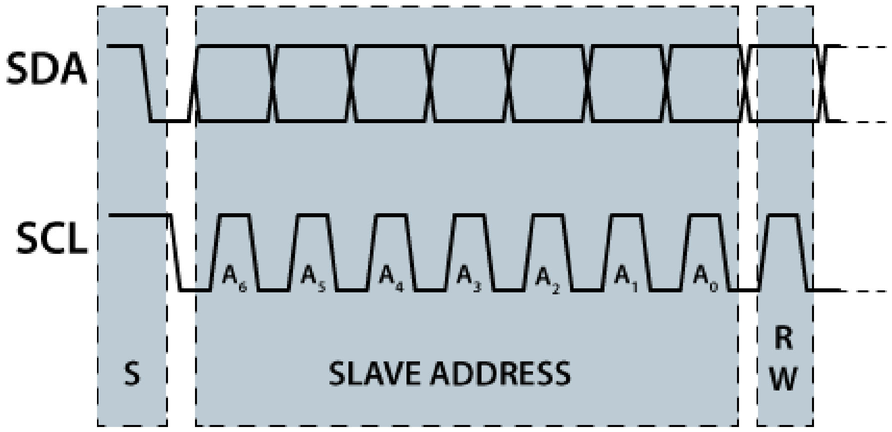
- To continue transmission with the same slave device in the opposite direction. After the repeated START condition, the master sends the same slave device address followed by another direction bit.
- To start transmission to or from another slave device. After the repeated START condition, the master sends another slave address.
- To provide a READ operation from internal address. See READ Operation for details.



DLN adapters use the repeated START condition to read from the internal address (the `DlnI2cMasterRead()` function) and to write to and then read from the same slave device (the `DlnI2cMasterTransfer()` function). If a DLN adapter needs to communicate with different slaves, it finishes one transmission (with the STOP condition) and starts another transmission.

Slave Address and Data Direction

Every byte on the SDA line must be eight bits long. The first byte after START contains seven bits of the slave device address and one bit that defines the direction of the transmission.



As any other data, the address is transmitted sequentially starting with the Most Significant Bit (MSB) and ending with the Least Significant Bit (LSB).

The direction bit has the following values:

- 0 – Write: the master transmits data to the slave;
- 1 – Read: the master receives data from the slave.

Reserved I2C Slave Addresses

There are 16 reserved I^{2}C addresses. The following table shows the purposes of these addresses:

I2C slave address	Direction bit (R/W)	Description
0000 000	0	General call address
0000 000	1	START byte

I2C slave address	Direction bit (R/W)	Description
0000 001	X	CBUS address
0000 010	X	Reserved for different bus format
0000 011	X	Reserved for future purposes
0000 1XX	X	Hs-mode master code
1111 1XX	1	Device ID
1111 0XX	X	10-bit slave addressing

The general call address is for addressing all devices connected to the I^{2}C bus. If a device does not need the provided data, it can ignore this address (it does not issue the acknowledgement). If a device requires data from a general call address, it acknowledges this address and behaves as a slave-receiver. If one or more slaves acknowledge the general call address, the master does not know how many devices did it and does not see not-acknowledged slaves.

If you use a DLN-series adapter as I^{2}C slave, you can configure it to support general call addressing or to ignore it.

4.1.4 Using Internal Addresses

Some I^{2}C slave devices have fixed internal address setting. The internal address is the slave's internal register. Possible internal addresses depend on the slave device. Some very simple devices do not have any, but most do.

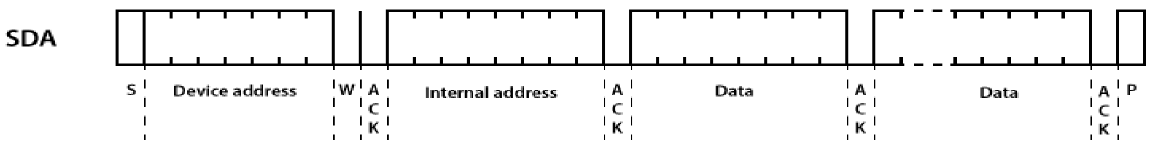
To communicate with a certain register, after the I^{2}C master addressed the slave device and received acknowledgement, it sends the internal address inside the slave where it wants to transmit data to or from.

Both read and write operations can use an internal address. When an internal address is set, the same address is used in every READ and WRITE operations that follows the previous operation.

WRITE Operation

To write to a slave device, the I2C master follows these steps:

1. Sends the START (S) condition.
2. Sends the I^{2}C address of the slave device.
3. Sends the WRITE (W) direction bit.
4. Receives the acknowledgement (ACK) bit.
5. Sends the internal address where it wants to write.
6. Receives the acknowledgement (ACK) bit.
7. Sends data bytes and receives the acknowledgement (ACK) bit after each byte.
8. Sends the STOP (P) condition.



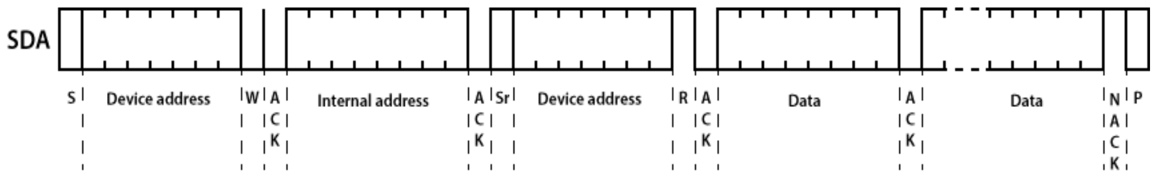
To write data using internal address, call the `DlnI2cMasterWrite()` function. It requires the slave device address (the **slaveDeviceAddress** parameter), the length of the internal address (the **memoryAddressLength** parameter) and the internal address (the **memoryAddress** parameter).

READ Operation

Before reading data from the slave device, the master tells the slave which of the internal addresses it wants to read. Therefore, a read operation starts by writing to the slave.

To read from a slave device, the I^{2}C master follows these steps:

1. Sends a START (S) condition.
2. Sends the I^{2}C address of the slave device.
3. Sends the WRITE (W) direction bit.
4. Receives the acknowledgement (ACK) bit.
5. Sends the internal address where it wants to read from.
6. Receives the acknowledgement (ACK) bit.
7. Sends the repeated START (Sr) condition.
8. Sends the READ (R) direction bit.
9. Receives the acknowledgement (ACK) bit.
10. Receives data bytes and sends acknowledgement (ACK) bits to continue reading or a not acknowledgement (NACK) bit to stop reading.
11. Sends the STOP (P) condition.



To read data using internal address, call the `DlnI2cMasterRead()` function. It requires the slave device address (the **slaveDeviceAddress** parameter), the length of the internal address (the **memoryAddressLength** parameter) and the internal address (the **memoryAddress** parameter).

4.1.5 Avoiding Conflicts in a Multi-master I2C Bus

In a multi-master I^{2}C bus, the collision when more than one master simultaneously initiate data transmission is possible. To avoid the chaos that may ensue from such an event, DLN adapters, like all I^{2}C master devices, support clock synchronization and arbitration. These procedures allow only one master to control the bus; other masters cannot corrupt the winning message.

Clock synchronization allows to perform the level on the SCL line. Arbitration determines which master completes transmission. If a master loses arbitration, it turns off its SDA output driver and stops transmitting data.

Slaves are not involved in clock synchronization and arbitration procedures.

Clock Synchronization

A DLN adapter and one or more I^{2}C masters can begin transmitting on a free I^{2}C bus at the same time. Each master generates its own clock on the SCL line. Therefore, there must be a method for deciding which master generates LOW and HIGH periods of the SCL line. Clock synchronization does it.

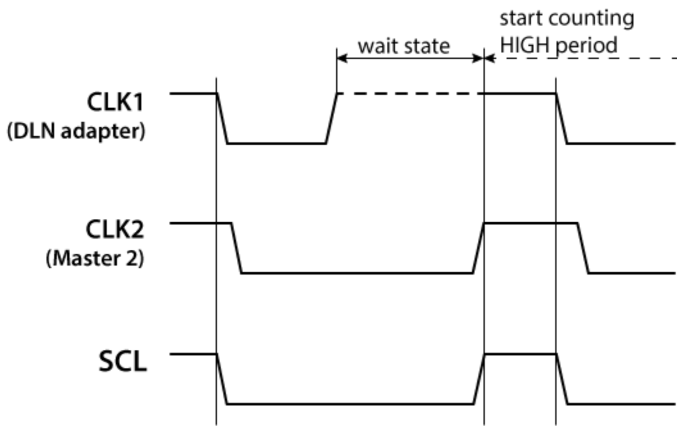
Once a DLN adapter or any other I^{2}C master outputs LOW on its clock line, the SCL line goes LOW. When a master releases its clock line, the SCL line goes HIGH only if no other master has its clock line in LOW state. The master with the longest LOW period holds the SCL line in LOW state. Masters with shorter LOW periods stay in a HIGH wait-state during this time.

When all masters concerned have released their clock lines, the SCL line goes HIGH and all the masters start counting their HIGH periods. The first master that completes its HIGH period pulls the SCL line LOW again.

Therefore, the master with the longest clock LOW period determines a LOW period on the SCL line; the master with the shortest clock HIGH period determines a HIGH period on the SCL line.

Example

The following figure shows clock synchronization for the DLN adapter and the Master2 device. The DLN adapter has a shorter HIGH period; it pulls the SCL line LOW. The Master2 device has a longer LOW period, only when it releases its clock line, both masters start counting HIGH period.



Arbitration

Arbitration, like clock synchronization, is required only if more than one master is used in the system. A master may start transmission only if the bus is free. A DLN adapter and one or more other masters may generate a START condition within the minimum hold time, which results in a valid START condition on the bus. Arbitration is then required to determine which master will complete its transmission.

Arbitration proceeds bit by bit. During every bit, while SCL is HIGH, each master checks to see if the SDA level matches what it has sent. If at least one master outputs LOW, the SDA line will have the LOW level. If a master changes the state of the SDA line to HIGH, but the line stays in LOW, then this indicates that this master lost arbitration and it needs to back off.

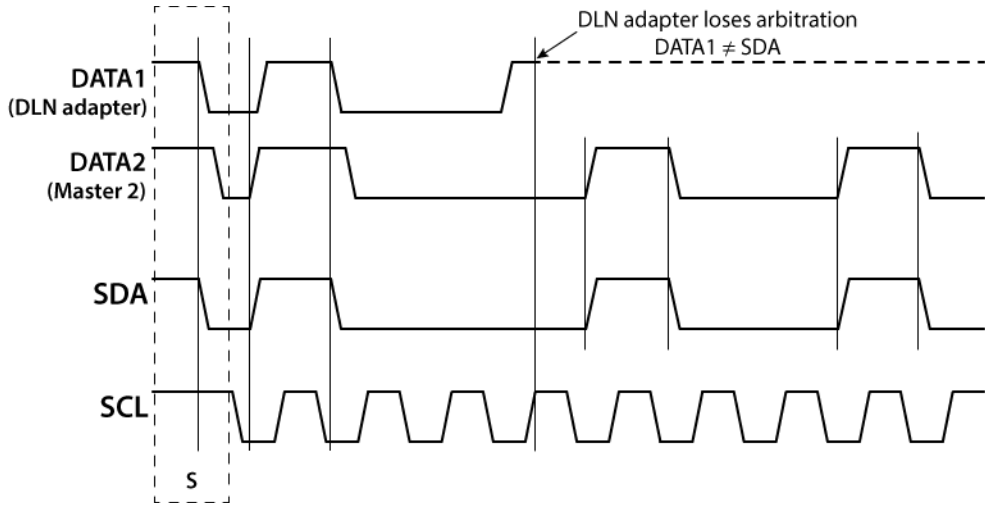
The arbitration process may take many bits. More than one masters can even complete an entire transaction without error if their transmissions are identical.

A master that loses the arbitration can generate clock pulses until the end of the byte in which it loses the arbitration and can restart its transaction when the bus is free.

If a master can act as a slave and it loses arbitration during the addressing stage, it must switch immediately to its slave mode because the winning master may try to address it.

Example

The following figure shows the arbitration procedure for the DLN adapter and the Master2 device. The moment when there is a difference between the DATA1 level and the actual level on the SDA line, the DLN adapter switches off the DATA1 output.



4.1.6 Clock Stretching

In an I^2C communication, a master device determines the clock speed. The I^2C bus provides an explicit clock signal that relieves a master and a slave from synchronizing exactly to a predefined baud rate.

However, some slave devices may receive or transmit bytes of data at a fast rate, but need more time to store a received byte or prepare another byte to be transmitted. Slaves can then hold the SCL line LOW to force the master into a wait-state until the slave is ready for the next byte transmission. This mechanism is called clock stretching.

An I^{2}C slave is allowed to hold the SCL line LOW if it needs to reduce the bus speed. The master on the other hand is required to read back the SCL signal after releasing it to the HIGH state and wait until the SCL line has actually gone HIGH. DLN-series adapters support clock stretching.

Taking into consideration the impacts of clock stretching, the total speed of the I^{2}C bus might be significantly decreased.

4.2 I2C Master Interface

All DLN-series adapters support I^{2}C master interface. Some of them can have several independent I^{2}C ports. To know the number of available I^{2}C master ports, use the `DlnI2cMasterGetPortCount()` function.

Before using the I^{2}C bus for transmitting data, you need to configure the I^{2}C master port and enable it (see Configuring the I2C Master Interface). To stop using the I^{2}C master port, you can disable it by the `DlnI2cMasterDisable()` function.

4.2.1 Configuring the I2C Master Interface

To start using the I^{2}C master port, you need to configure the I^{2}C master interface:

1. Configure the I^{2}C frequency. This parameter influences the speed of data transmission. For details, read I2C Speed and Frequency.
2. Configure the number of attempts to resend data if Not Acknowledgement is received. For details, read Reply Count.
3. Enable the I^{2}C master port. If the pins of the I^{2}C port are not used by other modules, you can enable the I^{2}C master port by the `DlnI2cMasterEnable()` function.

4.2.2 I2C Speed and Frequency

I^{2}C bus specification describes four operating speed categories for bidirectional data transmission:

Standard-mode (Sm)	a bit rate up to 100 kbit/s
Fast-mode (Fm)	a bit rate up to 400 kbit/s
Fast-mode Plus (Fm+)	a bit rate up to 1 Mbit/s
High-speed mode (Hs)	a bit rate up to 3.4 Mbit/s

One more speed category, Ultra-fast mode (UFm), stands for unidirectional data transmission up to 5 Mbit/s.

Configuring the I^{2}C master interface, you can specify the frequency value by calling the `DlnI2cMasterSetFrequency()` function.

The range of supported frequency values depends on the DLN adapter:

- DLN-1 and DLN-2 adapters support frequency from 1kHz up to 4MHz.

- DLN-4 adapters support frequency from 1.47kHz up to 1MHz.

The quality of I^2C lines, the values of pull-up resistors and the number of slaves connected to the I^2C bus may influence the working frequency of the I^2C bus. Besides, the frequency reflects the speed of a single byte transmission, but not the speed of transmitting all data. It is a fact that the time of data processing can exceed significantly the time of data transmission. That is why data transmitted at high speed can have no effect on the speed of the I^2C bus if delays between bytes are longer than the bytes themselves.

4.2.3 Reply Count

The I^2C transmission expects an Acknowledge bit after every byte. This bit is sent by a slave and by a receiver:

- A slave sends an Acknowledge bit after the slave address and direction bit to signal that the device with the specified address is present on the I^2C bus;
- A receiver sends an Acknowledge bit after a data byte to signal that the byte was successfully received and another byte may be sent.

The Acknowledge signal is LOW on the SDA line that remains stable during the HIGH period of the ninth pulse on the SCL line. If the SDA line remains HIGH during this clock pulse, this is defined as Not Acknowledge signal. In this case, the master has the following options:

- Generate a STOP (P) condition to abort the transmission;
- Generate a repeated START (Sr) condition to start a new transmission.

DLN-1 and DLN-2 adapters provide one more option for the I^2C master:

- Generate a STOP (P) condition followed by a START (S) condition to start the same transmission from the very beginning.

This option allows to repeat transmission if acknowledgement was not received. By default, transmissions can repeat 10 times. If all these times acknowledgement was not received, the transmission is supposed to fail. If acknowledgement was received, the transmission is successful.

Using the `DlnI2cMasterSetMaxReplyCount()` function, you can change the maximum number of attempts to transmit data. The `DlnI2cMasterGetMaxReplyCount()` function allows to check the currently specified number of attempts.

Note: This option is available only for DLN-1 and DLN-2 adapters.

4.2.4 I2C Addresses

DLN-series adapters support only 7-bit addressing. To start transmission, the I^{2}C master generates the START (S) condition followed by seven bits of a slave address and an eighth bit which is a data direction bit.

Note: Some vendors provide 8-bit addresses for their devices. Actually, this is the 7-bit slave address and the direction bit. It is important to use only top 7 bits as the slave address and to discard the least significant bit of such address.

7-bit addressing allows 127 different addresses. Some addresses are reserved (See Slave Address and Data Direction), only 112 devices can actually be connected to the I^{2}C bus. To scan all possible addresses and to find devices connected to the I^{2}C bus, us the **`DlnI2cMasterScanDevices()`** function. It returns the number of connected devices and the list of their addresses.

You can use these addresses for I^{2}C transmission in one of the following functions:

`DlnI2cMasterRead()`
: Receives data from the specified slave. Internal address can be specified (See READ Operation for details).

`DlnI2cMasterWrite()`
: Sends data to the specified slave. Internal address can be specified (See WRITE Operation for details).

`DlnI2cMasterTransfer()`
: Sends data to the specified slave, then reads data from the same slave (only DLN-1 and DLN-2 adapters support this function).

4.2.5 Simple I2C Master Module Example

The following example shows how to operate with I2C master module. You can find the complete example in the “..*\Program Files\Diolan\DLN\examples\c_cpp\examples\simple*” folder after DLN setup package installation.

```
#include "..\..\..\common\dln_generic.h"
#include "..\..\..\common\dln_i2c_master.h"
#pragma comment(lib, "..\\..\\..\\bin\\dln.lib")


int _tmain(int argc, _TCHAR* argv[])
{
// Open device
HDLN device;
DlnOpenUsbDevice(&device);

// Set frequency
uint32_t frequency;
DlnI2cMasterSetFrequency(device, 0, 100000, &frequency);
// Enable I2C master
uint16_t conflict;
DlnI2cMasterEnable(device, 0, &conflict);
18
19  // Prepare output buffer
20  uint8_t output[8], input[8];
21  for (int i = 0; i < 8; i++) output[i] = i;
22  // Write bytes
23  DlnI2cMasterWrite(device, 0, 0x50, 1, 0, 8, output);
24
25  // Read bytes
26  DlnI2cMasterRead(device, 0, 0x50, 1, 0, 8, input);
27  // Print input data
28  for (int i = 0; i < 8; i++) printf("%02x ", input[i]);
29
30  // Disable I2C master
31  DlnI2cMasterDisable(device, 0);
32  // Close device
33  DlnCloseHandle(device);
34  return 0;
35  }
```

Line 1: `#include "..\..\..\common\dln_generic.h"`

The *dln_generic..h* header file declares functions and data structures for the generic interface.

Line 2: `#include "..\..\..\common\dln_i2c_master.h"`

The *dln_i2c_master.h* header file declares functions and data structures for the I2C master interface.

Line 3: `#pragma comment(lib, "..\\..\\..\\bin\\dln.lib")`

Use *dln.lib* library while project linking.

Line 10: `DlnOpenUsbDevice(&device);`

The function establishes the connection with the DLN adapter. This application uses the USB connectivity of the adapter. For additional options, refer to the Device Opening & Identification section.

Line 14: `DlnI2cMasterSetFrequency(device, 0, 100000, &frequency);`

This function sets frequency on the I2C bus. It is set in Hertz. Any frequency value can be provided to the function, but only device compatible frequency will be set. You can read the actual frequency value by providing pointer to the unsigned 32-bit integer variable. You can read more about I2C bus speed and frequency by navigating to I2C Speed and Frequency section.

Line 17: `DlnI2cMasterEnable(device, 0, &conflict);`

This function enables I2C master module.

Line 21: `for (int i = 0; i < 8; i++) output[i] = i;`

Fill output array with the values from 0 to 8. It will be used as data buffer for sending it via I2C bus.

Line 23: DlnI2cMasterWrite(device, 0, 0x50, 1, 0, 8, output);

This function sends provided data buffer via I2C bus to connected I2C slave device. To send data properly to slave device it is required to provide also slave device address and memory address. You can read more about I2C addressing at I2C Addresses.

Line 26: DlnI2cMasterRead(device, 0, 0x50, 1, 0, 8, &input);

This function reads data from the I2C slave device. The parameters are almost similar to the data writing process.

Line 28: for (int i = 0; i < 8; i++) printf("%02x ", input[i]);

Print to console data, which was read from the I2C slave device.

Line 31: DlnI2cMasterDisable(device, 0);

Disable I2C master port.

Line 33: DlnCloseHandle(device);

Closing handle to the previously opened DLN-series adapter.

4.2.6 I2C Master Functions

Use the I^{2}C Master Interface functions to control and monitor the I^{2}C Master module of a DLN-series adapter. The *dln_i2c_master.h* file declares the I^{2}C Master Interface functions.

General port information:

`DlnI2cMasterGetPortCount()`
Retrieves the total number of I^{2}C master ports available at your DLN-series adapter.

`DlnI2cMasterEnable()`
Assigns a port to the I^{2}C Master module.

`DlnI2cMasterDisable()`
Releases a port from the I^{2}C Master module.

`DlnI2cMasterIsEnabled()`
Retrieves whether a port is assigned to the I^{2}C Master module.

`DlnI2cMasterScanDevices()`
Scans all slave addresses searching for connected I^{2}C slave devices.

I^{2}C Master module configuration functions:

`DlnI2cMasterSetFrequency()`
Configures frequency for the specified I^{2}C master port.

`DlnI2cMasterGetFrequency()`
Retrieves frequency configuration for an I^{2}C Master port.

`DlnI2cMasterSetMaxReplyCount()`
Configures the maximum reply count for an I^{2}C master port.

`DlnI2cMasterGetMaxReplyCount()`
Retrieves the maximum reply count configuration.

Transmission functions:

`DlnI2cMasterRead()`

Receives data from the specified slave. Internal address can be specified.

`DlnI2cMasterWrite()`

Sends data to the specified slave. Internal address can be specified.

`DlnI2cMasterTransfer()`

Sends data to the specified slave, then reads data from the same slave (only DLN-1 and DLN-2 adapters support this function).

DlnI2cMasterGetPortCount() Function

The **`DlnI2cMasterGetPortCount()`** function retrieves the total number of I^2C master ports available at your DLN-series adapter.

Syntax

```
DLN_RESULT DlnI2cMasterGetPortCount(
   HDLN handle,
   uint8_t* count
);
```

Parameters

handle

A handle to the DLN-series adapter.

count

A pointer to an unsigned 8-bit integer that receives the number of available I^2C master ports.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function retrieved the number of available I^2C master ports successfully.

DLN_RES_HARDWARE_NOT_FOUND (0x81)

The handle parameter is invalid, or the corresponding DLN adapter was disconnected.

Remarks

The **`DlnI2cMasterGetPortCount()`** function is defined in the *dln_i2c_master.h* file.

DlnI2cMasterEnable() Function

The **`DlnI2cMasterEnable()`** function assigns the specified port to the I^2C Master module.

Syntax

```
DLN_RESULT DlnI2cMasterEnable(
   HDLN handle,
   uint8_t port,
   uint16_t* conflict
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the I^2C master port.

conflict

A pointer to an unsigned 16-bit integer that receives a number of the conflicted pin, if any.

A conflict arises if a pin is already assigned to another module of the DLN adapter and cannot be used by the I^2C master module. To fix this, check which module uses the pin (call the **`DlnGetPinCfg()`** function), disconnect the pin from that module and call the **`DlnI2cMasterEnable()`** function once again. If there are any more conflicting pins, the next conflicted pin number will be returned.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function activated the I^2C master port successfully.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnI2cMasterGetPortCount()`** function to find the maximum possible port number.

DLN_RES_PIN_IN_USE (0xA5)

The port cannot be activated as the I^2C master port because one or more pins of the port are assigned to another module. The **conflict** parameter contains the number of a conflicting pin.

Remarks

The **`DlnI2cMasterEnable()`** function is defined in the *dln_i2c_master.h* file.

DlnI2cMasterDisable() Function

The **`DlnI2cMasterDisable()`** function releases the specified port from the I^2C Master module.

Syntax

```
DLN_RESULT DlnI2cMasterDisable(
   HDLN handle,
   uint8_t port
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the I^{2}C master port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function deactivated the I^{2}C master port successfully.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnI2cMasterGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnI2cMasterDisable()`** function is defined in the *dln_i2c_master.h* file.

DlnI2cMasterIsEnabled() Function

The **`DlnI2cMasterIsEnabled()`** function checks whether the specified I^{2}C master port is active or not.

Syntax

```
DLN_RESULT DlnI2cMasterIsEnabled(
   HDLN handle,
   uint8_t port,
   uint8_t* enabled
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the I2C master port.

enabled

A pointer to an unsigned 8-bit integer that receives information whether the specified I^{2}C master port is activated. There are two possible values:

- **`DLN_I2C_MASTER_DISABLED`** (0) - The port is not configured as an I^{2}C master.

- `DLN_I2C_MASTER_ENABLED` (1) - The port is configured as an I^{2}C master.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function retrieved the state of the I^{2}C master port successfully.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnI2cMasterGetPortCount()` function to find the maximum possible port number.

Remarks

The `DlnI2cMasterIsEnabled()` function is defined in the *dln_i2c_master.h* file.

DlnI2cMasterSetFrequency() Function

The `DlnI2cMasterSetFrequency()` function configures the clock frequency for the specified I^{2}C port.

Syntax

```
DLN_RESULT DlnI2cMasterSetFrequency(
   HDLN handle,
   uint8_t port,
   uint32_t frequency,
   uint32_t* actualFrequency
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the I^{2}C master port.

frequency

I^{2}C frequency value, specified in Hz. You may specify any value within the range, supported by your DLN adapter. This range can be retrieved using the respective function. If you enter an incompatible value, it will be approximated as the closest lower frequency value, supported by the adapter.

actualFrequency

A pointer to an unsigned 32-bit integer that receives the actual frequency applied by this function. The frequency is specified in Hz. If the frequency specified in frequency parameter is supported, the actual frequency will be equal to it. Otherwise, the closest lower value will be applied. If NULL is specified in this parameter, the actual frequency value will not be returned. You can still use the `DlnI2cMasterGetFrequency()` function to check the actual frequency.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function set the I^2C master port clock frequency value successfully.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnI2cMasterGetPortCount()`** function to find the maximum possible port number.

DLN_RES_BUSY (0xB6)

The function cannot change frequency while the I^2C master port is busy transferring.

DLN_RES_VALUE_ROUNDED (0x21)

The function approximated the frequency value to the closest supported value.

Remarks

The **`DlnI2cMasterSetFrequency()`** function is defined in the *dln_i2c_master.h* file.

DlnI2cMasterGetFrequency() Function

The **`DlnI2cMasterGetFrequency()`** function retrieves the current I^2C bus clock frequency.

Syntax

```
DLN_RESULT DlnI2cMasterGetFrequency(
   HDLN handle,
   uint8_t port,
   uint32_t* frequency
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the I2C master port.

frequency

A pointer to an unsigned 32-bit integer that receives the current I^2C bus clock frequency in Hz.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function retrieved the current clock frequency value successfully.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnI2cMasterGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnI2cMasterGetFrequency()`** function is defined in the *dln_i2c_master.h* file.

DlnI2cMasterSetMaxReplyCount() Function

The **`DlnI2cMasterSetMaxReplyCount()`** function sets maximum reply count for I^{2}C master port.

DLN-1 and DLN-2 adapters cannot send a single slave address and direction bit without data bytes. This is a firmware driver limitation. Therefore, an adapter cannot read a single byte at all possible addresses. Retrying every read/write operation 10 times is set by default in firmware I^{2}C driver, but using the **`DlnI2cMasterSetMaxReplyCount()`** function you can modify this parameter.

Attention: This function can be used only for DLN-1 and DLN-2 adapters.

Syntax

```
DLN_RESULT DlnI2cMasterSetMaxReplyCount(
    HDLN handle,
    uint8_t port,
    uint16_t maxReplyCount
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the I2C master port.

maxReplyCount

Maximum reply count value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function set the maximum reply count successfully.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnI2cMasterGetPortCount()`** function to find the maximum possible port number.

DLN_RES_BUSY (0xB6)

The function cannot change the maximum reply count while the I^{2}C master port is busy transferring.

DLN_RES_INVALID_VALUE (0xE2)

The provided value is zero or not valid.

Remarks

The **`DlnI2cMasterSetMaxReplyCount()`** function is defined in *dln_i2c_master.h* file.

DlnI2cMasterGetMaxReplyCount() Function

The **`DlnI2cMasterGetMaxReplyCount()`** function retrieves maximum reply count for I^{2}C master port.

Attention: This function can be used only for DLN-1 and DLN-2 adapters.

Syntax

```
DLN_RESULT DlnI2cMasterGetMaxReplyCount(
    HDLN handle,
    uint8_t port,
    uint16_t *maxReplyCount
);
```

Parameters

handle

A handle to the DLN adapter.

port

A number of the I^{2}C master port.

maxReplyCount

Pointer to the variable that receives the current maximum reply count value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function set the maximum reply count successfully.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnI2cMasterGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnI2cMasterGetMaxReplyCount()`** function is defined in the *dln_i2c_master.h* file.

DlnI2cMasterScanDevices() Function

The **`DlnI2cMasterScanDevices()`** function scans all the 127 slave addresses searching for connected I^{2}C slave devices.

Syntax

```
DLN_RESULT DlnI2cMasterScanDevices(
   HDLN handle,
   uint8_t port,
   uint8_t* addressCount,
   uint8_t* addressList
);
```

Parameters

handle

A handle to the DLN adapter.

port

A number of the I^{2}C master port scan for I^{2}C slave devices.

addressCount

A pointer to an unsigned 8-bit integer that receives the number of found I^{2}C slave devices.

addressList

A pointer to an array of unsigned 8-bit integers that receives the addresses of found I^{2}C slave devices. I^{2}C address is 7-bit long, so the most significant bit of the received integers will always be equal to zero. Some vendors specify the 8-bit addresses in their documentation. If you need to convert a 8-bit I^{2}C address to 7-bit address, take a look at I^{2}C address article.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function scanned the I^{2}C addresses and found all connected I^{2}C slaves successfully.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnI2cMasterGetPortCount()` function to find the maximum possible port number.

DLN_RES_BUSY (0xB6)

The function cannot scan I^{2}C addresses while the I^{2}C master port is busy transferring.

DLN_RES_DISABLED (0xB7)

The function cannot scan I^{2}C addresses because the specified I^{2}C master port is not active. Use the `DlnI2cMasterEnable()` function to activate the I^{2}C master port.

DLN_RES_OPERATION_TIMEOUT

The function failed to scan all slave addresses.

Remarks

The `DlnI2cMasterScanDevices()` function is defined in the *dln_i2c_master.h* file.

DlnI2cMasterRead() Function

The `DlnI2cMasterRead()` function reads data from the specified I^{2}C slave device.

Syntax

```
DLN_RESULT DlnI2cMasterRead(
   HDLN handle,
   uint8_t port,
   uint8_t slaveDeviceAddress,
   uint8_t memoryAddressLength,
   uint32_t memoryAddress,
   uint16_t bufferLength,
   uint8_t* buffer
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the I^{2}C master port.

slaveDeviceAddress

A 7-bit number, assigned to each I^{2}C slave device. For additional details refer to the I^{2}C slave address section.

memoryAddressLength

An internal address length. If set to zero, no internal address is sent.

memoryAddress

An internal I^{2}C slave device address. For details, read Using Internal Addresses.

bufferLength

The size of the message buffer (in the range from 1 to 256 bytes).

buffer

A pointer to an array of unsigned 8-bit integers that receives data from the I^{2}C slave device during the function execution. The array must contain at least **bufferLength** elements.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function executed the I^{2}C read operation successfully.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnI2cMasterGetPortCount()` function to find the maximum possible port number.

DLN_RES_DISABLED (0xB7)

The function cannot execute the I^{2}C read operation because the specified I^{2}C master port is not active. Use the `DlnI2cMasterEnable()` function to activate the I^{2}C master port.

Remarks

The `DlnI2cMasterRead()` function is defined in the *dln_i2c_master.h* file.

DlnI2cMasterWrite() Function

The `DlnI2cMasterWrite()` function sends data to the specified I^{2}C slave device.

Syntax

```
DLN_RESULT DlnI2cMasterWrite(
   HDLN handle,
   uint8_t port,
   uint8_t slaveDeviceAddress,
   uint8_t memoryAddressLength,
   uint32_t memoryAddress,
   uint16_t bufferLength,
   uint8_t* buffer
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the I2C master port to be used.

slaveDeviceAddress

A 7-bit number, uniquely assigned to each I^{2}C slave device. See I2C address section for additional information.

memoryAddressLength

An internal address length. If set to zero, no internal address is sent.

memoryAddress

An internal I^{2}C slave device address. For details, read Using Internal Addresses.

bufferLength

The size of the message buffer in bytes. This value must fall within a range from 1 to 256.

buffer

A pointer to an array of unsigned 8-bit integers that receives information to be sent to a slave during the function execution. The buffer size must not exceed 256 bytes.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function executed the I^{2}C write operation successfully.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnI2cMasterGetPortCount()` function to find the maximum possible port number.

DLN_RES_DISABLED (0xB7)

The function cannot execute the I^{2}C write operation because the specified I^{2}C master port is not active. Use the `DlnI2cMasterEnable()` function to activate the I^{2}C master port.

Remarks

The **DlnI2cMasterWrite()** function is defined in the *dln_i2c_master.h* file.

DlnI2cMasterTransfer() Function

The **DlnI2cMasterTransfer()** function sends and receives data via the I^{2}C bus. The data is sent and received as an array of 1-byte elements.

Syntax

```
DLN_RESULT DlnI2cMasterTransfer(
    HDLN handle,
    uint8_t port,
    uint8_t slaveDeviceAddress,
    uint16_t writeLength,
    uint16_t readLength,
    uint8_t *writeBuffer,
    uint8_t *readBuffer
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the I^{2}C master port.

slaveDeviceAddress

The I^{2}C slave device address value.

writeLength

The length of data that should be written to the slave device.

readLength

The length of data that should be read from the slave device.

writeBuffer

A pointer to an array of unsigned 8-bit integers that receives data to be sent to the slave during the function execution.

readBuffer

A pointer to an array of unsigned 8-bit integers that receives data from the slave during the function execution.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function transmitted data via the I^{2}C bus successfully.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **DlnI2cMasterGetPortCount()** function to find the

maximum possible port number.

DLN_RES_DISABLED (0xB7)

The function transmit data via the I^{2}C bus because the specified I^{2}C master port is not active. Use the `DlnI2cMasterEnable()` function to activate the I^{2}C master port.

DLN_RES_INVALID_BUFFER_SIZE (0xAE)

The specified buffer size is not valid.

Remarks

Note: DLN-4 adapters do not support this function.

The `DlnI2cMasterTransfer()` function is defined in the *dln_i2c_master.h* file.

4.3 I2C Slave Interface

Some DLN-series adapters support I^{2}C slave interface.

Before you activate the I^{2}C slave port, you need to configure it (See Configuring I2C Slave Interface).

You can configure events to be informed when an I^{2}C master receives data from your slave and/or transmits data to it (See I2C Slave Events).

4.3.1 Configuring I2C Slave Interface

To provide the I^{2}C communication, you need to configure the I^{2}C slave port:

1. Specify the I^{2}C slave address of your device. DLN adapters support 7-bit addressing. Some DLN adapters allow to specify several I^{2}C slave addresses. For details, read I2C Slave Addressing.

2. Configure general call support. I^{2}C slave can ignore or acknowledge the general call addressing when data can be transmitted to all I^{2}C slaves simultaneously. For details, read General Call Support.

3. Configure generating events. Events can be generated when an I^{2}C master initiates data transmission. If you do not need these notifications, cancel generating events. Read I2C Slave Events.

After you have finished configuring the I^{2}C slave device, enable the I^{2}C slave port by the `DlnI2cSlaveEnable()` function.

4.3.2 I2C Slave Addressing

DLN adapters support only 7-bit addressing. To assign a I^{2}C slave address to your device, use the `DlnI2cSlaveSetAddress()` function. This function does not prevent you from assigning reserved addresses to your DLN adapter. For more information about reserved addresses, read Reserved I2C Slave Addresses.

Some DLN adapters can support more than one I^{2}C slave addresses simultaneously. To check how many I^{2}C slave addresses are supported by your DLN adapter, use the `DlnI2cSlaveGetAddressCount()` function. To assign several I^{2}C slave addresses to your DLN adapter, use the `DlnI2cSlaveSetAddress()` function for every address. In the function, you specify the **slaveAddressNumber** parameter; its value should be unique for every I^{2}C slave address but should not exceed the number of supported slave addresses.

To check an I^{2}C slave address assigned to your device, call the `DlnI2cSlaveGetAddress()` function and point the desired value of the **slaveAddressNumber** parameter. To check all assigned I^{2}C slave addresses, call the `DlnI2cSlaveGetAddress()` function for every possible **slaveAddressNumber** value.

General Call Support

I^{2}C bus allows to transmit data to all I^{2}C slaves simultaneously. This option is called General Call. To make a general call, the I^{2}C master generates the following address: 0000 000 followed by the Write (0) direction bit. The General Call address is one of the reserved addresses and cannot be assigned to any I^{2}C slave device.

When an I^{2}C slave receives the general call address, it can acknowledge it to receive transmitted data or ignore it.

You can configure the I^{2}C slave's behavior when it receives the general call address:

- If you want your DLN I^{2}C slave port to acknowledge the general call addressing, call the `DlnI2cSlaveGeneralCallEnable()` function.
- If you want your DLN I^{2}C slave port to ignore the general call addressing, call the `DlnI2cSlaveGeneralCallDisable()` function.

To check the current configuration of the general call support, use the `DlnI2cSlaveGeneralCallIsEnabled()` function.

4.3.3 I2C Slave Events

There are two ways to detect an I^{2}C transmission:

- To observe I^{2}C lines permanently. This consumes much CPU time. Besides, the more times the device polls the I^{2}C bus, the less time it can spend carrying out its intended function. That is why such devices are slow.
- To receive events about the requests from the I^{2}C bus. You can configure event generation when the I^{2}C master addresses for receiving or transmitting data.

To configure events, use the `DlnI2cSlaveSetEvent()` function. You need to specify the **slaveAddressNumber** and **eventType** parameters. The `DlnI2cSlaveGetSupportedEventTypes()` function returns the list of event types available for the I2C slave port.

The **eventType** parameter can have one of the following values:

I2C Events Types

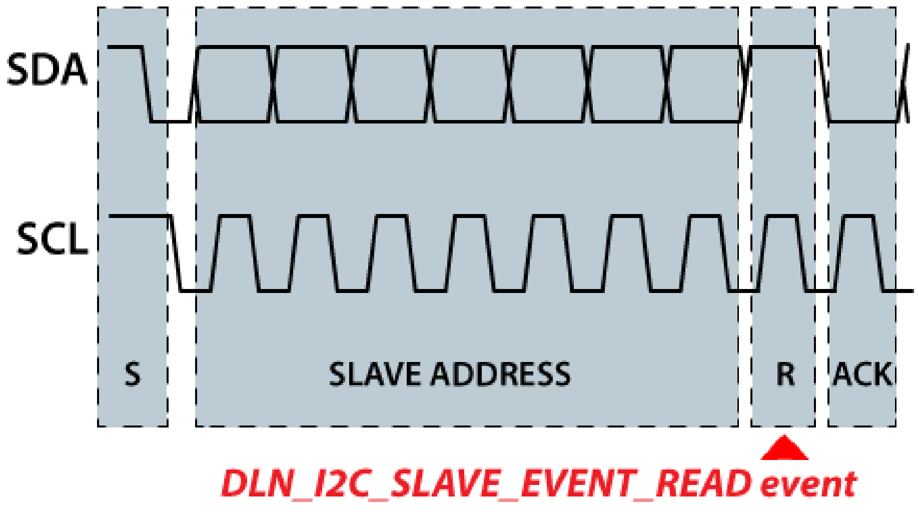
`DLN_I2C_SLAVE_EVENT_NONE`	A DLN adapter does not generate any I^{2}C events.
`DLN_I2C_SLAVE_EVENT_READ`	A DLN adapter generates events when the I^{2}C master device initiates receiving data from the I^{2}C slave address assigned to the DLN adapter. The `DLN_I2C_SLAVE_READ_EV` structure describes the event details. Read DLN_I2C_SLAVE_EVENT_READ Events
`DLN_I2C_SLAVE_EVENT_WRITE`	A DLN adapter generates events when the I^{2}C master device initiates transmitting data to the I^{2}C slave address assigned to the DLN adapter. The `DLN_I2C_SLAVE_WRITE_EV` structure describes the event details. Read DLN_I2C_SLAVE_EVENT_WRITE Events
`DLN_I2C_SLAVE_EVENT_READ_WRITE`	A DLN adapter generates events when the I^{2}C master device initiates receiving data from or transmitting data to the I^{2}C slave address assigned to the DLN adapter. The `DLN_I2C_SLAVE_READ_EV` structure describes the read event details. The `DLN_I2C_SLAVE_WRITE_EV` structure describes the write event details. Read DLN_I2C_SLAVE_EVENT_READ_WRITE Events

By default, event generation is disabled for all I^{2}C slave addresses (the **eventType** parameter is set to `DLN_I2C_SLAVE_EVENT_NONE`).

If your DLN adapter uses more than one I^{2}C slave address, you can specify different event configuration for each I^{2}C slave address.

DLN_I2C_SLAVE_EVENT_READ Events

A DLN adapter generates the `DLN_I2C_SLAVE_EVENT_READ` events each time the I^{2}C master device initiates receiving data from the I^{2}C slave address assigned to the DLN adapter.

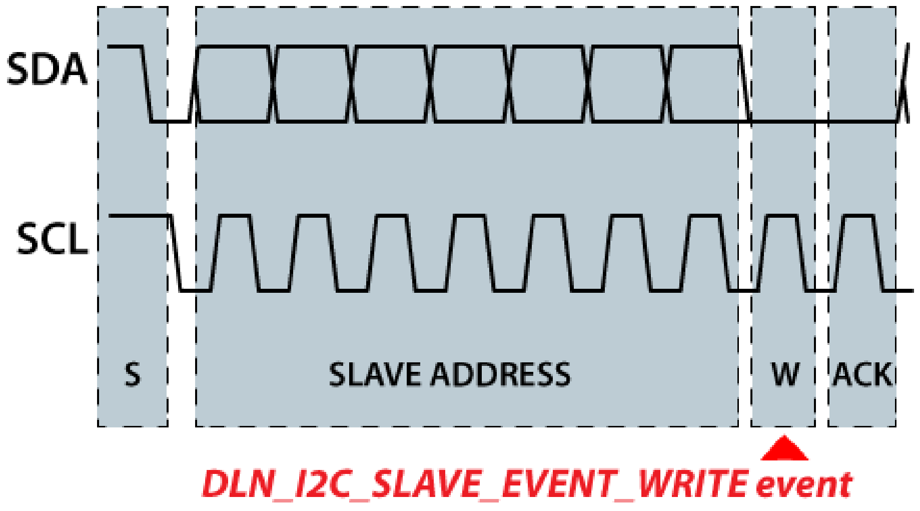


Use the `DlnI2cSlaveSetEvent()` function to configure events. Pass `DLN_I2C_SLAVE_EVENT_READ` for the **eventType** parameter.

The `DLN_I2C_SLAVE_READ_EV` structure describes the event details: event counter, I^{2}C slave address and port number, and the buffer size. The header of the structure contains the **msgId** field that is set to `DLN_MSG_ID_I2C_SLAVE_READ_EV (0x0C10)`.

DLN_I2C_SLAVE_EVENT_WRITE Events

A DLN adapter generates the `DLN_I2C_SLAVE_EVENT_WRITE` events each time the I^{2}C master device initiates transmitting data to the I^{2}C slave address assigned to the DLN adapter.

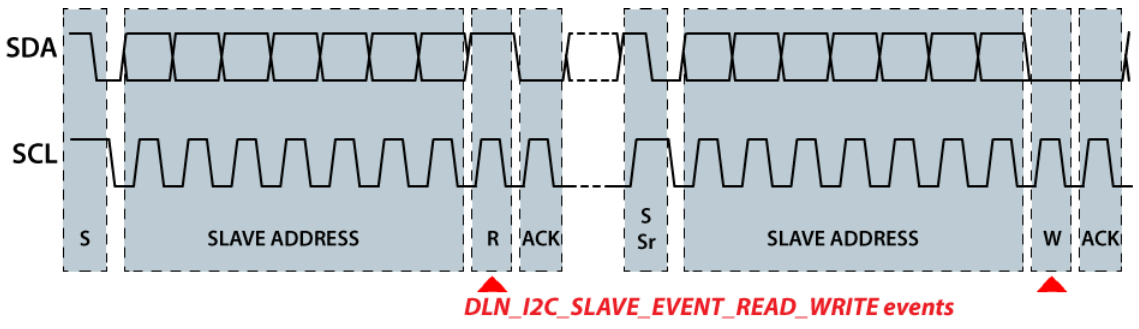


Use the `DlnI2cSlaveSetEvent()` function to configure events. Pass `DLN_I2C_SLAVE_EVENT_WRITE` for the **eventType** parameter.

The DLN_I2C_SLAVE_WRITE_EV structure describes the event details: event counter, I^{2}C slave address and port number, the buffer size and the received data. The header of the structure contains the **msgId** field that is set to `DLN_MSG_ID_I2C_SLAVE_WRITE_EV (0x0C11)`.

DLN_I2C_SLAVE_EVENT_READ_WRITE Events

A DLN adapter generates the DLN_I2C_SLAVE_EVENT_READ_WRITE events each time the I^{2}C master device initiates receiving data from or transmitting data to the I^{2}C slave address assigned to the DLN adapter.



Use the `DlnI2cSlaveSetEvent()` function to configure events. Pass DLN_I2C_SLAVE_EVENT_READ_WRITE for the **eventType** parameter.

The DLN_I2C_SLAVE_EVENT_READ_WRITE events are described by two structures:

- The DLN_I2C_SLAVE_READ_EV structure describes the I^{2}C read event details: event counter, I^{2}C slave address and port number, and the buffer size. The header of the structure contains the **msgId** field that is set to `DLN_MSG_ID_I2C_SLAVE_READ_EV (0x0C10)`.
- The DLN_I2C_SLAVE_WRITE_EV structure describes the I^{2}C write event details: event counter, I^{2}C slave address and port number, the buffer size and the received data. The header of the structure contains the **msgId** field that is set to `DLN_MSG_ID_I2C_SLAVE_WRITE_EV (0x0C11)`.

4.3.4 I2C Slave Functions

Use the I^{2}C Slave Interface functions to control and monitor the I^{2}C Slave module of a DLN-series adapter. The *dln_i2c_slave.h* file declares the I^{2}C Slave Interface functions.

General port information:

`DlnI2cSlaveGetPortCount()`
: Retrieves the total number of I^{2}C slave ports available at your DLN-series adapter.

`DlnI2cSlaveEnable()`
: Assigns a port to the I^{2}C Slave module.

`DlnI2cSlaveDisable()`
: Releases a port from the I^{2}C Slave module.

`DlnI2cSlaveIsEnabled()`
: Retrieves whether a port is assigned to the I^{2}C Slave module.

DlnI2cSlaveLoadReply()
: Loads data to be transmitted to an I^{2}C master device.

I^{2}C Slave module configuration functions:

DlnI2cSlaveGeneralCallEnable()
: Activates I^{2}C general call support.

DlnI2cSlaveGeneralCallDisable()
: Disables I^{2}C general call support.

DlnI2cSlaveGeneralCallIsEnabled()
: Retrieves whether I^{2}C general call support is activated.

DlnI2cSlaveGetAddressCount()
: Retrieves the number of I^{2}C slave addresses supported by the DLN adapter.

DlnI2cSlaveSetAddress()
: Assigns an I^{2}C slave address to the specified I^{2}C slave module.

DlnI2cSlaveGetAddress()
: Retrieves one of the I^{2}C slave addresses assigned to the specified I^{2}C slave module.

I^{2}C Slave event functions:

DlnI2cSlaveSetEvent()
: Configures event generation for an I^{2}C slave port.

DlnI2cSlaveGetEvent()
: Retrieves event generation configuration for an I^{2}C slave port.

DlnI2cSlaveGetSupportedEventTypes()
: Retrieves the list of event types available for an I^{2}C slave port.

DlnI2cSlaveGetPortCount() Function

The **DlnI2cSlaveGetPortCount()** function retrieves the total number of I^{2}C slave ports available in your DLN-series adapter.

Syntax

```
DLN_RESULT DlnI2cSlaveGetPortCount(
   HDLN handle,
   uint8_t* count
);
```

Parameters

handle

A handle to the DLN-series adapter.

count

A pointer to an unsigned 8-bit integer that receives the number of available I2C slave ports.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the number of I^{2}C slave ports.

Remarks

The **`DlnI2cSlaveGetPortCount()`** function is defined in the *dln_i2c_slave.h* file.

DlnI2cSlaveEnable() Function

The **`DlnI2cSlaveEnable()`** function activates the specified I^{2}C slave port at your DLN-series adapter.

Syntax

```
DLN_RESULT DlnI2cSlaveEnable(
   HDLN handle,
   uint8_t port,
   uint16_t* conflict
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the I^{2}C slave port.

conflict

A number of the pin that is currently occupied by another module. To fix this, check which module uses the pin (call the **`DlnGetPinCfg()`** function), disconnect the pin from that module and call the **`DlnI2cSlaveEnable()`** function once again.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully activated the I^{2}C slave port.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnI2cSlaveGetPortCount()`** function to find the maximum possible port number.

DLN_RES_PIN_IN_USE (0xA5)

At least one pin of the port is assigned to another module. The **conflict** parameter contains the number of the conflicting pin.

DLN_RES_I2C_SLAVE_ADDRESS_NEEDED (0xBE)

The I^{2}C slave address is not specified for this device. Use the **`DlnI2cSlaveSetAddress()`** function to specify the address.

Remarks

The I^{2}C bus interface requires two lines for data transmission and synchronization (SDA and SCL). If any of these lines is used by another module, the DLN adapter cannot activate the I^{2}C slave port. The **`DlnI2cSlaveEnable()`** function returns only one conflicting pin. If both SCL and SDA lines are in use, you need to call this function three times: two times to detect both conflicting pins and the third time to enable the I^{2}C slave port.

The **`DlnI2cSlaveEnable()`** function is defined in the *dln_i2c_slave.h* file.

DlnI2cSlaveDisable() Function

The **`DlnI2cSlaveDisable()`** function deactivates the specified I^{2}C slave port at your DLN-series adapter and releases the pins previously used for SDA and SCL lines.

Syntax

```
DLN_RESULT DlnI2cSlaveDisable(
   HDLN handle,
   uint8_t port,
   uint8_t waitForTransferCompletion
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the I^{2}C slave port.

waitForTransferCompletion

The parameter that contains your choice according the function behavior if the I^{2}C slave port is busy transmitting data. The following values are possible:

- **`DLN_I2C_SLAVE_WAIT_FOR_TRANSFERS(1)`** - Wait until the transmission completes.
- **`DLN_I2C_SLAVE_CANCEL_TRANSFERS(0)`** - Cancel all pending data transmissions and disable the module.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully deactivated the I^{2}C slave port.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnI2cSlaveGetPortCount()`** function to find the maximum possible port number.

DLN_RES_TRANSFER_CANCELLED (0x20)

The function cancelled the pending data transmission.

Remarks

The **`DlnI2cSlaveDisable()`** function is defined in the *dln_i2c_slave.h* file.

DlnI2cSlaveIsEnabled() Function

The **`DlnI2cSlaveIsEnabled()`** function checks whether the specified I^{2}C slave port is active or not.

Syntax

```
DLN_RESULT DlnI2cSlaveIsEnabled(
  HDLN handle,
  uint8_t port,
  uint8_t* enabled
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the I^{2}C slave port.

enabled

A pointer to an unsigned 8-bit integer that receives information whether the specified I^{2}C slave port is active or not. There are two possible values:

- **`DLN_I2C_SLAVE_DISABLED (0)`** - The I^{2}C slave port is inactive.
- **`DLN_I2C_SLAVE_ENABLED (1)`** - The I^{2}C slave port is active.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved information.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnI2cSlaveGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnI2cSlaveIsEnabled()`** function is defined in the *dln_i2c_slave.h* file.

DlnI2cSlaveGeneralCallEnable() Function

The **`DlnI2cSlaveGeneralCallEnable()`** function activates I^{2}C general call support. With general call all I^{2}C slave devices on the circuit can be addressed by sending zero as I^{2}C slave address.

Syntax

```
DLN_RESULT DlnI2cSlaveGeneralCallEnable(
   HDLN handle,
   uint8_t port,
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the I^2C slave port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully enabled the general call support.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnI2cSlaveGetPortCount()` function to find the maximum possible port number.

Remarks

The `DlnI2cSlaveGeneralCallEnable()` function is defined in the *dln_i2c_slave.h* file.

DlnI2cSlaveGeneralCallDisable() Function

The `DlnI2cSlaveGeneralCallDisable()` function disables the I^2C general call support to make this slave ignore general call addressing.

Syntax

```
DLN_RESULT DlnI2cSlaveGeneralCallDisable(
   HDLN handle,
   uint8_t port,
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the I^2C slave port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully disabled the general call support.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnI2cSlaveGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnI2cSlaveGeneralCallDisable()`** function is defined in the *dln_i2c_slave.h* file.

DlnI2cSlaveGeneralCallIsEnabled() Function

The **`DlnI2cSlaveGeneralCallIsEnabled()`** function checks whether I^{2}C general call support is enabled for the specified I^{2}C slave port.

Syntax

```
DLN_RESULT DlnI2cSlaveGeneralCallIsEnabled(
   HDLN handle,
   uint8_t port,
   uint8_t* enabled
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the I^{2}C slave port.

enabled

A pointer to an unsigned 8-bit integer that receives information whether general call support is enabled. There are two possible values:

- **`DLN_I2C_SLAVE_GENERAL_CALL_DISABLED (0)`** - The I^{2}C slave port ignores the general call address.
- **`DLN_I2C_SLAVE_GENERAL_CALL_ENABLED (1)`** - The ^{2}C slave port acknowledges the general call address and receives transmitted data.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved whether the general call support is enabled.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnI2cSlaveGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnI2cSlaveGeneralCallIsEnabled()`** function is defined in the *dln_i2c_slave.h* file.

DlnI2cSlaveGetAddressCount() Function

The `DlnI2cSlaveGetAddressCount()` function retrieves the number of I^2C slave addresses supported by the DLN adapter.

DLN-series adapters can acknowledge any I^2C slave address. The limitation is only in the amount of slave addresses to be used simultaneously. You can use the `DlnI2cSlaveSetAddress()` function to configure the I^2C slave module to acknowledge specific addresses.

Syntax

```
DLN_RESULT DlnI2cSlaveGetAddressCount(
   HDLN handle,
   uint8_t port,
   uint8_t* count
);
```

Parameters

handle

A handle to the DLN-series adapter

port

A number of the I^2C slave port.

count

A pointer to an unsigned 8-bit integer that receives the number of supported I^2C slave addresses.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the number of possible slave addresses.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnI2cSlaveGetPortCount()` function to find the maximum possible port number.

Remarks

The `DlnI2cSlaveGetAddressCount()` function is defined in the *dln_i2c_slave.h* file.

DlnI2cSlaveSetAddress() Function

The `DlnI2cSlaveSetAddress()` function assigns I^2C addresses to the specified I^2C slave module. You can assign any 7-bit address, the limitation is only in quantity of addresses that can be used simultaneously. Use the `DlnI2cSlaveGetAddressCount()` function to retrieve the number of simultaneously supported I^2C slave addresses.

Syntax

```
DLN_RESULT DlnI2cSlaveSetAddress(
   HDLN handle,
   uint8_t port,
   uint8_t slaveAddressNumber,
   uint8_t address
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

An I^{2}C slave port to be configured.

slaveAddressNumber

A number of the I^{2}C slave address to be assigned. Use the `DlnI2cSlaveGetAddressCount()` function to retrieve the number of simultaneously supported I^{2}C slave addresses. The **slaveAddressNumber** is zero based.

address

An I^{2}C slave address to be set to the specified I^{2}C slave port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully specified I^{2}C slave addresses for your DLN adapter.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnI2cSlaveGetPortCount()` function to find the maximum possible port number.

DLN_RES_BUSY (0xB6)

The function cannot specify I^{2}C slave addresses for the DLN adapter while the I^{2}C slave port is busy transmitting data.

DLN_RES_INVALID_ADDRESS (0xB4)

The I^{2}C slave address is not valid.

Remarks

The `DlnI2cSlaveSetAddress()` function is defined in the *dln_i2c_slave.h* file.

DlnI2cSlaveGetAddress() Function

The `DlnI2cSlaveGetAddress()` function retrieves one of the I^{2}C slave addresses, assigned to the specified I^{2}C slave module. The total number of simultaneously assigned addresses can be retrieved with the `DlnI2cSlaveGetAddressCount()` function.

Syntax

```
DLN_RESULT DlnI2cSlaveGetAddress(
   HDLN handle,
   uint8_t port,
   uint8_t slaveAddressNumber,
   uint8_t* address
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the I^{2}C slave port.

slaveAddressNumber

A number of the I^{2}C slave address to be retrieved. The **slaveAddressNumber** value can be in the range from 0 up to (but not including) the value returned by the `DlnI2cSlaveGetAddressCount()` function.

address

A pointer to an unsigned 8-bit integer that receives the I^{2}C slave address.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the I^{2}C slave address.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnI2cSlaveGetPortCount()` function to find the maximum possible port number.

Remarks

The **`DlnI2cSlaveGetAddress()`** function is defined in the *dln_i2c_slave.h* file.

DlnI2cSlaveLoadReply() Function

The **`DlnI2cSlaveLoadReply()`** function loads data to be transferred to an I^{2}C master device.

Syntax

```
DLN_RESULT DlnI2cSlaveLoadReply(
   HDLN handle,
   uint8_t port,
   uint16_t size,
   uint8_t* buffer
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the I^{2}C slave port.

size

A size of the data buffer to be loaded. The size is specified as a number bytes and can vary from 1 to 256.

buffer

A pointer to an array of unsigned 8-bit integers that receives data to be sent to an I^{2}C master. The size of the buffer is specified in the **size** parameter.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully loaded data to the buffer.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnI2cSlaveGetPortCount()` function to find the maximum possible port number.

DLN_RES_BUSY (0xB6)

The function cannot specify I^{2}C slave addresses for the DLN adapter while the I^{2}C slave port is busy transmitting data.

DLN_RES_INVALID_BUFFER_SIZE (0xAE)

The specified buffer size is not valid.

Remarks

The `DlnI2cSlaveLoadReply()` function is defined in the *dln_i2c_slave.h* file.

DlnI2cSlaveSetEvent() Function

The `DlnI2cSlaveSetEvent()` function configures the I^{2}C slave events generation conditions for the specified I^{2}C slave port and I^{2}C slave address.

I^{2}C slave events can vary for different I^{2}C slave addresses. Specify the number of the I^{2}C slave address in the **slaveAddressNumber** parameter.

Syntax

```
DLN_RESULT DlnI2cSlaveSetEvent(
   HDLN handle,
   uint8_t port,
   uint8_t slaveAddressNumber,
   uint8_t eventType,
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the I^{2}C slave port.

slaveAddressNumber

A number of the I^{2}C slave address. The **slaveAddressNumber** parameter is zero base. Possible value are from zero up to (but not including) the value returned by the `DlnI2cSlaveGetAddressCount()` function.

eventType

A condition for I^{2}C slave event generation. The following values are available:

- `I2C_SLAVE_EVENT_NONE (0)` - No I^{2}C slave events are generated.
- `I2C_SLAVE_EVENT_READ (1)` - I^{2}C slave events are generated when an I^{2}C master device reads data from the PC-I2C adapter.
- `I2C_SLAVE_EVENT_WRITE (2)` - I^{2}C slave events are generated when an I^{2}C master device writes data to the PC-I^{2}C adapter.
- `I2C_SLAVE_EVENT_READ_WRITE (3)` - I^{2}C slave events are generated both for read and write transactions.

For additional details, read the I2C Slave Events section.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured I^{2}C slave events.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnI2cSlaveGetPortCount()` function to find the maximum possible port number.

DLN_RES_BUSY (0xB6)

The function cannot configure I^{2}C slave addresses for the DLN adapter while the I^{2}C slave port is busy transmitting data.

DLN_RES_INVALID_EVENT_TYPE (0xA9)

The specified event type is not valid. Use the `DlnI2cSlaveGetSupportedEventTypes()` function to check the list of supported I^{2}C slave events.

Remarks

The **`DlnI2cSlaveSetEvent()`** function is defined in the *dln_i2c_slave.h* file.

DlnI2cSlaveGetEvent() Function

The **`DlnI2cSlaveGetEvent()`** function retrieves settings for I^{2}C event generation for the specified I^{2}C slave port and I^{2}C slave address.

Syntax

```
DLN_RESULT DlnI2cSlaveGetEvent(
   HDLN handle,
   uint8_t port,
   uint16_t slaveAddressNumber
   uint8_t* eventType,
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the I^{2}C slave port.

slaveAddressNumber

A number of the I^{2}C slave address to retrieve settings for. Use the **`DlnI2cSlaveGetAddressCount()`** function to obtain the total number of supported I^{2}C slave addresses.

eventType

A pointer to an unsigned 8-bit integer that receives the current event generation settings. The following values are supported:

- **`I2C_SLAVE_EVENT_NONE (0)`** - No I^{2}C slave events are generated.
- **`I2C_SLAVE_EVENT_READ (1)`** - I^{2}C slave events are generated when an I^{2}C master device reads data from the PC-I2C adapter.
- **`I2C_SLAVE_EVENT_WRITE (2)`** - I^{2}C slave events are generated when an I^{2}C master device writes data to the PC-I^{2}C adapter.
- **`I2C_SLAVE_EVENT_READ_WRITE (1)`** - I^{2}C slave events are generated both for read and write transactions.

Refer to I2C Slave Events section for additional information.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the current I^{2}C slave events configuration.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnI2cSlaveGetPortCount()`** function to find the

maximum possible port number.

Remarks

The **`DlnI2cSlaveGetEvent()`** function is defined in the *dln_i2c_slave.h* file.

DlnI2cSlaveGetSupportedEventTypes() Function

The **`DlnI2cSlaveGetSupportedEventTypes()`** function returns all supported I^{2}C slave event types for opened DLN-series adapter.

Syntax

```
DLN_RESULT DlnI2cSlaveGetSupportedEventTypes(
    HDLN handle,
    uint8_t port,
    DLN_I2C_SLAVE_EVENT_TYPES *supportedEventTypes
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

I^{2}C slave port number.

supportedEventTypes

The pointer to **`DLN_I2C_SLAVE_EVENT_TYPES`** structure that receives the supported event types.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the supported I^{2}C slave events.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnI2cSlaveGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnI2cSlaveGetSupportedEventTypes()`** function is defined in *dln_spi_slave.h* file.

4.3.5 I2C Event Structures

This section describes the structures used for I^{2}C events. These structures are declared in the *dln_i2c_slave.h* file.

DLN_I2C_SLAVE_READ_EV Structure

The **`DLN_I2C_SLAVE_READ_EV`** structure contains information about the I^{2}C read event.

Syntax

```
typedef struct
{
        DLN_MSG_HEADER header;
        uint16_t eventCount;
        uint8_t eventType;
        uint8_t port;
        uint8_t slaveAddress;
        uint16_t size;
} __PACKED_ATTR DLN_I2C_SLAVE_READ_EV;
```

Members

header

Defines the DLN message header. It should have the following value: **`DLN_MSG_ID_I2C_SLAVE_READ_EV (0x0C10)`**.

eventCount

The number of events generated after the event configuration changed.

eventType

The type of generating events. The following values are possible:

Value	Constant	Description
1	**`DLN_I2C_SLAVE_EVENT_READ`**	Events are generated when the I^{2}C master initiates receiving data from the I^{2}C slave device.
3	**`DLN_I2C_SLAVE_EVENT_READ_WRITE`**	Events are generated when the I^{2}C master initiates receiving data from or transmitting data to the I^{2}C slave device.

port

The number of the I^{2}C slave port where the event is generated.

slaveAddress

The I^{2}C slave address assigned to the DLN adapter that generated the event.

size

A size of the buffer that stores the event data.

DLN_I2C_SLAVE_WRITE_EV Structure

The **`DLN_I2C_SLAVE_WRITE_EV`** structure contains information about the I^{2}C write event.

Syntax

```
typedef struct
{
        DLN_MSG_HEADER header;
        uint16_t eventCount;
        uint8_t eventType;
        uint8_t port;
        uint8_t slaveAddress;
        uint16_t size;
        uint8_t buffer[DLN_I2C_SLAVE_BUFFER_SIZE];
} __PACKED_ATTR DLN_I2C_SLAVE_WRITE_EV;
```

Members

header

Defines the DLN message header. It should have the following value: **`DLN_MSG_ID_I2C_SLAVE_WRITE_EV (0x0C10)`**.

eventCount

The number of events generated after the event configuration changed.

eventType

The type of generating events. The following values are possible:

Value	Constant	Description
2	**`DLN_I2C_SLAVE_EVENT_WRITE`**	Events are generated when the I^{2}C master initiates transmitting data from the I^{2}C slave device.
3	**`DLN_I2C_SLAVE_EVENT_READ_WRITE`**	Events are generated when the I^{2}C master initiates receiving data from or transmitting data to the I^{2}C slave device.

port

The number of the I^{2}C slave port where the event is generated.

slaveAddress

The I^{2}C slave address assigned to the DLN adapter that generated the event.

size

A size of the buffer that stores the event data.

buffer

The buffer that contains data received from the I^{2}C master.

5. SPI Bus Interface

The Serial Peripheral Interface (SPI) bus is a synchronous serial communication interface used for short distance communication between various peripheral devices. The SPI bus interconnects a single master device with one or more slave devices.

The master device originates the data transmission by selecting a slave device and generating a clock. Data is transmitted in both directions simultaneously (full-duplex mode) or in one direction (half-duplex mode).

DLN adapters can operate as a master device (read SPI Master Interface). DLN-4S adapter can also operate as a slave device (read SPI Slave Interface).

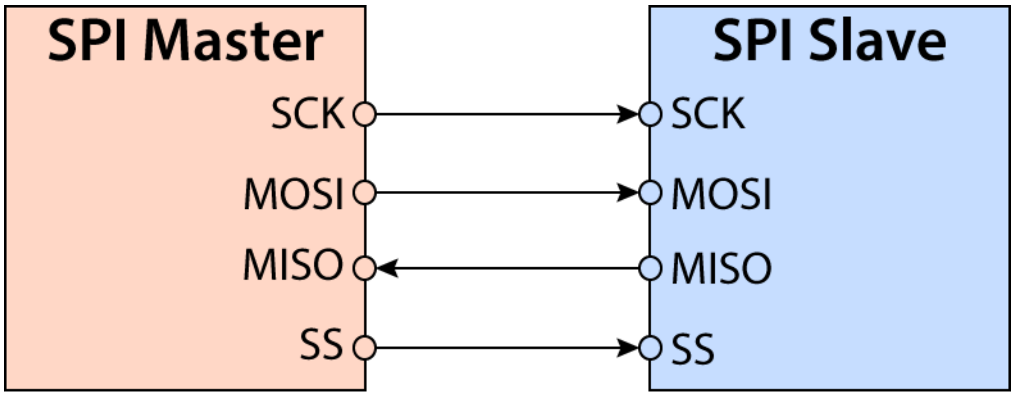
5.1 SPI Bus Structure

The SPI bus uses four signal lines:

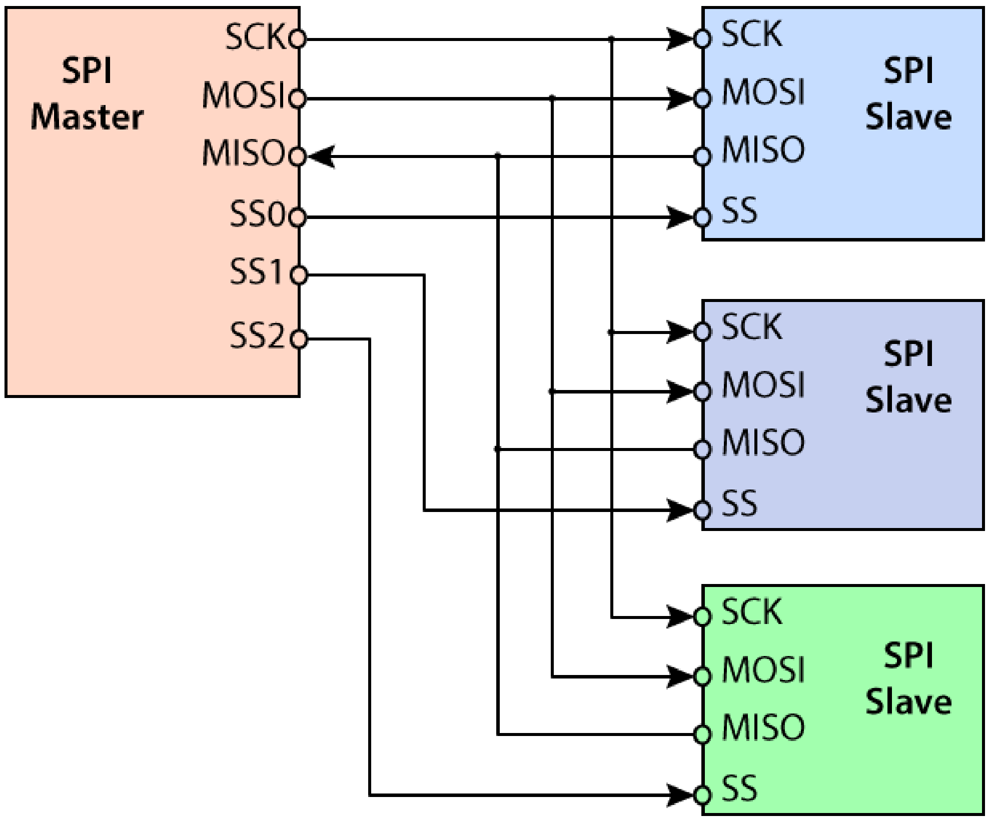
- SCK (Clock) – The master generates the clock to synchronize data transmission.
- MOSI (Master Output, Slave Input) – The master sends data on the MOSI line, the slave receives it.
- MISO (Master Input, Slave Output) – The slave sends data on the MISO line, the master receives it.
- CS or SS (Chip Select or Slave Select) – The master drops the SS line to select the specific SPI slave device. The DLN adapters have several SS lines that you can connect to different slave devices.

Among these four lines, two of them (MOSI and MISO) are data lines, the other two (SS and SCK) are control and synchronization lines.

The SPI bus with a single master and a single slave connects the devices in the following way:



The SPI bus with a single master and multiple slaves connects the devices in the following way:



There are different ways to connect multiple slaves to a single master. See Connecting Multiple Slave Devices for additional information.

5.2 SPI Communication

To initiate communication, the SPI master selects the slave (by pulling the corresponding SS line low) and starts generating the clock signal.

The clock signal synchronizes data transmission both from the master to a slave (the MOSI line) and from the slave to the master (the MISO line). The clock phase and polarity defines the clock phases where the master and the slave can sample data on the MOSI and MISO lines.

5.3 SPI Flash Interface

5.3.1 SPI Flash Functions

DlnSpiFlashGetPortCount() Function

The `DlnSpiFlashGetPortCount()` function retrieves the total number of SPI flash ports available in your DLN-series adapter.

Syntax

```
DLN_RESULT DlnSpiFlashGetPortCount(
  HDLN handle,
  uint8_t *count
);
```

Parameters

handle

A handle to the DLN-series adapter.

count

A pointer to an unsigned 8-bit integer that receives the number of available SPI flash ports.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The operation completed successfully and total number of SPI flash ports were retrieved.

Remarks

The **`DlnSpiFlashGetPortCount()`** function is defined in the *dln_spi_flash.h* file.

DlnSpiFlashEnable() Function

The **`DlnSpiFlashEnable()`** function activates the specified SPI flash port on your DLN-series adapter.

Syntax

```
DLN_RESULT DlnSpiFlashEnable(
  HDLN handle,
  uint8_t port,
  uint16_t *conflict
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI flash port.

conflict

A pointer to an unsigned 16-bit integer that receives a number of the conflicted pin, if any.

A conflict arises if a pin is already assigned to another module of the DLN-series adapter and cannot be used by the SPI flash module. To fix this, check which module uses the pin (call the **`DlnGetPinCfg()`** function), disconnect the pin from that module and call the **`DlnSpiMasterEnable()`** function once again. If there is another conflicting pin, its number will be returned.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully activated the SPI flash port.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiFlashGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiFlashEnable()`** function is defined in the *dln_spi_flash.h* file.

DlnSpiFlashDisable() Function

The **`DlnSpiFlashDisable()`** function deactivates the specified SPI flash port on your DLN-series adapter.

Syntax

```
DLN_RESULT DlnSpiFlashDisable(
  HDLN handle,
  uint8_t port
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI flash port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully deactivated the SPI flash port.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiFlashGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiFlashDisable()`** function is defined in the *dln_spi_flash.h* file.

DlnSpiFlashIsEnabled() Function

The **`DlnSpiFlashIsEnabled()`** function retrieves information whether the specified SPI flash port is activated.

Syntax

```
DLN_RESULT DlnSpiFlashIsEnabled(
  HDLN handle,
  uint8_t port,
  uint8_t *enabled
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI flash port.

enabled

A pointer to an unsigned 8-bit integer that receives information whether the specified SPI flash port is activated or not. There are two possible values:

- 0 or **`DLN_SPI_FLASH_DISABLED`** - the port is not configured as SPI master.
- 1 or **`DLN_SPI_FLASH_ENABLED`** - the port is configured as SPI master.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the SPI flash port status.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiFlashGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiFlashIsEnabled()`** function is defined in the *dln_spi_flash.h* file.

DlnSpiFlashSetFrequency() Function

The **`DlnSpiFlashSetFrequency()`** function sets the clock frequency on the SCK line.

Syntax

```
DLN_RESULT DlnSpiFlashSetFrequency(
  HDLN handle,
  uint8_t port,
  uint32_t frequency
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI flash port.

frequency

SCK line frequency value, specified in Hz. You can specify any value within the range, supported by the DLN-series adapter.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured the clock frequency.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiFlashGetPortCount()` function to find the maximum possible port number.

Remarks

The `DlnSpiFlashSetFrequency()` function is defined in *dln_spi_flash.h* file.

DlnSpiFlashGetFrequency() Function

The `DlnSpiFlashGetFrequency()` function retrieves the current setting for SPI clock frequency.

Syntax

```
DLN_RESULT DlnSpiFlashGetFrequency(
  HDLN handle,
  uint8_t port,
  uint32_t *frequency
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI flash port.

frequency

A pointer to an unsigned 32-bit integer that receives the current SPI clock frequency value in Hz.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the current clock frequency.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiFlashGetPortCount()` function to find the

maximum possible port number.

Remarks

The **`DlnSpiFlashGetFrequency()`** function is defined in the *dln_spi_flash.h* file.

DlnSpiFlashSetSS() Function

The **`DlnSpiFlashSetSS()`** function selects a Slave Select (SS) line.

Syntax

```
DLN_RESULT DlnSpiFlashSetSS(
  HDLN handle,
  uint8_t port,
  uint8_t ss
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI flash port.

ss

The value on the SS lines. The bits 4-7 are reserved and must be set to 1.

Attention: If you expect slaves to output data, you must ensure that only one slave is activated. If several slaves start outputting data simultaneously, the equipment can be damaged.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully selected the SS line.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiFlashGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiFlashSetSS()`** function is defined in *dln_spi_flash.h* file.

DlnSpiFlashGetSS() Function

The **`DlnSpiFlashGetSS()`** function retrieves the current Slave Select (SS) line.

Syntax

```
DLN_RESULT DlnSpiFlashGetSS(
  HDLN handle,
  uint8_t port,
  uint8_t *ss
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI flash port.

ss

A pointer to an unsigned 8-bit integer that receives the value on the SS lines.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieves the selected SS line.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiFlashGetPortCount()` function to find the maximum possible port number.

Remarks

The **`DlnSpiFlashGetSS()`** function is defined in *dln_spi_flash.h* file.

DlnSpiFlashSetSSMask() Function

The **`DlnSpiFlashSetSSMask()`** function selects a required Slave Select (SS) lines by using mask value.

Syntax

```
DLN_RESULT DlnSpiFlashSetSSMask(
  HDLN handle,
  uint8_t port,
  uint8_t ssMask
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI flash port.

ssMask

The mask value to set the SS lines. The bits 4-7 are reserved and must be set to 1.

Attention: If you expect slaves to output data, you must ensure that only one slave is activated. If several slaves start outputting data simultaneously, the equipment can be damaged.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully selected SS lines.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiFlashGetPortCount()` function to find the maximum possible port number.

Remarks

The **`DlnSpiFlashSetSSMask()`** function is defined in *dln_spi_flash.h* file.

DlnSpiFlashGetSSMask() Function

The **`DlnSpiFlashGetSSMask()`** function retrieves the mask value of current selected Slave Select (SS) lines.

Syntax

```
DLN_RESULT DlnSpiFlashGetSSMask(
  HDLN handle,
  uint8_t port,
  uint8_t *ssMask
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI flash port to retrieve the information from.

ssMask

A pointer to an unsigned 8-bit integer that receives the mask value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the selected SS lines.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiFlashGetPortCount()` function to find the

maximum possible port number.

Remarks

The **`DlnSpiFlashGetSSMask()`** function is defined in the *dln_spi_flash.h* file.

DlnSpiFlashProgramPage() Function

The **`DlnSpiFlashProgramPage()`** function transfers data by using SPI flash interface.

Syntax

```
DLN_RESULT DlnSpiFlashProgramPage(
  HDLN handle,
  uint8_t port,
  uint32_t address,
  uint8_t *buffer,
  uint16_t size,
  uint32_t timeout
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI flash port.

address

Address value.

buffer

Pointer to byte variable with data to be sent.

size

Buffer size.

timeout

Timeout value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the selected SS lines.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiFlashGetPortCount()`** function to find the maximum possible port number.

DLN_RES_INVALID_BUFFER_SIZE (0xAE)

The buffer size is not valid. This value cannot exceed 256 bytes.

DLN_RES_DISABLED (0xB7)

The SPI flash port is disabled. Use the **`DlnSpiFlashEnable()`** function to activate the SPI flash port.

Remarks

The **`DlnSpiFlashProgramPage()`** function is defined in *dln_spi_flash.h* file.

DlnSpiFlashReadPage() Function

The **`DlnSpiFlashReadPage()`** function receives data via SPI flash interface. The data is received as an array of 1-byte elements.

Syntax

```
DLN_RESULT DlnSpiFlashReadPage(
  HDLN handle,
  uint8_t port,
  uint32_t address,
  uint8_t *buffer,
  uint16_t size
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI flash port.

address

Device address for reading data from.

buffer

A pointer to an array of unsigned 8-bit integers. This array will be filled with data received.

size

Read data size.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the selected SS lines.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiFlashGetPortCount()`** function to find the maximum possible port number.

DLN_RES_INVALID_BUFFER_SIZE (0xAE)

The buffer size is not valid. This value cannot exceed 256 bytes.

DLN_RES_DISABLED (0xB7)

The SPI flash port is disabled. Use the **`DlnSpiFlashEnable()`** function to activate the SPI flash port.

Remarks

The `DlnSpiFlashReadPage()` function is defined in the *dln_spi_flash.h* file.

5.4 SPI Master Interface

Most of the DLN adapters support the SPI master interface. Some of them have several independent SPI master ports.

Before transmitting data, you need to configure the SPI master ports according to the slave requirements and enable it (See Configuring the SPI Master Interface).

You can either transmit data in full-duplex or in half-duplex mode (See SPI Data Transmission).

If you need to work with multi slave devices, you can interconnect them following the instructions in Connecting Multiple Slave Devices.

5.4.1 Configuring the SPI Master Interface

For reliable SPI communication, you have to configure the SPI master interface according to the SPI slave requirements:

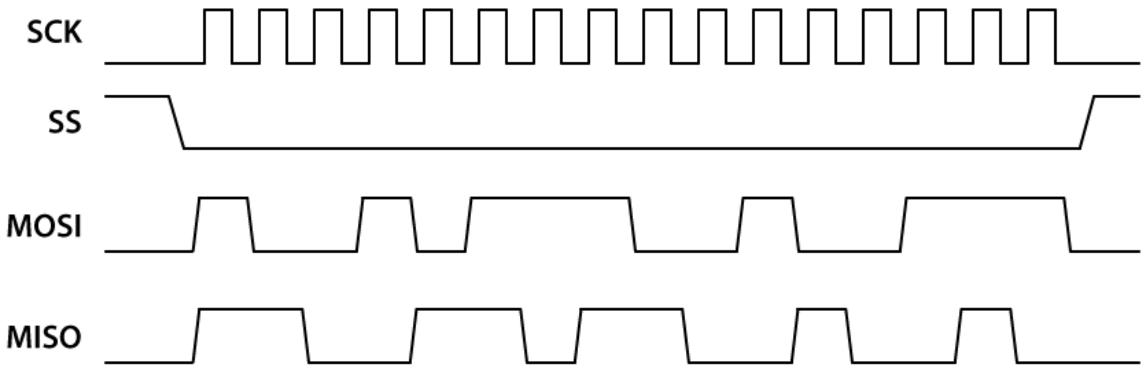
1. Configure the clock (SCK) signal, using a frequency, which does not exceed the maximum frequency that the slave device supports. For details, read Clock Frequency.
2. Configure the transmission mode (clock polarity and clock phase). The clock polarity (CPOL) and clock phase (CPHA) configuration must be the same for both SPI master and SPI slave devices. For details, read Clock Phase and Polarity.
3. Configure the frame size. The frame size instructs the DLN adapter how to treat data inside the buffer. The data can be treated as 8-16 bit integers. If the configured frame size exceeds 8 bits, the bytes inside the frame are stored in Little Endian format. For details, read SPI Data Frames.
4. Customize releasing the SS line between frames. Some slave devices require the signal in the SS line going LOW to initiate data transmission. When the frame transmission is complete, the master should pull the SS line HIGH and then LOW again to reinitiate the frame transmission with the same slave. This parameter is optional. For details, read SS Line Release Between Frames.
5. If a slave device needs additional time to process or generate data, configure delays. For details, read SPI Delays.

Note: DLN-1 and DLN-2 adapters do not allow changing the SPI configuration after the SPI master port is enabled. Therefore, if you use any of these adapters, first configure the SPI master port and then enable it.

To start communication, you need to select the slave. Dropping the signal on an SS line initiates data transmission between the master and the appropriate slave device. The Connecting Multiple Slave Devices section describes possible configurations. You can select a slave device before or after enabling the SPI master port.

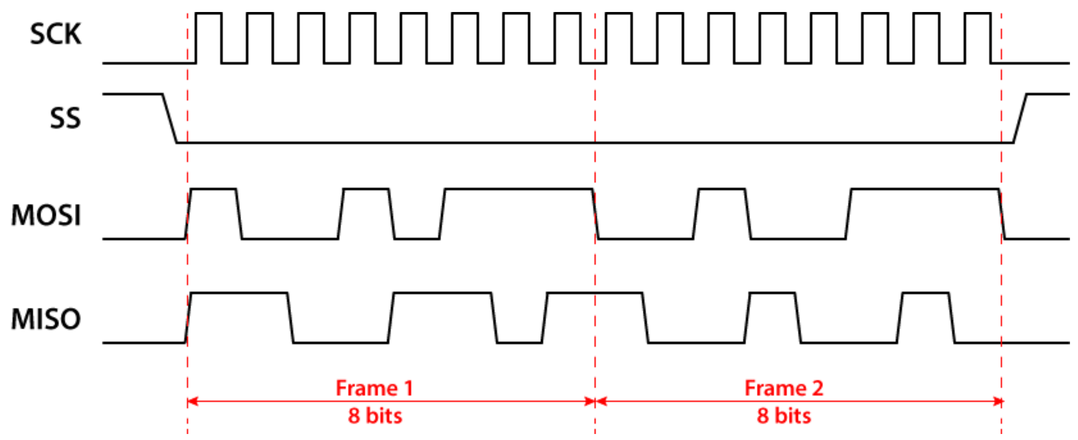
5.4.2 SPI Data Frames

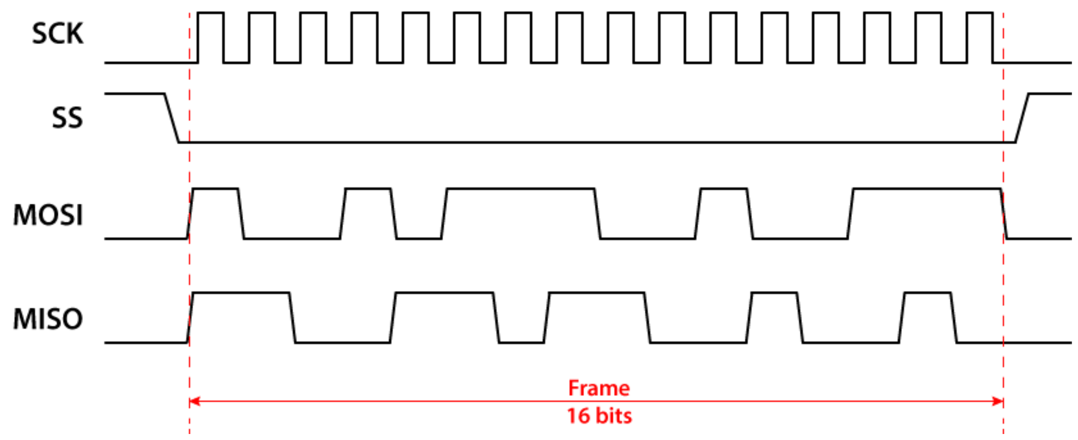
SPI bus allows continuous data transmission. When you pass a buffer (array of words) to one of the SPI Master Transmission Functions, the DLN adapter transmits the data bit after bit as in the following figure.



Logically, your application and an SPI slave device treat this data as an array of 8-bit or 16-bit words. Some SPI slave devices (for example, digital-to-analog or analog-to-digital converters) operate with 12-bit words.

DLN adapters allow you to support a wide range of SPI slave devices. You can configure the frame size (number of bits in the word) by using the `DlnSpiMasterSetFrameSize()` function. DLN adapters support 8 to 16 bits per frame.



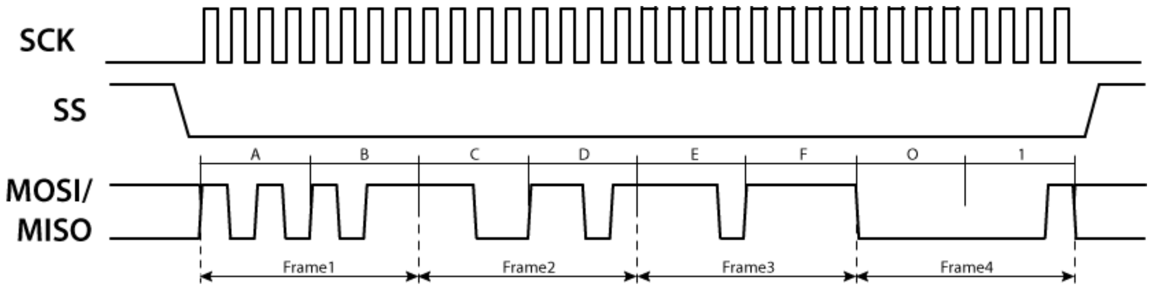


In DLN adapters, the frame data is transmitted in a Little Endian format (starting from the most-significant bit and up to the least-significant bit). If the frame size is not a multiple of 8, the unused (most-significant) bits are discarded, regardless of their content.

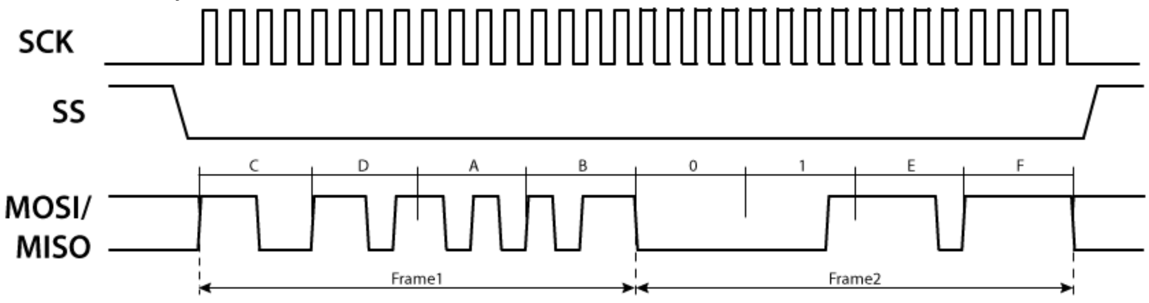
Example

You want to transmit an array with the following data: `ABCDEF01`

If the frame size is 8 bits, the transmission proceeds as four 8-bit words: `AB CD EF 01`

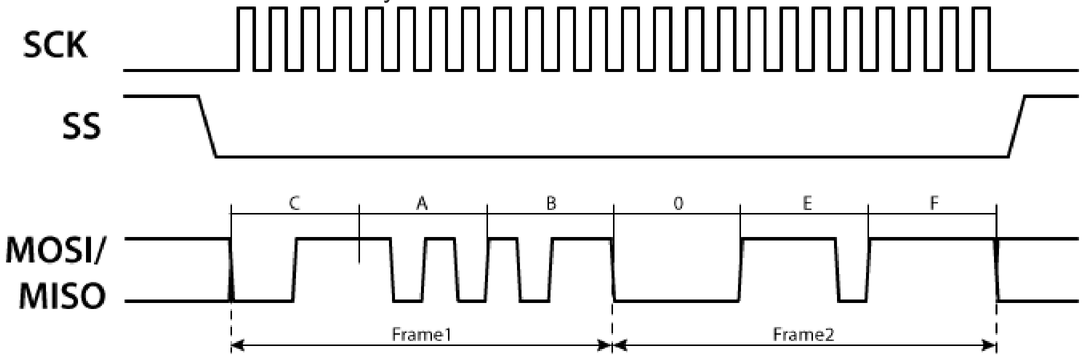


If the frame size is 16 bits, the transmission proceeds as two 16-bit words. Each word is stored in the array in little endian format: `CDAB 01EF`



If the frame size is 12 bits, the transmission proceeds as two 12-bit words. This frame is not a multiple of 8, so the unused (most-significant) bits of the second byte in each word are discarded: `ABC EF0`.

Each word is stored in the array in little endian format: `CAB 0EF`

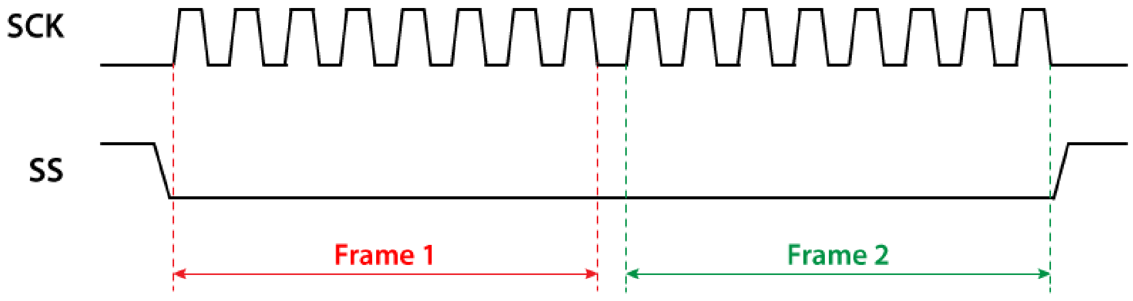


To transmit data as arrays of more than 8 bits, you can use the **`DlnSpiMasterReadWrite16()`** function. It transmits data in little endian format that is very useful because most microcontrollers store data in this format.

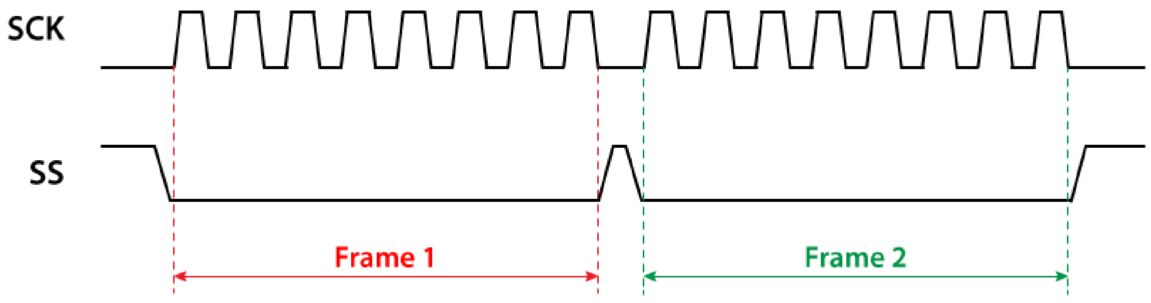
For information about all functions that you can use to transmit data, read the SPI Master Transmission Functions section.

SS Line Release Between Frames

By default, if an array of several frames is transmitted between the master and the same slave, the SS line stays enabled until data transmission completes.



However, some SPI slave devices require the SS line to be deasserted between frames.



To release the SS line, use the `DlnSpiMasterReleaseSS()`. Use the `DlnSpiMasterSSEnable()` function to enable the SS line.

Note: DLN-1 and DLN-2 adapters do not support the SS line release between frames.

If an SPI slave device is not fast enough to process continuously incoming data, you can configure your DLN adapter to introduce a specified delay between frames. For more information about this and other delays, read SPI Delays.

5.4.3 Clock Frequency

SPI bus can operate at very high speeds, which may be too fast for some slave devices. To accommodate such devices, the SPI bus contains the clock (CLK). The signal on the SCK line is transmitted with the same frequency as data flows. Thus, there is no need to synchronize the transmission speed of master and slave devices.

To configure the clock signal of the master, use the `DlnSpiMasterSetFrequency()` function.

When configuring the SPI master interface, you specify the frequency value supported by the slave device. If the specified value is not compatible with your DLN adapter, the function approximates the value to the closest lower frequency value supported by the adapter.

A range of supported clock frequency values depends on the DLN adapter:

- DLN-1 adapters support clock frequency from 2kHz up to 4MHz.
- DLN-2 adapters support clock frequency from 2kHz up to 18MHz.
- DLN-4 adapters support clock frequency from 376kHz up to 48MHz.

In addition to setting the clock frequency, the master also configures the clock polarity (CPOL) and clock phase (CPHA). For detailed information, read Clock Phase and Polarity.

5.4.4 Clock Phase and Polarity

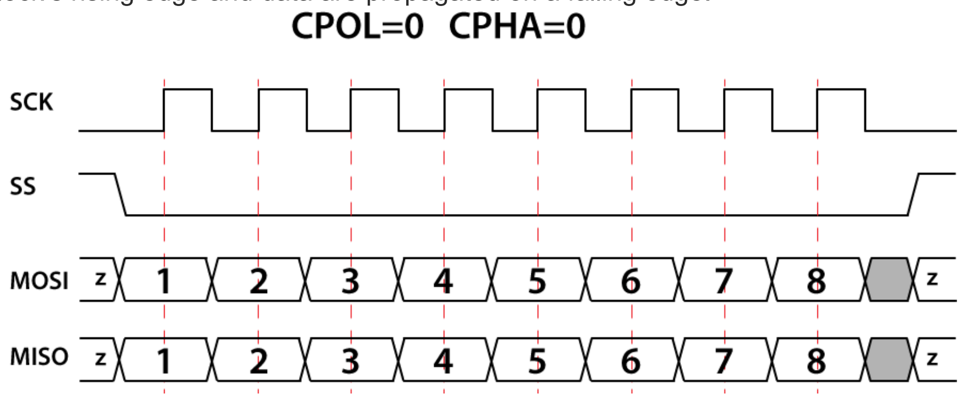
The master configures the clock polarity (CPOL) and clock phase (CPHA) to correspond to slave device requirements. These parameters determine when the data must be stable, when it should be changed according to the clock line and what the clock level is when the clock is not active.

The CPOL parameter assigns the clock level when the clock is not active. The clock (SCK) signal may be inverted (CPOL=1) or non-inverted (CPOL=0). For the inverted clock signal, the first clock edge is falling. For the non-inverted clock signal, the first clock edge is rising.

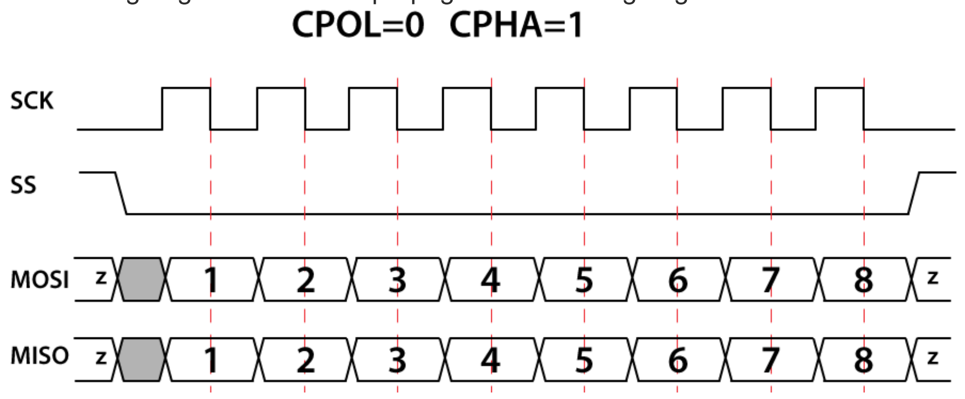
The CPHA parameter is used to shift the capturing phase. If CPHA=0, the data are captured on the leading (first) clock edge, regardless of whether that clock edge is rising or falling. If CPHA=1, the data are captured on the trailing (second) clock edge; in this case, the data must be stable for a half cycle before the first clock cycle.

There are four possible modes that can be used in an SPI protocol:

1. For CPOL=0, the base value of the clock is zero. For CPHA=0, data are captured on the clock's rising edge and data are propagated on a falling edge.

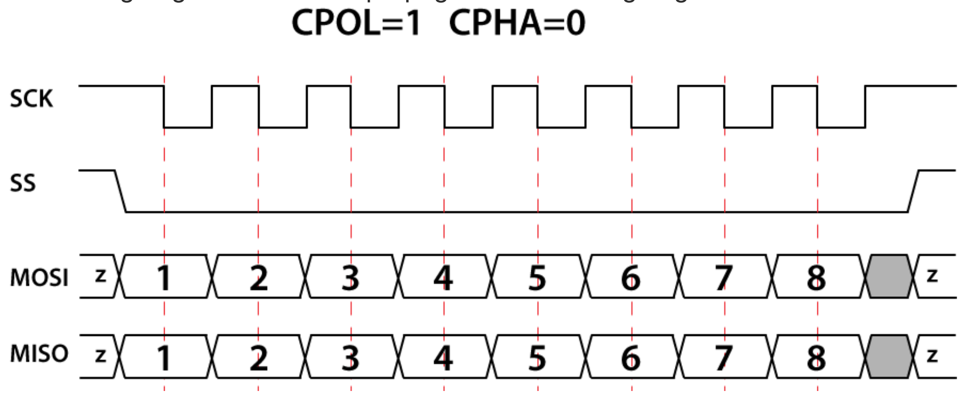


2. For CPOL=0, the base value of the clock is zero. For CPHA=1, data are captured on the clock's falling edge and data are propagated on a rising edge.

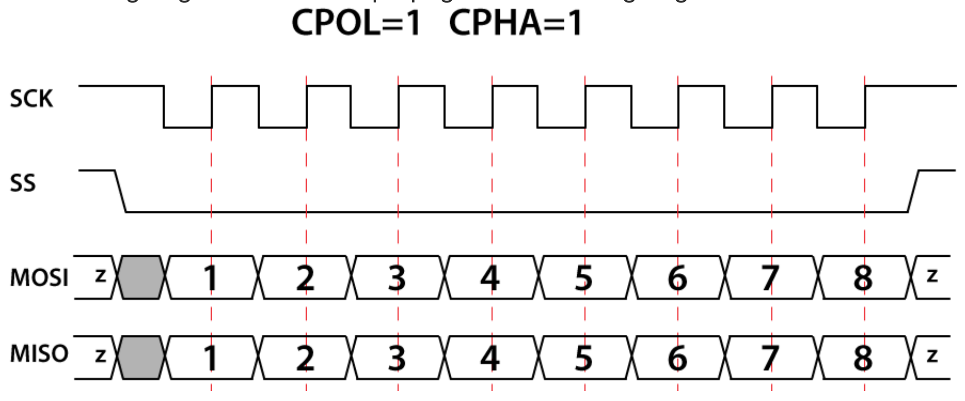


3. For CPOL=1, the base value of the clock is one. For CPHA=0, data are captured on the

clock's rising edge and data are propagated on a falling edge.



4. For CPOL=1, the base value of the clock is one. For CPHA=1, data are captured on the clock's falling edge and data are propagated on a rising edge.



In DLN adapters, the default transmission mode configuration has CPOL=0, CPHA=0 values.

You can specify the mode using the `DlnSpiMasterSetMode()` function. To configure the CPOL and CPHA values separately, use the `DlnSpiMasterSetCpol()` and `DlnSpiMasterSetCpha()` functions.

The master and a slave must use the same set of parameters (clock frequency, CPOL and CPHA); otherwise, a communication will be impossible. If multiple slaves are used, the master configuration should change each time before the master initiates communication with a different slave.

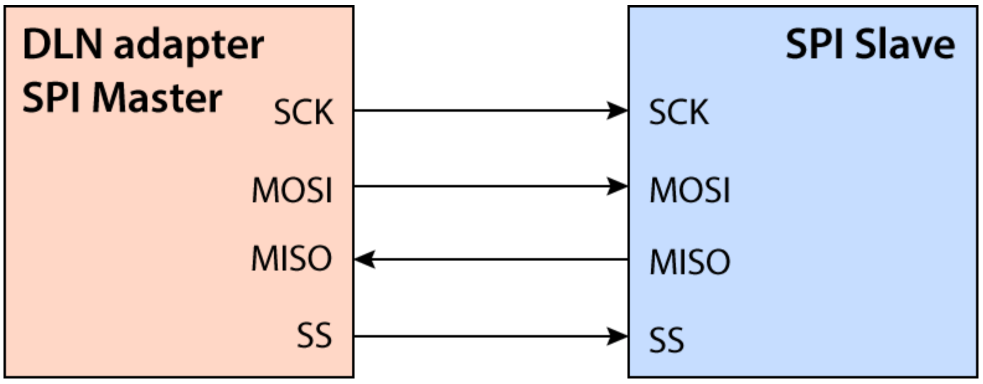
5.4.5 SPI Data Transmission

DLN adapters can operate in the following modes:

- Full-duplex – the master sends data to a slave and receives data from the slave simultaneously.
- Half-duplex (read) – the master receives data from a slave.
- Half-duplex (write) – the master sends data to a slave.

Full-duplex communication

A full-duplex data transmission occurs during each SPI clock cycle: the master transmits data to the slave on the MOSI line and the slave receives it; at the same time, the slave transmits data to the master on the MISO line and the master receives it.



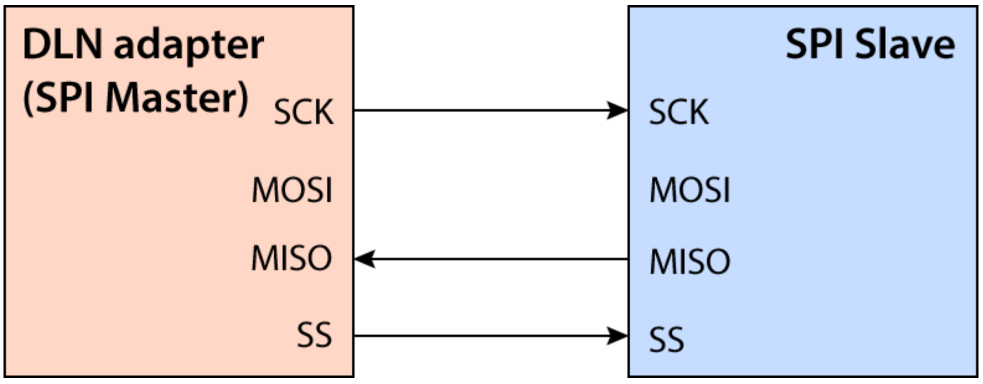
To transmit data in a full-duplex mode, use the `DlnSpiMasterReadWrite()` function (for 8-bit frames) or the `DlnSpiMasterReadWrite16()` function (for 9-16-bit frames).

Transmission may continue for any number of clock cycles. When complete, the master idles the clock and releases the SS line. Using the `DlnSpiMasterReadWriteEx()` function, you can configure whether to release SS line after the transmission or to leave it on the low level.

The `DlnSpiMasterReadWriteSS()` function allows to select the SS line, transmit data and release the SS line. For details, read Connecting Multiple Slave Devices.

Half-duplex (read) communication

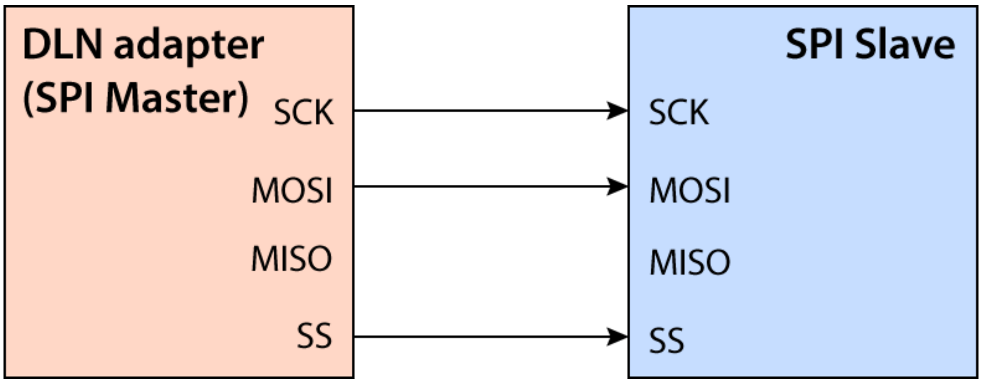
In a half-duplex read mode, the slave transmits data on the MISO line and the master receives it. The MOSI line remains inactive.



To receive data in a half-duplex mode, use the `DlnSpiMasterReadEx()` function. This function allows to configure whether the SS line should be released after the transmission or left on the low level.

Half-duplex (write) communication

In a half-duplex write mode, the master sends data on the MOSI line and the slave receives it. The MISO line remains inactive.



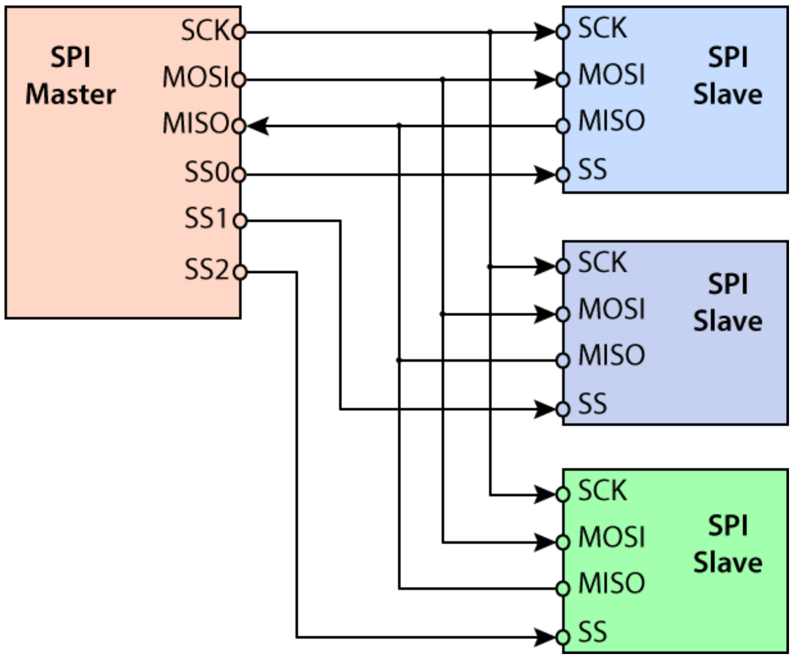
To send data in a half-duplex mode, use the `DlnSpiMasterWriteEx()` function. This function allows to configure whether the SS line should be released after the transmission or left on the low level.

In a half-duplex mode, the master can send data to multiple slaves simultaneously. For details, read Connecting Multiple Slave Devices.

5.4.6 Connecting Multiple Slave Devices

In SPI, a master can communicate with a single or multiple slaves. For applications using multiple slaves, the following configurations are possible:

1. **Independent slaves.** This is a most common configuration of the SPI bus. The MOSI, MISO and SCK lines of all slaves are interconnected. The SS line of every slave device is connected to a separate pin of SPI master device. Since the MISO pins of the slaves are connected together, they are required to be tristate pins (high, low or high-impedance).



To select the slave, the master pulls the corresponding SS line low. Only one slave can be selected.

For DLN-1 or DLN-2 adapters, you can use the following functions:

`DlnSpiMasterSetSS()`
: to select the SS line. The SS line value can include only one bit set to 0;

`DlnSpiMasterReadWrite()`
: to transmit data to/from the slave device;

`DlnSpiMasterReleaseSS()`
: to release the SS line after transmission if the master does not need to transmit more data to the same slave.

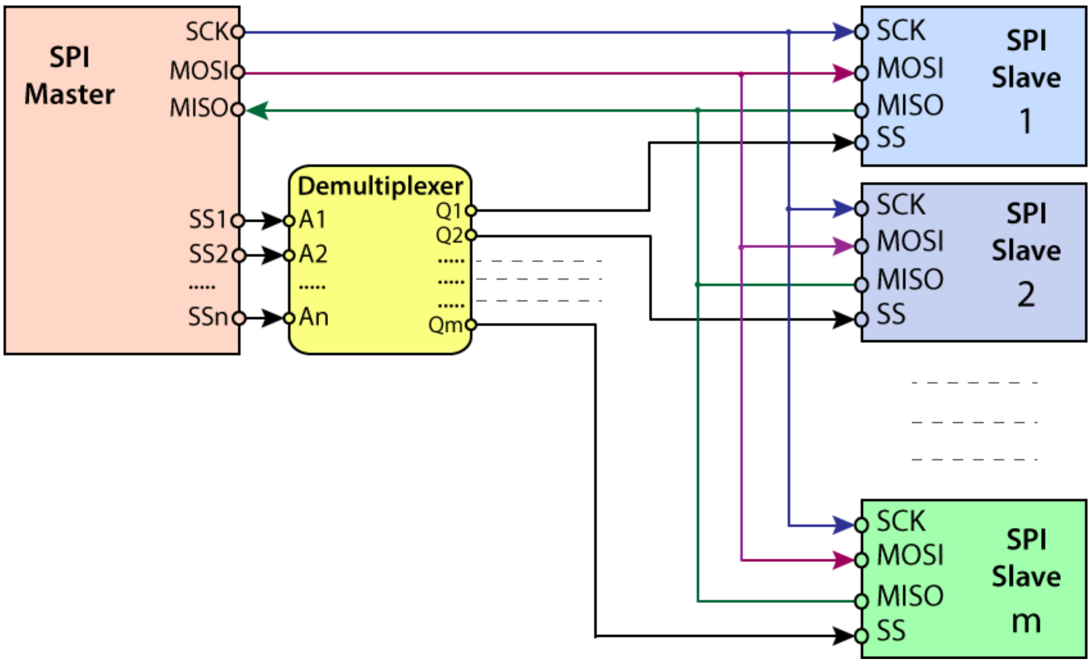
For a DLN-4M adapter, use the `DlnSpiMasterReadWriteSS()` function – it selects the specified SS line, transmits data and releases the SS line.

In a half-duplex read mode, you can also select only one slave. The following functions can be used:

`DlnSpiMasterSetSS()`
: to select the SS line. The SS line value can include only one bit set to 0;

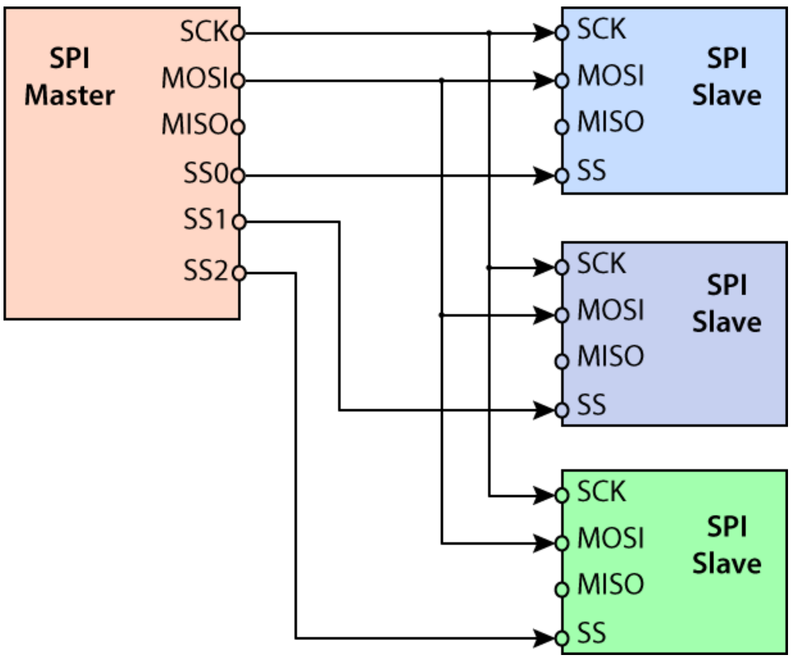
`DlnSpiMasterReadEx()`
: to read data from the slave device and to release the SS line after transmission if required.

2. **Independent slave configuration with decoder/demultiplexer.** The SS lines are used to send an n-bit value, which is the number of the selected slave. Here, the master can communicate with $m=2^n$ slaves.



If you use this configuration, in the `DlnSpiMasterSetSS()` function the value of the SS lines can include more than one zeros. The demultiplexer converts this value and activates only one SS line.

3. **Half-duplex write configuration.** The master transmits data on the MOSI line and the slave receives it. The MISO line remains inactive.

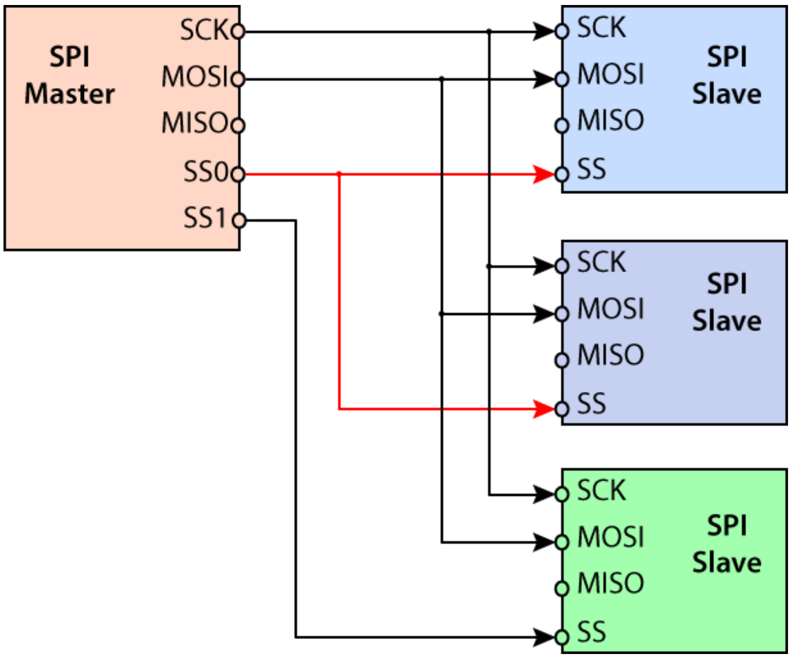


If your slave devices only receive data, you can address multi slaves simultaneously. The following functions can be used:

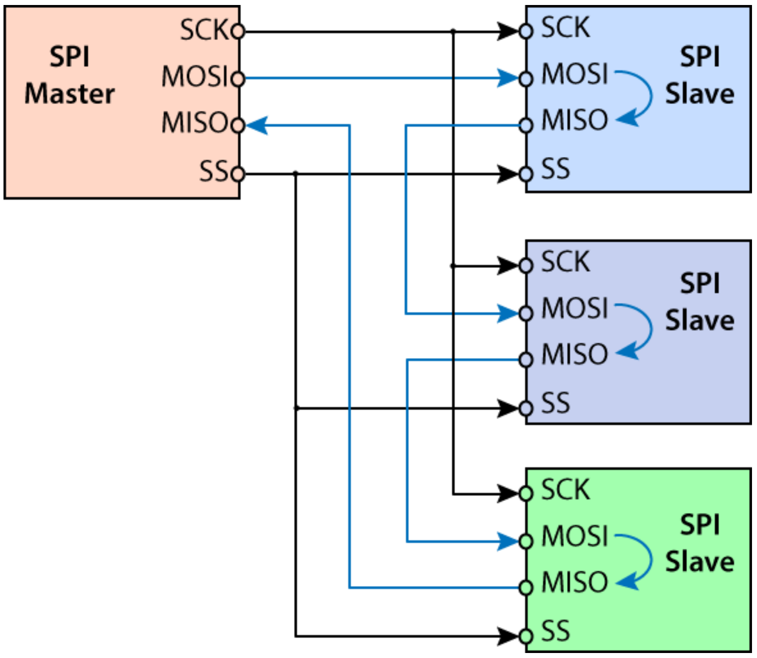
DlnSpiMasterSetSS()
: to select the SS line; the SS line value can include more than one bits set to 0 (several SS lines can be selected – the master can send data to several slave devices simultaneously);

DlnSpiMasterWriteEx()
: to write data from the slave device(s) and to release the SS line(s) after transmission (optionally).

If the master always sends equal data to some group of slaves in a half-duplex write configuration, all the slaves from the group can be connected to one SS line:



4. **Daisy-chain slaves:** the first slave output is connected to the second slave input, etc. The SPI port of each slave is designed to send out during the second group of clock pulses an exact copy of the data it received during the first group of clock pulses.



Note: In DLN-series adapters, you can use unengaged GPIO pins as slave select (SS) lines. In this case, you can control such pins with the GPIO module.

To check the number of available SS lines in the SPI port, use the `DlnSpiMasterGetSSCount()` function.

To select the slave, use the `DlnSpiMasterSetSS()` function. By default, the first SS line (SS0) is selected.

Only one slave can be activated (the signal level on the SS line is low). However, if you use a demultiplexer, you do not select the line, but set the value of the line. The demultiplexer converts this value and selects only one SS line.

The `DlnSpiMasterEnable()` function enables the selected SS line. Use this function after you configure communication settings.

If you do not need to use all SS lines available in a SPI port, you can disable some of them to use them in other modules. On the contrary, you can enable SS lines that were used for other modules. To disable one SS line, use the `DlnSpiMasterSSDisable()` function. To disable several SS lines, use the `DlnSpiMasterSSMultiDisable()` function. To enable one or more SS lines, use the `DlnSpiMasterSSEnable()` or `DlnSpiMasterSSMultiEnable()` function accordingly.

5.4.7 SPI Delays

Sometimes slave devices need additional time to process data. In order to provide this time, DLN-series adapters can insert delays at different data transmission stages:

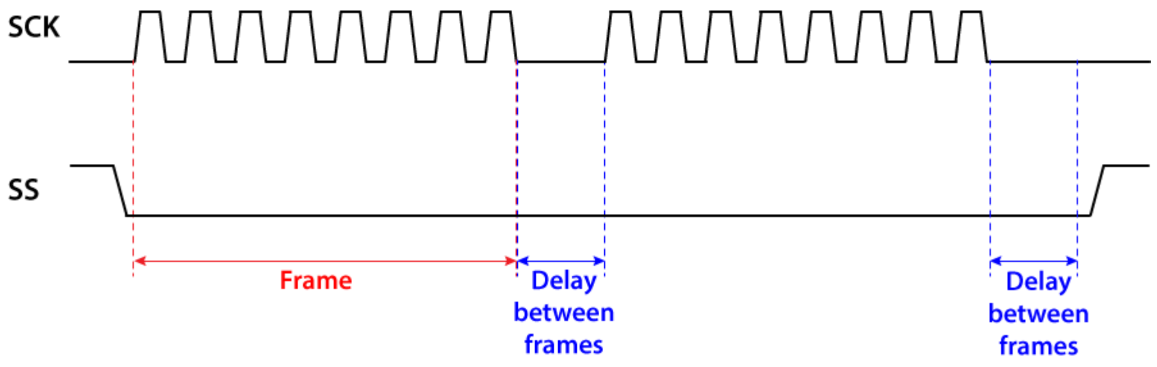
- Delay between data frames;
- Delay after slave selection;
- Delay between slave selections.

All of the delays are set in nanoseconds (ns) and are configured only once. The defined delay values are the same for all SS lines of the port.

Note: DLN-1 and DLN-2 adapters do not support SPI delays.

Delay between data frames

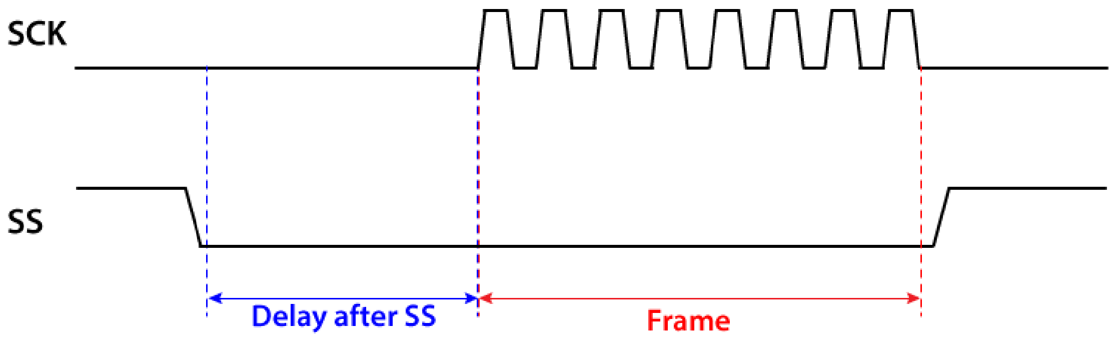
In case a slave device is not fast enough to process continuously incoming data, you can configure the DLN adapter to insert delays between each two consecutive frames. This gives the slave device additional time to process the data from the previous frame. Once enabled, the delay is inserted after each frame.



The default value of the delay between frames is 0ns. The delay value is adjusted using the `DlnSpiMasterSetDelayBetweenFrames()` function. The current delay between frames value can be retrieved by calling the `DlnSpiMasterGetDelayBetweenFrames()` function.

Delay after slave selection

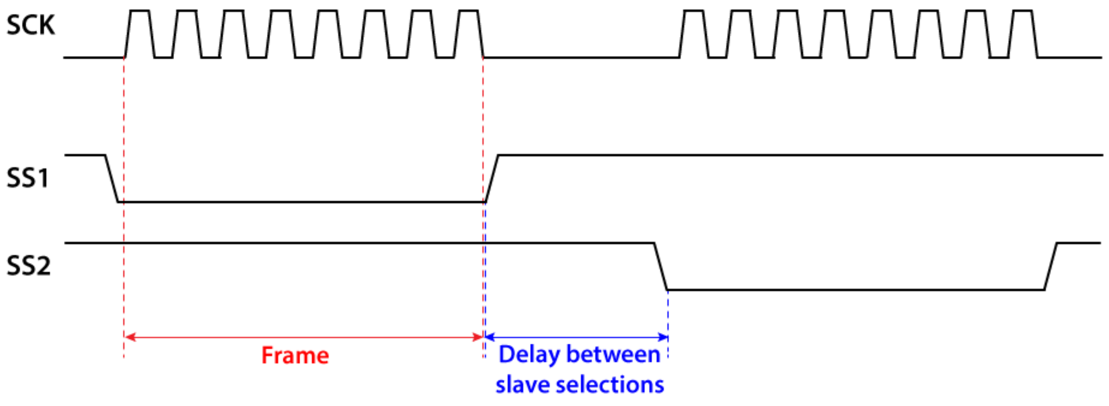
When an SPI slave device needs additional time for initialization, you can configure delay after slave selection (SS). When enabled, the delay is introduces after SS line assertion and before transmission of the first data bit.



The default value of the delay after slave selection is 0ns. To adjust the delay value, use the `DlnSpiMasterSetDelayAfterSS()` function. To retrieve the current delay value, call the `DlnSpiMasterGetDelayAfterSS()` function.

Delay between slave selections

Delay between slave selections is inserted after one SS line release and before assertion of another SS line.



The default value of the delay between slave selections is 62ns. The delay value is adjusted using the `DlnSpiMasterSetDelayBetweenSS()` function, and can be retrieved using the `DlnSpiMasterGetDelayBetweenSS()` function.

5.4.8 SPI Master Transmission Functions

SPI master interface allows you a wide choice of functions for transmission data. These functions differ:

- by operation mode;
- by frame size;
- by additional attributes.

The following list includes possible functions with brief descriptions:

`DlnSpiMasterReadWrite()`
: Provides SPI communication in a full-duplex mode (sends and receives data). The function treats data as arrays of 8-bit frames.

`DlnSpiMasterReadWrite16()`
: Provides SPI communication in a full-duplex mode (sends and receives data). The function treats data as arrays of frames up to 16 bits.

`DlnSpiMasterReadWriteEx()`
: Provides SPI communication in a full-duplex mode (sends and receives data).The function treats data as arrays of 8-bit frames. When transmission completes, the function can release the SS line or leave it active.

`DlnSpiMasterReadWriteSS()`
: Provides SPI communication in a full-duplex mode (sends and receives data). The function treats data as arrays of 8-bit frames. The function allows to select the specified SS line. When transmission completes, the function releases the SS line.

`DlnSpiMasterRead()`
: Provides SPI communication in a half-duplex read mode (receives data). The function treats data as an array of 8-bit frames.

`DlnSpiMasterReadEx()`

Provides SPI communication in a half-duplex read mode (receives data). The function treats data as an array of 8-bit frames. When transmission completes, the function can release the SS line or leave it active.

`DlnSpiMasterWrite()`

Provides SPI communication in a half-duplex write mode (sends data). The function treats data as an array of 8-bit frames.

`DlnSpiMasterWriteEx()`

Provides SPI communication in a half-duplex write mode (sends data). The function treats data as an array of 8-bit frames. When transmission completes, the function can release the SS line or leave it active.

5.4.9 Simple SPI Master Example

The following example shows how to transmit data over SPI bus. For brevity, this example uses default SPI configuration and does not include error detection. You can find the complete example in the "*..\Program Files\Diolan\DLN\examples\c_cpp\examples\simple*" folder after DLN setup package installation.

```
1   #include "..\..\..\common\dln_generic.h"
2   #include "..\..\..\common\dln_spi_master.h"
3   #pragma comment(lib, "..\\..\\..\\bin\\dln.lib")
4
5   int _tmain(int argc, _TCHAR* argv[])
6   {
7   // Open device
8   HDLN device;
9   DlnOpenUsbDevice(&device);
10
11  // Set SPI frequency
12  uint32_t frequency;
13  DlnSpiMasterSetFrequency(device, 0, 100000, &frequency);
14
15  // Enable SPI master
16  uint16_t conflict;
17  DlnSpiMasterEnable(device, 0, &conflict);
18
19  // Prepare output buffer
20  uint8_t input[10], output[10];
21  for (int i = 0; i < 10; i++) output[i] = i;
22
23  // Perform SPI transaction
24  DlnSpiMasterReadWrite(device, 0, 10, output, input);
25
26  // Print received data
27  for (int i = 0; i < 10; i++) printf("%02x ", input[i]);
28
29  // Disable SPI and close device
30  DlnSpiMasterDisable(device, 0, 0);
31  DlnCloseHandle(device);
32
33  return 0;
34  }
```

Line 1: `#include "..\..\..\common\dln_generic.h"`

The dln_generic..h header file declares functions and data structures for the generic interface.

Line 2: `#include "..\..\..\common\dln_spi_master.h"`

The *dln_spi_master.h* header file declares functions and data structures for the SPI master interface.

Line 3: `#pragma comment(lib, "..\\..\\..\\bin\\dln.lib")`

Use specified *dln.lib* library while project linking.

Line 9: `DlnOpenUsbDevice(&device);`

The application establishes the connection with the DLN adapter. This application uses the USB connectivity of the adapter. For additional options, refer to the Device Opening & Identification section.

Line 13: `DlnSpiMasterSetFrequency(device, 0, 100000, frequency);`

The application configures the frequency of the SPI master port 0. You can also configure SPI transmission mode, frame size, etc. See Configuring the SPI Master Interface for details.

Line 17: `DlnSpiMasterEnable(device, 0, conflict);`

The application enables the SPI master port 0. The **`DlnSpiMasterEnable()`** function assigns the corresponding pins to the SPI master module and configures them. If some other module uses a pin required for the SPI bus interface, the **`DlnSpiMasterEnable()`** function returns the **`DLN_RES_PIN_IN_USE`** error code. The **conflict** parameter receives the pin's number.

Lines 21: `for (int i = 0; i < 10; i++) output[i] = i;`

The application allocates buffers for output and input data. The output buffer is filled with the sequential numbers from 0 to 9.

Line 24: `DlnSpiMasterReadWrite(device, 0, 10, output, input);`

The **`DlnSpiMasterReadWrite()`** function transmits the data to and from the SPI slave device. See the SPI Master Transmission Functions section for additional data transmission functions.

Line 27: `for (int i = 0; i < 10; i++) printf("%02x ", input[i]);`

The application prints the buffer received from the SPI slave device.

Line 30: `DlnSpiMasterDisable(device, 0);`

The application releases the SPI master port.

Line 31: `DlnCloseHandle(device);`

The application closes the handle to the DLN adapter.

5.4.10 SPI Master Functions

The default configuration for the SPI master port is as 8-bit SPI master with CPOL=0 and CPHA=0 transmission parameters. By default, the clock is selected with a frequency of 1Mhz.

Using the SPI master functions allows you to change and check the current SPI master configuration, to control data transmission.

The SPI master functions include the following:

General port information:

`DlnSpiMasterGetPortCount()`
: Retrieves the number of SPI master ports available in the DLN adapter.

`DlnSpiMasterEnable()`
: Assigns the selected port to the SPI master module.

`DlnSpiMasterDisable()`
: Releases the selected SPI master port.

`DlnSpiMasterIsEnabled()`
: Retrieves whether the selected port is assigned to the SPI master module.

Configuration functions:

`DlnSpiMasterSetFrequency()`
: Configures the clock frequency for the selected SPI master port.

`DlnSpiMasterGetFrequency()`
: Retrieves the clock frequency configuration for the SPI master port.

`DlnSpiMasterSetFrameSize()`
: Configures the size of a single data frame for the selected SPI master port.

`DlnSpiMasterGetFrameSize()`
: Retrieves the data frame configuration for the selected SPI master port.

`DlnSpiMasterSetMode()`
: Configures the transmission mode based on CPOL and CPHA values.

`DlnSpiMasterGetMode()`
: Retrieves the transmission mode configuration for the selected SPI master port.

`DlnSpiMasterSetCpol()`
: Configures the CPOL value for the selected SPI master port.

`DlnSpiMasterGetCpol()`
: Retrieves the CPOL value for the selected SPI master port.

`DlnSpiMasterSetCpha()`
: Configures the CPHA value for the selected SPI master port.

`DlnSpiMasterGetCpha()`
: Retrieves the CPHA value for the selected SPI master port.

`DlnSpiMasterGetSupportedModes()`
: Retrieves the supported transmission modes for the selected SPI master port.

`DlnSpiMasterGetSupportedCpolValues()`
: Retrieves the supported CPOL values for the selected SPI master port.

`DlnSpiMasterGetSupportedCphaValues()`
: Retrieves the supported CPHA values for the selected SPI master port.

Slave selection functions:

`DlnSpiMasterGetSSCount()`
: Retrieves the available number of SS lines for the selected SPI master port.

`DlnSpiMasterSetSS()`
: Selects a Slave Select (SS) line.

`DlnSpiMasterGetSS()`
: Retrieves the selected SS line.

`DlnSpiMasterSSEnable()`
: Activates the selected SS line.

`DlnSpiMasterSSDisable()`
: Disables the selected SS line.

`DlnSpiMasterSSIsEnabled()`
: Retrieves whether the selected SS line is activated.

`DlnSpiMasterSSMultiEnable()`
: Activates several selected SS lines.

`DlnSpiMasterSSMultiDisable()`
: Disables several selected SS lines.

`DlnSpiMasterSSMultiIsEnabled()`
: Retrieves whether several selected SS lines are activated.

`DlnSpiMasterReleaseSS()`
: Releases the selected SS line.

`DlnSpiMasterSSBetweenFramesEnable()`
: Activates releasing SS line between data frames exchanged with a single slave device.

`DlnSpiMasterSSBetweenFramesDisable()`
: Disables releasing SS line between data frames exchanged with a single slave device.

`DlnSpiMasterSSBetweenFramesIsEnabled()`
: Retrieves whether releasing SS line between data frames is activated.

Transmission functions:

`DlnSpiMasterReadWrite()`
: Provides SPI communication in a full-duplex mode with arrays of 8-bit frames.

`DlnSpiMasterReadWrite16()`
: Provides SPI communication in a full-duplex mode with arrays of frames up to 16 bits.

`DlnSpiMasterReadWriteEx()`
: Provides SPI communication in a full-duplex mode with arrays of 8-bit frames; can release the SS line after transmission.

`DlnSpiMasterReadWriteSS()`
: Provides SPI communication in a full-duplex mode with arrays of 8-bit frames; releases the SS line after transmission.

`DlnSpiMasterRead()`
: Provides SPI communication in a half-duplex read mode with arrays of 8-bit frames.

`DlnSpiMasterReadEx()`
: Provides SPI communication in a half-duplex read mode with arrays of 8-bit frames; can release the SS line after transmission.

`DlnSpiMasterWrite()`
: Provides SPI communication in a half-duplex write mode with arrays of 8-bit frames.

`DlnSpiMasterWriteEx()`
: Provides SPI communication in a half-duplex write mode with arrays of 8-bit frames; can release the SS line after transmission.

Delay functions:

`DlnSpiMasterSetDelayBetweenFrames()`
: Configures a delay between data frames exchanged with a single slave device.

`DlnSpiMasterGetDelayBetweenFrames()`
: Retrieves current delay between data frames exchanged with a single slave device.

`DlnSpiMasterSetDelayAfterSS()`
: Configures a delay between activating SS line and the first data frame.

`DlnSpiMasterGetDelayAfterSS()`
: Retrieves current delay between activating SS line and the first data frame.

`DlnSpiMasterSetDelayBetweenSS()`
: Configures a minimum delay between releasing one SS line and activating another SS line.

`DlnSpiMasterGetDelayBetweenSS()`
: Retrieves current delay between releasing one SS line and activating another SS line.

The *dln_spi_master.h* file declares the SPI Master Interface functions.

DlnSpiMasterGetPortCount() Function

The **`DlnSpiMasterGetPortCount()`** function retrieves the total number of SPI master ports available in your DLN-series adapter.

Syntax

```
DLN_RESULT DlnSpiMasterGetPortCount(
  HDLN handle,
  uint8_t* count
);
```

Parameters

handle

A handle to the DLN-series adapter.

count

A pointer to an unsigned 8-bit integer that receives the number of available SPI master ports.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The operation completed successfully and total number of SPI master ports were retrieved.

Remarks

The **`DlnSpiMasterGetPortCount()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterEnable() Function

The **`DlnSpiMasterEnable()`** function activates the selected SPI master port on your DLN-series adapter.

Syntax

```
DLN_RESULT DlnSpiMasterEnable(
   HDLN handle,
   uint8_t port,
   uint16_t* conflict
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

conflict

A pointer to an unsigned 16-bit integer that receives the number of a conflicted pin, if any.

A conflict arises if the pin is already assigned to another module and cannot be used by the SPI module. To fix this, check which module uses the pin (call the **`DlnGetPinCfg()`** function), disconnect the pin from that module and call the **`DlnSpiMasterEnable()`** function once again. If there is another conflicting pin, its number will be returned.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully activated the SPI master port.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiMasterGetPortCount()`** function to find the maximum possible port number.

DLN_RES_PIN_IN_USE (0xA5)

The port cannot be activated as the SPI master port because one or more pins of the port are assigned to another module. The **conflict** parameter contains the number of a conflicting pin.

DLN_RES_NO_FREE_DMA_CHANNEL (0xAF)

The function cannot be executed because all DMA channels are assigned to another modules of the adapter.

Remarks

DLN-1 and DLN-2 adapters do not allow to change the SPI master configuration after the port is assigned to the SPI master module. Therefore, make sure you have configured the SPI master port before you enable it.

The **`DlnSpiMasterEnable()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterDisable() Function

The **`DlnSpiMasterDisable()`** function releases the selected SPI master port on your DLN-series adapter.

Syntax

```
DLN_RESULT DlnSpiMasterDisable(
   HDLN handle,
   uint8_t port,
   uint8_t waitForTransferCompletion
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

waitForTransferCompletion

Used to choose whether the device should wait for current data transmissions to complete before disabling the SPI master. The following values are available:

- 1 or **`DLN_SPI_MASTER_WAIT_FOR_TRANSFERS`** - wait until transmissions complete.
- 0 or **`DLN_SPI_MASTER_CANCEL_TRANSFERS`** - cancel all pending data transmissions and release the port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully deactivated the SPI master port.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiMasterGetPortCount()`** function to find the maximum possible port number.

DLN_RES_TRANSFER_CANCELLED (0x20)

The pending transmissions were cancelled.

Remarks

The **`DlnSpiMasterDisable()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterIsEnabled() Function

The **`DlnSpiMasterIsEnabled()`** function retrieves whether the specified SPI master port is activated.

Syntax

```
DLN_RESULT DlnSpiMasterIsEnabled(
    HDLN handle,
    uint8_t port,
    uint8_t* enabled
 );
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

enabled

A pointer to an unsigned 8-bit integer that receives the information whether the specified SPI master port is activated. There are two possible values:

- 0 or **`DLN_SPI_MASTER_DISABLED`** - the port is not configured as SPI master.
- 1 or **`DLN_SPI_MASTER_ENABLED`** - the port is configured as SPI master.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved information about the SPI master port.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiMasterGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiMasterIsEnabled()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterSetFrequency() Function

The **`DlnSpiMasterSetFrequency()`** function sets the clock frequency on the SCK line, used to synchronize data transmission between master and slave devices.

Syntax

```
DLN_RESULT DlnSpiMasterSetFrequency(
   HDLN handle,
   uint8_t port,
   uint32_t frequency,
   uint32_t* actualFrequency
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

frequency

SCK line frequency value, specified in Hz. You can specify any value within the range, supported by the DLN-series adapter. This range can be retrieved using the respective function. In case you enter an incompatible value, the function approximates it as the closest lower frequency value, supported by the adapter.

actualFrequency

A pointer to an unsigned 32-bit integer that receives the frequency approximated as the closest to user-defined lower value. The value is specified in Hz and can be null.

Frequency for DLN-4S and DLN-4M adapters is determined by 96 Mhz divided into 2.255. So the nearest frequency values will be 96/2 = 48 Mhz, 96/5 = 19,2 Mhz, 96/10 = 9,6 Mhz.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully set the clock frequency of the SCK line.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

DLN_RES_BUSY (0xB6)

The SPI master is busy transmitting.

DLN_RES_VALUE_ROUNDED (0x21)

The frequency value has been approximated as the closest supported value.

Remarks

By default, the clock is selected with a frequency of 1Mhz. You can check the current value by using the `DlnSpiMasterGetFrequency()` function.

The `DlnSpiMasterSetFrequency()` function is defined in *dln_spi_master.h* file.

DlnSpiMasterGetFrequency() Function

The **`DlnSpiMasterGetFrequency()`** function retrieves the current setting for SPI clock frequency.

Syntax

```
DLN_RESULT DlnSpiMasterGetFrequency(
   HDLN handle,
   uint8_t port,
   uint32_t* frequency
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

frequency

A pointer to an unsigned 32-bit integer that receives the current SPI clock frequency value in Hz.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the clock frequency.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiMasterGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiMasterGetFrequency()`** function is defined in *dln_spi_master.h* file.

DlnSpiMasterSetFrameSize() Function

The **`DlnSpiMasterSetFrameSize()`** function sets the size of a single SPI data frame.

Syntax

```
DLN_RESULT DlnSpiMasterSetFrameSize(
   HDLN handle,
   uint8_t port,
   uint8_t frameSize
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

frameSize

A number of bits to be transmitted. The DLN-series adapter supports 8 to 16 bits per frame.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured the size of a SPI data frame.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

DLN_RES_SPI_INVALID_FRAME_SIZE (0xB8)

The frame size is not valid. The DLN-series adapters support 8 to 16 bits per transmit.

DLN_RES_BUSY (0xB6)

The SPI master is busy transmitting.

Remarks

The default frame size is set to 8 bits. To check the current frame size value, use the `DlnSpiMasterGetFrameSize()` function.

The **`DlnSpiMasterSetFrameSize()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterGetFrameSize() Function

The **`DlnSpiMasterGetFrameSize()`** function retrieves the current size setting for SPI data frame.

Syntax

```
DLN_RESULT DlnSpiMasterGetFrameSize(
   HDLN handle,
   uint8_t port,
   uint8_t* frameSize
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

frameSize

A number of bits to be transmitted in a single frame. The DLN-series adapters support 8 to 16 bits per transmission.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the SPI data frame size.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiMasterGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiMasterGetFrameSize()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterSetMode() Function

The **`DlnSpiMasterSetMode()`** function sets SPI transmission parameters (CPOL and CPHA).

Syntax

```
DLN_RESULT DlnSpiMasterSetMode(
    HDLN handle,
    uint8_t port,
    uint8_t mode
 );
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

mode

A bit field consisting of 8 bits. The bits 0 and 1 correspond to CPOL and CPHA parameters respectively and define the SPI mode. The rest of the bits are not used. You can also use the special constants, defined in the *dln_spi_master.h* file for each of the bits. See Clock Phase and Polarity for additional info.

Bit	Value	Description	Constant
0	0	CPOL=0	DLN_SPI_MASTER_CPOL_0
0	1	CPOL=1	DLN_SPI_MASTER_CPOL_1
1	0	CPHA=0	DLN_SPI_MASTER_CPHA_0
1	1	CPHA=1	DLN_SPI_MASTER_CPHA_1

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured the SPI transmission parameters.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

DLN_RES_BUSY (0xB6)

The SPI master is busy transmitting.

Remarks

By default, the DLN adapter's SPI master port is configured to CPOL=0 CPHA=0 transmission mode.

The **`DlnSpiMasterSetMode()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterGetMode() Function

The **`DlnSpiMasterGetMode()`** function retrieves current configuration of the selected SPI master port.

Syntax

```
DLN_RESULT DlnSpiMasterGetMode(
   HDLN handle,
   uint8_t port,
   uint8_t* mode
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

mode

A pointer to an unsigned 8 bit integer that receives the SPI mode description. See `DlnSpiMasterSetMode()` for details.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the SPI transmission parameters.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

Remarks

The **`DlnSpiMasterGetMode()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterSetCpol() Function

The **`DlnSpiMasterSetCpol()`** function sets the clock polarity value.

Syntax

```
DLN_RESULT DlnSpiMasterSetCpol(
    HDLN handle,
    uint8_t port,
    uint8_t cpol
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

cpol

Clock polarity value. Possible values: 0 or 1.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured the SPI master CPOL value.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiMasterGetPortCount()`** function to find the maximum possible port number.

Remarks

By default, SPI master ports of DLN adapters have CPOL=0 configuration.

The **`DlnSpiMasterSetCpol()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterGetCpol() Function

The **`DlnSpiMasterGetCpol()`** function retrieves the current value of clock polarity (CPOL).

Syntax

```
DLN_RESULT DlnSpiMasterGetCpol(
    HDLN handle,
    uint8_t port,
    uint8_t *cpol
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

cpol

The pointer to uint8_t variable which will be filled with current CPOL value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the SPI master CPOL value.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

Remarks

The **`DlnSpiMasterGetCpol()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterSetCpha() Function

The **`DlnSpiMasterSetCpha()`** function allows to set clock phase (CPHA) value.

Syntax

```
DLN_RESULT DlnSpiMasterSetCpha(
    HDLN handle,
    uint8_t port,
    uint8_t cpha
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the selected SPI master port.

cpha

The clock phase value. Can be 0 or 1.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured the SPI master CPHA value.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

Remarks

By default, SPI master ports of DLN adapters have CPHA=0 configuration.

The **`DlnSpiMasterSetCpha()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterGetCpha() Function

The **`DlnSpiMasterGetCpha()`** function retrieves the current value of clock phase (CPHA).

Syntax

```
DLN_RESULT DlnSpiMasterGetCpha(
    HDLN handle,
    uint8_t port,
    uint8_t *cpha
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

cpha

The pointer to `uint8_t` variable which will be filled with current CPHA value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the SPI master CPHA value.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiMasterGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiMasterGetCpha()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterGetSupportedModes() Function

The **`DlnSpiMasterGetSupportedModes()`** function retrieves all supported SPI master modes (CPOL and CPHA values). For more information, read Clock Phase and Polarity.

Syntax

```
DLN_RESULT DlnSpiMasterGetSupportedModes(
  HDLN handle,
  uint8_t port,
  DLN_SPI_MASTER_MODE_VALUES *values
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

values

Pointer to `DLN_SPI_MASTER_MODE_VALUES` type variable that receives available mode values.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved supported transmission modes.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiMasterGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiMasterGetSupportedModes()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterGetSupportedCpolValues() Function

The **`DlnSpiMasterGetSupportedCpolValues()`** function retrieves supported CPOL values.

Syntax

```
DLN_RESULT DlnSpiMasterGetSupportedCpolValues(
  HDLN handle,
  uint8_t port,
  DLN_SPI_MASTER_CPOL_VALUES *values
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

values

Pointer to `DLN_SPI_MASTER_CPOL_VALUES` type variable that receives available CPOL values.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved supported CPOL values.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiMasterGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiMasterGetSupportedCpolValues()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterGetSupportedCphaValues() Function

The **`DlnSpiMasterGetSupportedCphaValues()`** function retrieves the available CPHA values.

Syntax

```
DLN_RESULT DlnSpiMasterGetSupportedCphaValues(
  HDLN handle,
  uint8_t port,
  DLN_SPI_MASTER_CPHA_VALUES *values
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

values

Pointer to `DLN_SPI_MASTER_CPHA_VALUES` type variable that receives the supported CPHA values.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved supported CPHA values.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiMasterGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiMasterGetSupportedCphaValues()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterGetSSCount() Function

The **`DlnSpiMasterGetSSCount()`** function returns the available number of SS lines.

Syntax

```
DLN_RESULT DlnSpiMasterGetSSCount(
    HDLN handle,
    uint8_t port,
    uint16_t *count
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

count

The pointer to `uint16_t` type variable that receives the number of available SS lines.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the number of SS lines.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiMasterGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiMasterGetSSCount()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterSetSS() Function

The **`DlnSpiMasterSetSS()`** function selects a Slave Select (SS) line.

Syntax

```
DLN_RESULT DlnSpiMasterSetSS(
   HDLN handle,
   uint8_t port,
   uint8_t ss
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port to be configured.

ss

The value on the SS lines. The bits 4-7 are reserved and must be set to 1.

Attention: If you expect slaves to output data, you must ensure that only one slave is activated. If several slaves start outputting data simultaneously, the equipment can be damaged.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully selected the SS line.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

DLN_RES_BUSY (0xB6)

The SPI master is busy transmitting.

DLN_RES_SPI_MASTER_INVALID_SS_NUMBER (0xB9)

The SS value is not valid.

Remarks

The `DlnSpiMasterSetSS()` function is defined in the *dln_spi_master.h* file.

DlnSpiMasterGetSS() Function

The `DlnSpiMasterGetSS()` function retrieves current Slave Select (SS) line.

Syntax

```
DLN_RESULT DlnSpiMasterGetSS(
   HDLN handle,
   uint8_t port,
   uint8_t* ss
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

ss

A pointer to an unsigned 8-bit integer that receives the value on the SS lines.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the current value on the SS lines.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

Remarks

The **`DlnSpiMasterGetSS()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterSSEnable() Function

The **`DlnSpiMasterSSEnable()`** function enables the specified SS line.

Syntax

```
DLN_RESULT DlnSpiMasterSSEnable(
    HDLN handle,
    uint8_t port,
    uint8_t ss
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

ss

The number of the SS line.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully enabled the SS line.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

DLN_RES_SPI_MASTER_INVALID_SS_VALUE (0xB9)

The number of the SS line is invalid. Use the `DlnSpiMasterGetSSCount()` function to check the available number of SS lines.

Remarks

The **`DlnSpiMasterSSEnable()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterSSDisable() Function

The **`DlnSpiMasterSSDisable()`** function disables the specified SS line.

Syntax

```
DLN_RESULT DlnSpiMasterSSDisable(
    HDLN handle,
    uint8_t port,
    uint8_t ss
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

ss

The number of the SS line.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully disabled the SS line.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

DLN_RES_SPI_MASTER_INVALID_SS_VALUE (0xB9)

The number of the SS line is invalid. Use the `DlnSpiMasterGetSSCount()` function to check the available number of SS lines.

Remarks

The **`DlnSpiMasterSSDisable()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterSSIsEnabled() Function

The **`DlnSpiMasterSSIsEnabled()`** function enables the specified SS line.

Syntax

```
DLN_RESULT DlnSpiMasterSSEnable(
    HDLN handle,
    uint8_t port,
    uint8_t ss,
    uint8_t *enabled
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

ss

The number of the SS line.

enabled

A pointer to an unsigned 8-bit integer that receives the information whether the specified SS line is enabled.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully disabled the SS line.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiMasterGetPortCount()`** function to find the maximum possible port number.

DLN_RES_SPI_MASTER_INVALID_SS_VALUE (0xB9)

The number of the SS line is invalid. Use the **`DlnSpiMasterGetSSCount()`** function to check the available number of SS lines.

Remarks

The **`DlnSpiMasterSSIsEnabled()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterSSMultiEnable() Function

The **`DlnSpiMasterSSMultiEnable()`** function enables the specified SS lines.

Syntax

```
DLN_RESULT DlnSpiMasterSSMultiEnable(
    HDLN handle,
    uint8_t port,
    uint8_t ssMask
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

ssMask

A pointer to an unsigned 8-bit integer that receives status of SS lines.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved status of SS lines.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

Remarks

The **`DlnSpiMasterSSMultiEnable()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterSSMultiDisable() Function

The **`DlnSpiMasterSSMultiDisable()`** function enables the specified SS lines.

Syntax

```
DLN_RESULT DlnSpiMasterSSEnable(
    HDLN handle,
    uint8_t port,
    uint8_t ssMask
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

ssMask

The field that defines what SS lines of the port should be disabled. To disable the SS line, set 0 to the corresponding bit.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully disabled the SS lines.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

Remarks

The **`DlnSpiMasterSSMultiDisable()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterSSMultiIsEnabled() Function

The **`DlnSpiMasterSSMultiIsEnabled()`** function enables the specified SS lines.

Syntax

```
DLN_RESULT DlnSpiMasterSSMultiIsEnabled(
    HDLN handle,
    uint8_t port,
    uint8_t *enabled
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

enabled

A pointer to an unsigned 8-bit integer that receives the information about status of SS lines. There are two possible values for every bit:

- 0 - the SS line is disabled.
- 1 - the SS line is enabled.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved status of the SS lines.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

Remarks

The **`DlnSpiMasterSSMultiEnable()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterReleaseSS() Function

The **`DlnSpiMasterReleaseSS()`** function releases SS lines.

Syntax

```
DLN_RESULT DlnSpiMasterReleaseSS(
    HDLN handle,
    uint8_t port
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully released SS lines.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiMasterGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiMasterReleaseSS()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterSSBetweenFramesEnable() Function

The **`DlnSpiMasterSSBetweenFramesEnable()`** function enables release of an SS line between data frames exchanged with a single slave device.

Syntax

```
DLN_RESULT DlnSpiMasterSSBetweenFramesEnable(
   HDLN handle,
   uint8_t port
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the selected SPI master port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully enabled release of the SS line.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

DLN_RES_BUSY (0xB6)

The SPI master is busy transmitting.

Remarks

Note: DLN-1 and DLN-2 adapters do not support releasing the slave select line between frames.

The **`DlnSpiMasterSSBetweenFramesEnable()`** function is defined in *dln_spi_master.h* file.

DlnSpiMasterSSBetweenFramesDisable() Function

The **`DlnSpiMasterSSBetweenFramesDisable()`** function disables release of an SS line between data frames exchanged with a single slave device.

Syntax

```
DLN_RESULT DlnSpiMasterSSBetweenFramesDisable(
   HDLN handle,
   uint8_t port
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the selected SPI master port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully disabled release of the SS line.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

DLN_RES_BUSY (0xB6)

The SPI master is busy transmitting.

Remarks

Note: DLN-1 and DLN-2 adapters do not support releasing the slave select line between frames.

The **`DlnSpiMasterSSBetweenFramesDisable()`** function is defined in *dln_spi_master.h* file.

DlnSpiMasterSSBetweenFramesIsEnabled() Function

The **`DlnSpiMasterSSBetweenFramesIsEnabled()`** function checks whether the DLN-series adapter releases SS line between successive SPI frames transmission.

Syntax

```
DLN_RESULT DlnSpiMasterSSBetweenFramesIsEnabled(
   HDLN handle,
   uint8_t port,
   uint8_t* enabled
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the selected SPI master port.

enabled

A pointer to an unsigned 8-bit integer that receives information whether release of an SS line is enabled. The following values are available:

- 1 or **`DLN_SPI_MASTER_SS_BETWEEN_TRANSFERS_ENABLED`** - the SS line release is enabled.
- 0 or **`DLN_SPI_MASTER_SS_BETWEEN_TRANSFERS_DISABLED`** - the SS line release is disabled.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieves information about the release of the SS line.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiMasterGetPortCount()`** function to find the maximum possible port number.

Remarks

Note: DLN-1 and DLN-2 adapters do not support releasing the slave select line between frames.

The **`DlnSpiMasterSSBetweenFramesIsEnabled()`** function is defined in *dln_spi_master.h* file.

DlnSpiMasterReadWrite() Function

The **`DlnSpiMasterReadWrite()`** function sends and receives data via SPI bus. The received and sent data are arrays of 1-byte elements. This function is suited to transmit data frames of 8 bits or less. In case you set a frame size more than 8 bits, it is advised to use the `DlnSpiMasterReadWrite16()` function.

Syntax

```
DLN_RESULT DlnSpiMasterReadWrite(
    HDLN handle,
    uint8_t port,
    uint16_t size,
    uint8_t *writeBuffer,
    uint8_t *readBuffer
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the selected SPI master port.

size

The size of the message buffer. This parameter is specified in bytes. The maximum value is 256 bytes.

writeBuffer

A pointer to an array of unsigned 8-bit integers that receives data to be sent to a slave during the function execution.

readBuffer

A pointer to an array of unsigned 8-bit integers that receives data from the slave during the function execution.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully transmitted data.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

DLN_RES_DISABLED (0xB7)

The SPI master port is disabled. Use the `DlnSpiMasterEnable()` function to activate the SPI master port.

Remarks

The `DlnSpiMasterReadWrite()` function is defined in *dln_spi_master.h* file.

DlnSpiMasterReadWrite16() Function

The `DlnSpiMasterReadWrite16()` function sends and receives 2-byte frames via SPI bus.

Syntax

```
DLN_RESULT DlnSpiMasterReadWrite16(
   HDLN handle,
   uint8_t port,
   uint16_t count,
   uint16_t* writeBuffer,
   uint16_t* readBuffer
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the selected SPI master port.

count

The number of 2-byte array elements.

writeBuffer

A pointer to an array of unsigned 16-bit integer that receives data to be sent to the slave during the function execution.

readBuffer

A pointer to an array unsigned 16-bit integer that receives data from the slave during the function execution.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully transmitted data.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

DLN_RES_DISABLED (0xB7)

The SPI master port is disabled. Use the **`DlnSpiMasterEnable()`** function to activate the SPI master port.

Remarks

The **`DlnSpiMasterReadWrite16()`** function transmits data frames of 9 to 16 bits. To specify the size of data frames, use the **`DlnSpiMasterSetFrameSize()`** function. In DLN adapters, the frame data is transmitted in a Little Endian format. If the frame size is not a multiple of 8, the most-significant bits are discarded. For details, read SPI Data Frames.

The **`DlnSpiMasterReadWrite16()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterReadWriteSS() Function

The **`DlnSpiMasterReadWriteSS()`** function selects the specified SS line, transmits data via SPI bus and releases the SS line.

Syntax

```
DLN_RESULT DlnSpiMasterReadWriteSS(
    HDLN handle,
    uint8_t port,
    uint8_t ss,
    uint16_t size,
    uint8_t *writeBuffer,
    uint8_t *readBuffer
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the selected SPI master port.

ss

The value on the SS lines. This value can include only one zero bit.

size

The size of the message buffer. This parameter is specified in bytes.

writeBuffer

A pointer to an array of unsigned 8-bit integers that receives data to be sent to a slave.

readBuffer

A pointer to an array of unsigned 8-bit integers that receives data from the slave.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully transmitted data.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

DLN_RES_DISABLED (0xB7)

The SPI master port is disabled. Use the `DlnSpiMasterEnable()` function to activate the SPI master port.

Remarks

The **`DlnSpiMasterReadWriteSS()`** function is defined in *dln_spi_master.h* file.

DlnSpiMasterReadWriteEx() Function

The **`DlnSpiMasterReadWriteEx()`** function sends and receives data via SPI bus. The data is transmitted as an array of 1-byte elements. In this function you can use additional attributes to configure the SS line state after data transmission.

Syntax

```
DLN_RESULT DlnSpiMasterReadWriteEx(
  HDLN handle,
  uint8_t port,
  uint16_t size,
  uint8_t *writeBuffer,
  uint8_t *readBuffer,
  uint8_t attribute
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

size

The size of the message buffer. This parameter is specified in bytes. The maximum value is 256 bytes.

writeBuffer

A pointer to an array of unsigned 8-bit integers that receives data to be sent to a slave during the function execution.

readBuffer

A pointer to an array of unsigned 8-bit integers that receives data from slave during the function execution.

attribute

Additional transmission attribute:

- **`DLN_SPI_MASTER_ATTR_LEAVE_SS_LOW`** - Leave SS low after transmission.
- **`DLN_SPI_MASTER_ATTR_RELEASE_SS`** - Release SS after transmission.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully transmitted data.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

DLN_RES_DISABLED (0xB7)

The SPI master port is disabled. Use the `DlnSpiMasterEnable()` function to activate the SPI master port.

Remarks

The **`DlnSpiMasterReadWriteEx()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterRead() Function

The **`DlnSpiMasterRead()`** function receives data via SPI bus. The received data is an array of 1-byte elements.

Syntax

```
DLN_RESULT DlnSpiMasterRead(
  HDLN handle,
  uint8_t port,
  uint16_t size,
  uint8_t *readBuffer
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

size

The size of the message buffer. This parameter is specified in bytes. The maximum value is 256 bytes.

readBuffer

A pointer to an array of unsigned 8-bit integers that receives data from slave during the function execution.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully received data.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

DLN_RES_DISABLED (0xB7)

The SPI master port is disabled. Use the `DlnSpiMasterEnable()` function to activate the SPI master port.

Remarks

The **`DlnSpiMasterRead()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterReadEx() Function

The **`DlnSpiMasterReadEx()`** function receives data via SPI bus. The received data is an array of 1-byte elements. In this function you can use additional attributes to configure the SS line state after data transmission.

Syntax

```
DLN_RESULT DlnSpiMasterReadEx(
  HDLN handle,
  uint8_t port,
  uint16_t size,
  uint8_t *readBuffer,
  uint8_t attribute
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

size

The size of the message buffer. This parameter is specified in bytes. The maximum value is 256 bytes.

readBuffer

A pointer to an array of unsigned 8-bit integers that receives data from slave during the function execution.

attribute

Additional transmission attribute.

- **`DLN_SPI_MASTER_ATTR_LEAVE_SS_LOW`** - Leave SS low after transmission.
- **`DLN_SPI_MASTER_ATTR_RELEASE_SS`** - Release SS after transmission.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully received data.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

DLN_RES_DISABLED (0xB7)

The SPI master port is disabled. Use the `DlnSpiMasterEnable()` function to activate the SPI master port.

Remarks

The `DlnSpiMasterReadEx()` function is defined in the *dln_spi_master.h* file.

DlnSpiMasterWrite() Function

The `DlnSpiMasterWrite()` function sends data via SPI bus. The data is sent as an array of 1-byte elements.

Syntax

```
DLN_RESULT DlnSpiMasterWrite(
  HDLN handle,
  uint8_t port,
  uint16_t size,
  uint8_t *writeBuffer
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

size

The size of the message buffer. This parameter is specified in bytes. The maximum value is 256 bytes.

writeBuffer

A pointer to an array of unsigned 8-bit integers that receives data to be sent to a slave.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully sent data.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

DLN_RES_DISABLED (0xB7)

The SPI master port is disabled. Use the `DlnSpiMasterEnable()` function to activate the

SPI master port.

Remarks

The **`DlnSpiMasterWriteEx()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterWriteEx() Function

The **`DlnSpiMasterWriteEx()`** function sends data via SPI bus. The data is sent as an array of 1-byte elements. By using this function you can use additional attributes.

Syntax

```
DLN_RESULT DlnSpiMasterWriteEx(
  HDLN handle,
  uint8_t port,
  uint16_t size,
  uint8_t *writeBuffer,
  uint8_t attribute
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

size

The size of the message buffer. This parameter is specified in bytes. The maximum value is 256 bytes.

writeBuffer

A pointer to an array of unsigned 8-bit integers that receives data to be sent to a slave.

attribute

Additional transmission attribute.

- **`DLN_SPI_MASTER_ATTR_LEAVE_SS_LOW`** - Leave SS low after transmission.
- **`DLN_SPI_MASTER_ATTR_RELEASE_SS`** - Release SS after transmission.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully sent data.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiMasterGetPortCount()`** function to find the maximum possible port number.

DLN_RES_DISABLED (0xB7)

The SPI master port is disabled. Use the **`DlnSpiMasterEnable()`** function to activate the SPI master port.

Remarks

The **`DlnSpiMasterWriteEx()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterSetDelayBetweenFrames() Function

The **`DlnSpiMasterSetDelayBetweenFrames()`** function sets a delay between data frames exchanged with a single slave device. For more information, read SPI Delays.

Syntax

```
DLN_RESULT DlnSpiMasterSetDelayBetweenFrames(
   HDLN handle,
   uint8_t port,
   uint32_t delayBetweenFrames,
   uint32_t* actualDelayBetweenFrames
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port to be configured.

delayBetweenFrames

The delay value in nanoseconds. If you specify an unsupported value, it will be approximated as the closest higher supported value.

actualDelayBetweenFrames

Actual set delay value in nanoseconds.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully set the delay between data frames.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiMasterGetPortCount()`** function to find the maximum possible port number.

DLN_RES_BUSY (0xB6)

The SPI master is busy transmitting.

DLN_RES_VALUE_ROUNDED (0x21)

The delay value has been approximated as the closest supported value.

Remarks

Note: DLN-1 and DLN-2 adapters do not support delays between frames.

The **`DlnSpiMasterSetDelayBetweenFrames()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterGetDelayBetweenFrames() Function

The **`DlnSpiMasterGetDelayBetweenFrames()`** function retrieves current setting for delay between data frames exchanged with a single slave device.

Syntax

```
DLN_RESULT DlnSpiMasterGetDelayBetweenFrames(
   HDLN handle,
   uint8_t port,
   uint32_t* delayBetweenFrames
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

delayBetweenFrames

A pointer to an unsigned 32-bit integer that receives the current delay value in nanoseconds.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the delay between data frames.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiMasterGetPortCount()`** function to find the maximum possible port number.

Remarks

Note: DLN-1 and DLN-2 adapters do not support delays between frames.

The **`DlnSpiMasterGetDelayBetweenFrames()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterSetDelayAfterSS() Function

The **`DlnSpiMasterSetDelayAfterSS()`** function sets a delay duration between assertion of an SS line and first data frame. For more information, read SPI Delays.

Syntax

```
DLN_RESULT DlnSpiMasterSetDelayAfterSS(
   HDLN handle,
   uint8_t port,
   uint32_t delayAfterSS,
   uint32_t* actualDelayAfterSS
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

delayAfterSS

The delay value in nanoseconds. If you specify an unsupported value, it will be approximated as the closest higher supported value.

Attention: If you set a 0ns delay time, the actual delay will be equal to 1/2 of the SPI clock frequency, specified by the DlnSpiMasterSetFrequency() function.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully set the delay after slave selection.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiMasterGetPortCount()`** function to find the maximum possible port number.

DLN_RES_BUSY (0xB6)

The SPI master is busy transmitting.

DLN_RES_VALUE_ROUNDED (0x21)

The delay value has been approximated as the closest supported value.

Remarks

Note: DLN-1 and DLN-2 adapters do not support delays after slave selection.

The **`DlnSpiMasterSetDelayAfterSS()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterGetDelayAfterSS() Function

The **`DlnSpiMasterGetDelayAfterSS()`** function retrieves the current setting for minimum delay between assertion of an SS line and the first data frame.

Syntax

```
DLN_RESULT DlnSpiMasterGetDelayAfterSS(
   HDLN handle,
   uint8_t port,
   uint32_t* delayAfterSS
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

delayAfterSS

A pointer to an unsigned 32-bit integer that receives the current delay value in nanoseconds.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the delay after slave selection.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiMasterGetPortCount()`** function to find the maximum possible port number.

Remarks

Note: DLN-1 and DLN-2 adapters do not support delays after slave selection.

The **`DlnSpiMasterGetDelayAfterSS()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterSetDelayBetweenSS() Function

The **`DlnSpiMasterSetDelayBetweenSS()`** function sets a minimum delay between release of an SS line and assertion of another SS line. For more information, read SPI Delays.

Syntax

```
DLN_RESULT DlnSpiMasterSetDelayBetweenSS(
   HDLN handle,
   uint8_t port,
   uint32_t delayBetweenSS,
   uint32_t* actualDelayBetweenSS
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

delayBetweenSS

The delay value in nanoseconds. If you specify an unsupported value, it will be approximated as the closest higher supported value.

actualDelayBetweenSS

Actual set delay value in nanoseconds.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully set the delay after one SS line is released and before another SS line is asserted.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

DLN_RES_BUSY (0xB6)

The SPI master is busy transmitting.

DLN_RES_VALUE_ROUNDED (0x21)

The delay value has been approximated as the closest supported value.

Remarks

Note: DLN-1 and DLN-2 adapters do not support delays between slave selections.

The **`DlnSpiMasterSetDelayBetweenSS()`** function is defined in the *dln_spi_master.h* file.

DlnSpiMasterGetDelayBetweenSS() Function

The **`DlnSpiMasterGetDelayBetweenSS()`** function retrieves the current setting for minimum delay after one SS line is released and before another SS line is asserted.

Syntax

```
DLN_RESULT DlnSpiMasterGetDelayBetweenSS(
   HDLN handle,
   uint8_t port,
   uint32_t* delayBetweenSS
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port.

delayBetweenSS

A pointer to an unsigned 32-bit integer that receives the current delay value in nanoseconds.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully set the delay after one SS line is released and before another SS line is asserted.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiMasterGetPortCount()` function to find the maximum possible port number.

Remarks

Note: DLN-1 and DLN-2 adapters do not support delays between slave selections.

The `DlnSpiMasterGetDelayBetweenSS()` function is defined in the *dln_spi_master.h* file.

5.5 SPI Slave Interface

Some DLN-series adapters support the SPI Slave Interface.

5.5.1 Configuring the Slave for Communication

To configure the slave for the communication, follow these steps:

1. Configure SS idle timeout. When you enable the SPI slave, the SS line may be busy. The DLN adapter waits until the SS line drives high. If the SS line is not released during the SS idle timeout, the SPI slave cannot be enabled. Read SS Idle Timeout.

2. Configure the transmission mode (clock polarity and clock phase). The clock polarity (CPOL) and clock phase (CPHA) configuration must be the same for both SPI master and SPI slave devices.

3. Configure loading data to the reply buffer. The reply mode and reply type parameters can

prevent from losing data if the frame size, configured by you for the SPI slave, does not correspond the frame size set on the SPI master. Read SPI Slave Reply Buffer.

4. If needed, configure events. DLN adapters provide event-driven interface for SPI slave devices. Read SPI Slave Events.

5.5.2 SS Idle Timeout

When you try to enable the SPI slave module, the SS line can appear busy. To prevent transmitting incorrect data, the DLN adapter idles until the SS line is released (drives high). Only then, the SPI slave can be enabled.

To limit the idle time, you can specify the idle timeout value in milliseconds (ms) by calling the **`DlnSpiSlaveSetSSIdleTimeout()`** function. The default value is 100ms. The idle timeout can be from 1ms to 1000ms. If during the specified idle timeout the SS line was not released, the **`DlnSpiSlaveEnable()`** function returns the **`DLN_RES_SPI_SLAVE_SS_IDLE_TIMEOUT`** error.

Note: Do not mix SS idle timeout and idle event timeout.

For details about idle event timeout, read DLN_SPI_SLAVE_EVENT_IDLE Events.

5.5.3 SPI Slave Reply Buffer

When the master initiates transmission, the DLN adapter-slave fills the reply buffer with data that should be sent to the master. To fill the reply buffer, you can use the following functions: **`DlnSpiSlaveLoadReply()`** or **`DlnSpiSlaveEnqueueReply()`**. The **`DlnSpiSlaveLoadReply()`** function clears the buffer and makes the current reply the first in the queue. In contrast, the **`DlnSpiSlaveEnqueueReply()`** function puts the current reply to the end of the queue.

Transmitted data is limited to the number of frames received by the master. The sizes of slave buffer (reply buffer) and master buffer (received buffer) may differ.

If the reply buffer size exceeds the received buffer size, the last bytes of the reply buffer can be lost or sent with the following transmission. It depends on the reply mode you configure. For more information, read Reply Modes.

If the reply buffer size is less than the received buffer size, the last bytes can be filled by zeroes or by repeated reply bytes. It depends on the shortage action you configure. For details, read Reply Shortage Actions.

5.5.4 Reply Modes

If the reply buffer size exceeds the size of the buffer received by the master, the following options are possible:

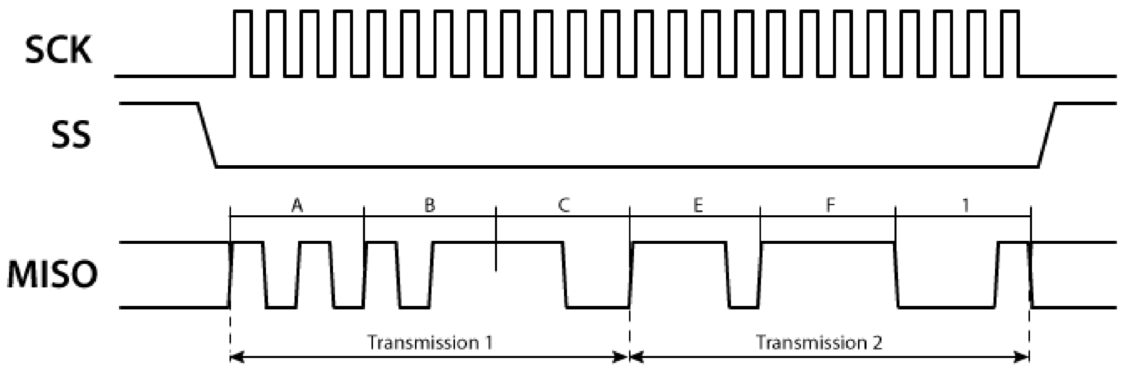
- If the last bytes of the reply are not weighty, they can be rejected. In this case, the master receives the bytes from the reply buffer until the SS line rises. The last bytes of the reply buffer are lost. This mode is called **`DLN_SPI_SLAVE_REPLY_MODE_SS_BASED`**.
- If you do not want to lose any data, the last buffers can be added to the queue. In this

case, the master receives the bytes from the reply buffer until the SS line rises. The last bytes of the reply buffer are not lost, they are added to the queue and the master receives them with the next transmission. This mode is called **DLN_SPI_SLAVE_REPLY_MODE_COUNT_BASED**.

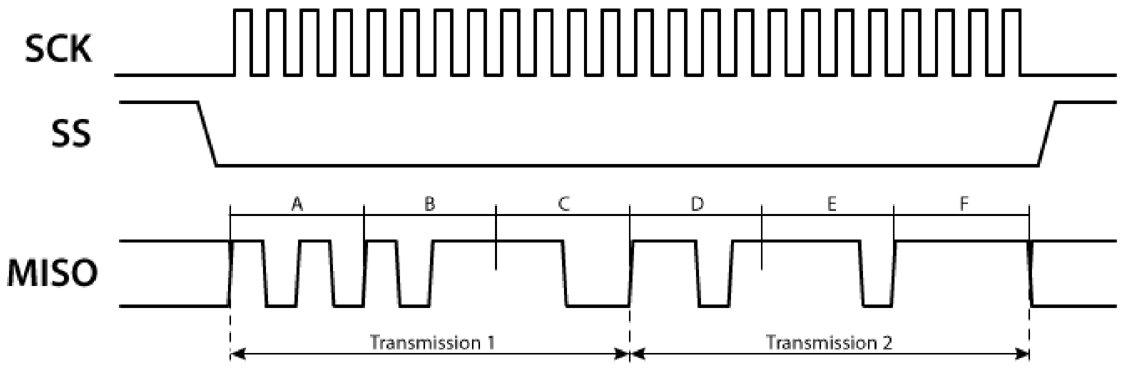
Example

You have two loaded replies: `ABCD` and `EF12`. The master initiates two transmissions of 3 words in each. From the slave, the master receives the following:

- In the **DLN_SPI_SLAVE_REPLY_MODE_SS_BASED** mode, the master receives `ABC` and `EF1`. The last bytes of every reply are lost.



- In the **DLN_SPI_SLAVE_REPLY_MODE_COUNT_BASED** mode, the master receives `ABC` and `DEF`. The last bytes of the second reply is queued for the following transmissions.



To configure the reply mode, use the **DlnSpiSlaveSetReplyMode()** function. The **DlnSpiSlaveGetSupportedReplyModes()** function retrieves all supported reply modes.

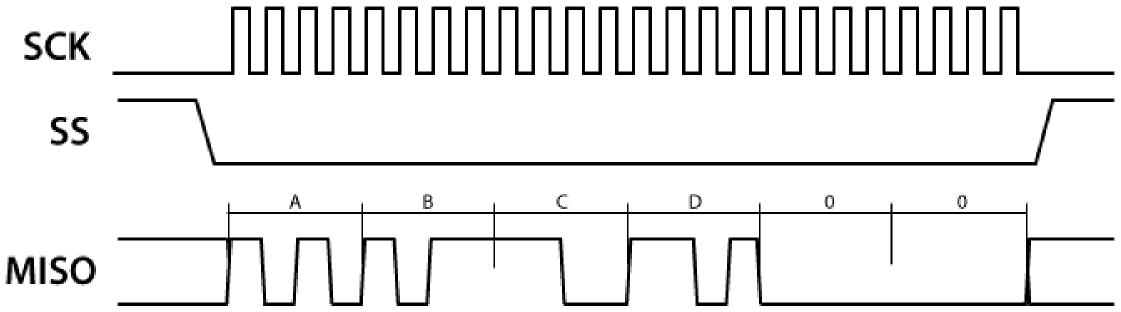
5.5.5 Reply Shortage Actions

If the reply buffer size is less than the size of the buffer received by the master, the missed bytes can be filled by zeroes (the **DLN_SPI_SLAVE_REPLY_SHORTAGE_SEND_ZEROES** reply type) or by repeating the reply bytes (the **DLN_SPI_SLAVE_REPLY_SHORTAGE_REUSE** reply type).

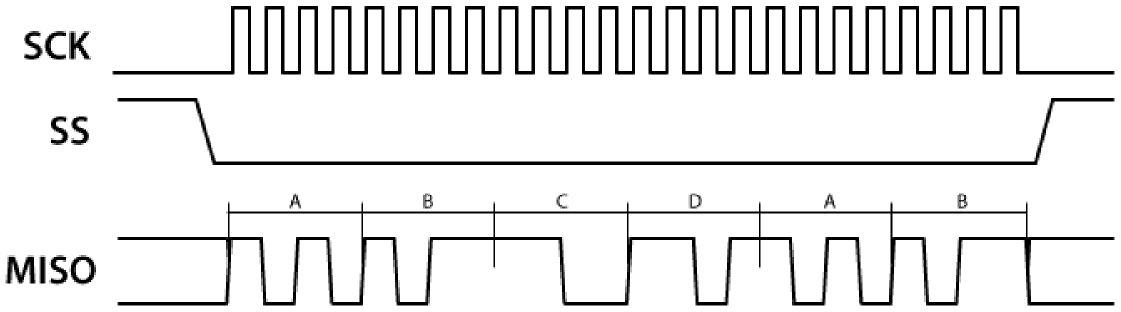
Example

You have a loaded reply: `ABCD`. The master initiates a transmission of 6 words. From the slave, the master receives the following:

- If the **DLN_SPI_SLAVE_REPLY_SHORTAGE_SEND_ZEROES** reply type is set, the master receives ABCD00. The last bytes are filled with zeroes.



- If the **DLN_SPI_SLAVE_REPLY_SHORTAGE_REUSE** reply type is set, the master receives ABCDAB. The last bytes are filled by repeating the first bytes of the reply.



5.5.6 SPI Slave Events

DLN adapters-slaves can be configured to send events. The events are generated when the slave meets the certain predefined conditions.

DLN adapters-slaves support the following events:

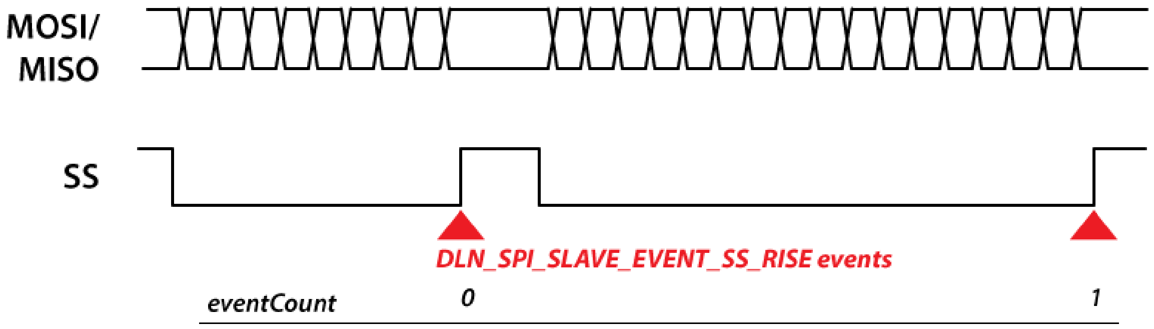
Event Type	Description
DLN_SPI_SLAVE_EVENT_NONE	A DLN adapter does not generate any events.
DLN_SPI_SLAVE_EVENT_SS_RISE	A DLN adapter generates events when the level on the SS line rises. For details, read DLN_SPI_SLAVE_EVENT_SS_RISE Events.
DLN_SPI_SLAVE_EVENT_BUFFER_FULL	A DLN adapter generates events when the buffer is full. For details, read DLN_SPI_SLAVE_EVENT_BUFFER_FULL Events.
DLN_SPI_SLAVE_EVENT_IDLE	A DLN adapter generates events when the slave idles for the configured time. For details, read DLN_SPI_SLAVE_EVENT_IDLE Events.

When an event occurs, your application is notified (see the Notifications section). Call the **DlnGetMessage()** function to obtain the event details. The DLN_SPI_SLAVE_DATA_RECEIVED_EV structure describes this information.

By default, event generation is disabled (DLN_SPI_SLAVE_EVENT_NONE).

DLN_SPI_SLAVE_EVENT_SS_RISE Events

A DLN adapter generates the `DLN_SPI_SLAVE_EVENT_SS_RISE` events each time the level on the SS line rises.



When the transmission from/to the SPI master completes, the master rises the level on the SS line. At this moment, a DLN adapter-slave generates the `DLN_SPI_SLAVE_EVENT_SS_RISE` event. After receiving this event, your application can perform certain actions, for example, show received data or start processing them.

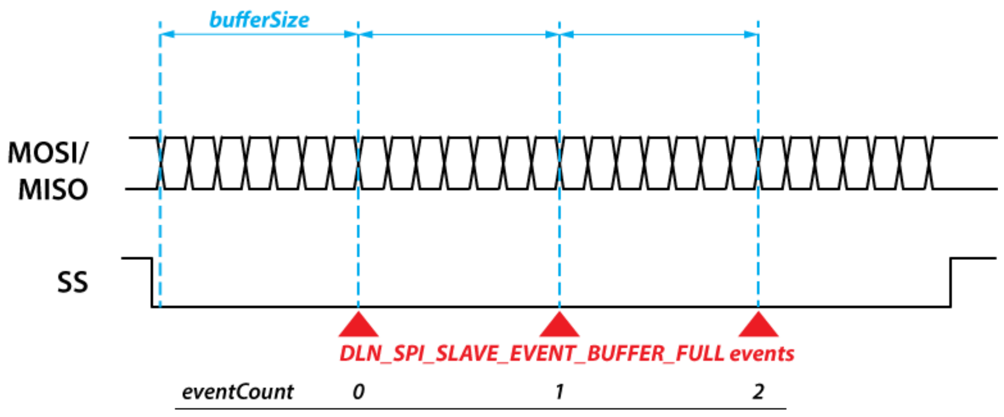
The **eventCount** field of the first event is set to zero. For every new `DLN_SPI_SLAVE_EVENT_SS_RISE` event, the **eventCount** field increments. When you reset the slave or change its configuration, the **eventCount** filed is reset to zero.

Use the `DlnSpiSlaveEnableSSRiseEvent()` function to enable generating these events. To disable this event, use the `DlnSpiSlaveDisableSSRiseEvent()` function. The `DlnSpiSlaveIsSSRiseEventEnabled()` function allows to check whether this event is enabled.

Note: The DLN_SPI_SLAVE_EVENT_SS_RISE and DLN_SPI_SLAVE_EVENT_IDLE events are conflicting; therefore when you enable one of these events, check that the other one is disabled.

DLN_SPI_SLAVE_EVENT_BUFFER_FULL Events

A DLN adapter generates the `DLN_SPI_SLAVE_EVENT_BUFFER_FULL` events each time the buffer becomes full.



You can configure the buffer size by calling the `DlnSpiSlaveSetEventSize()` function. The supported values are from 1 to 256 bytes. By default, the size parameter is set to 256 bytes. To check the current value, use the `DlnSpiSlaveGetEventSize()` function.

During the transmission from the SPI master, if the buffer is overfilled, a DLN adapter-slave generates the `DLN_SPI_SLAVE_EVENT_BUFFER_FULL` event.

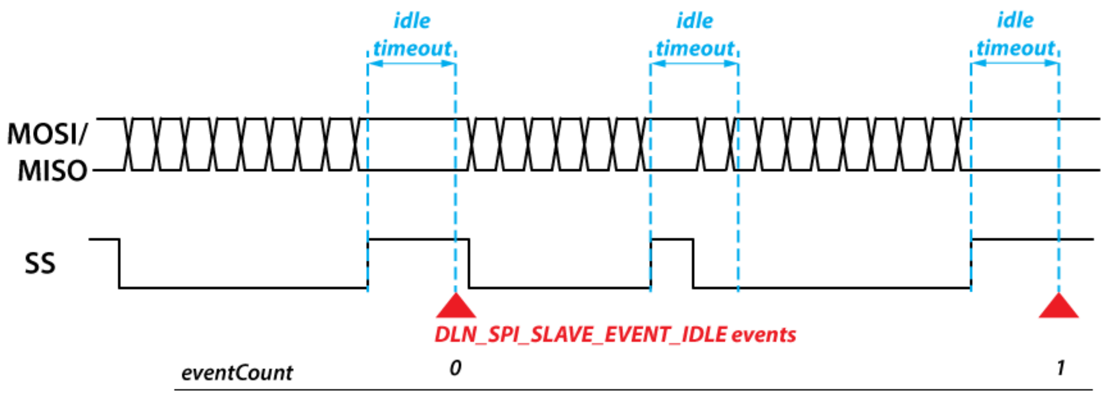
The **eventCount** field of the first event is set to zero. For every new `DLN_SPI_SLAVE_EVENT_SS_RISE` event, the **eventCount** field increments. When you reset the slave or change its configuration, the **eventCount** filed is reset to zero.

Use the `DlnSpiSlaveEnableEvent()` function to enable generating these events. To disable this event, use the `DlnSpiSlaveDisableEvent()` function. The `DlnSpiSlaveIsEventEnabled()` function allows to check whether this event is enabled.

The `DLN_SPI_SLAVE_EVENT_BUFFER_FULL` event does not conflict with other events.

DLN_SPI_SLAVE_EVENT_IDLE Events

A DLN adapter generates the `DLN_SPI_SLAVE_EVENT_IDLE` events each time the SS line stays raised for a certain time.



When the transmission from/to the SPI master completes, the master rises the level on the SS line. If the level stays high for the time set in the timeout parameter, a DLN adapter-slave generates the `DLN_SPI_SLAVE_EVENT_IDLE` event. If the SS line stays in the high level less than the required idle timeout, the event does not occur.

To configure the timeout parameter, use the `DlnSpiSlaveSetIdleEventTimeout()` function. The `DlnSpiSlaveSetIdleEventTimeout()` function allows to check the current value of this parameter.

Use the `DlnSpiSlaveEnableIdleEvent()` function to enable generating `DLN_SPI_SLAVE_EVENT_IDLE` events. To disable this event, use the `DlnSpiSlaveDisableIdleEvent()` function. The `DlnSpiSlaveIsIdleEventEnabled()` function allows to check whether this event is enabled.

5.5.7 SPI Slave Functions

The default configuration for the SPI slave port is as 8-bit SPI slave with CPOL=0 and CPHA=0 transmission parameters. By default, no events are generated.

Using the SPI slave functions allows you to change and check the current SPI slave configuration, to control data transmission.

The SPI slave functions include the following:

General port information:

`DlnSpiSlaveGetPortCount()`
: Retrieves the number of SPI slave ports available in the DLN adapter.

`DlnSpiSlaveEnable()`
: Assigns the selected port to the SPI slave module.

`DlnSpiSlaveDisable()`
: Releases the selected SPI slave port.

`DlnSpiSlaveIsEnabled()`
: Retrieves whether the selected port is assigned to the SPI slave module.

Configuration functions:

`DlnSpiSlaveSetSSIdleTimeout()`
: Specifies idle timeout for releasing the SS line.

`DlnSpiSlaveGetSSIdleTimeout()`
: Retrieves the idle timeout setting.

`DlnSpiSlaveSetFrameSize()`
: Specifies the data frame size for the selected SPI slave port.

`DlnSpiSlaveGetFrameSize()`
: Retrieves the current frame size for the selected SPI slave port.

`DlnSpiSlaveGetSupportedFrameSizes()`
: Retrieves the supported frame sizes.

`DlnSpiSlaveSetMode()`
: Specifies transmission parameters (CPOL and CPHA) for the selected SPI slave port.

`DlnSpiSlaveGetMode()`
: Retrieves the current transmission parameters (CPOL and CPHA) for the selected SPI slave port.

`DlnSpiSlaveGetSupportedModes()`
: Retrieves the supported transmission parameters (CPOL and CPHA).

`DlnSpiSlaveSetCpha()`
: Specifies CPHA value for the selected SPI slave port.

`DlnSpiSlaveGetCpha()`
: Retrieves the current CPHA value for the selected SPI slave port.

`DlnSpiSlaveGetSupportedCphaValues()`
: Retrieves the supported CPHA values.

`DlnSpiSlaveSetCpol()`

Specifies CPOL value for the selected SPI slave port.

`DlnSpiSlaveGetCpol()`
Retrieves the current CPOL value for the selected SPI slave port.

`DlnSpiSlaveGetSupportedCpolValues()`
Retrieves the supported CPOL values.

Reply functions:

`DlnSpiSlaveLoadReply()`
Clears the buffer and makes the current reply the first in the queue.

`DlnSpiSlaveEnqueueReply()`
Puts the current reply to the end of the queue.

`DlnSpiSlaveSetReplyMode()`
Specifies actions if the reply buffer size exceeds the size of the buffer received by the master.

`DlnSpiSlaveGetReplyMode()`
Retrieves the current reply configuration.

`DlnSpiSlaveGetSupportedReplyModes()`
Retrieves the supported reply modes.

`DlnSpiSlaveSetReplyShortageAction()`
Specifies actions if the reply buffer size is less than the size of the buffer received by the master.

`DlnSpiSlaveGetReplyShortageAction()`
Retrieves the current for reply shortage action settings.

`DlnSpiSlaveGetSupportedShortageActions()`
Retrieves the supported reply shortage actions.

Event configuration:

`DlnSpiSlaveEnableSSRiseEvent()`
Enables generating events when the level on the SS line rises.

`DlnSpiSlaveDisableSSRiseEvent()`
Disables generating events when the level on the SS line rises.

`DlnSpiSlaveIsSSRiseEventEnabled()`
Retrieves the current configuration for generating events when the level on the SS line rises.

`DlnSpiSlaveEnableEvent()`
Enables generating events when the buffer is full.

`DlnSpiSlaveDisableEvent()`
Disables generating events when the buffer is full.

`DlnSpiSlaveIsEventEnabled()`
Retrieves the current configuration for generating events when the buffer is full.

`DlnSpiSlaveSetEventSize()`

Specifies the buffer size.

DlnSpiSlaveGetEventSize()

Retrieves the current buffer size.

DlnSpiSlaveEnableIdleEvent()

Enables generating events when the slave idles for the configured time.

DlnSpiSlaveDisableEvent()

Disables generating events when the slave idles for the configured time.

DlnSpiSlaveIsEventEnabled()

Retrieves the current configuration for generating events when the slave idles for the configured time.

DlnSpiSlaveSetIdleEventTimeout()

Specifies the timeout value.

DlnSpiSlaveGetIdleEventTimeout()

Retrieves the current timeout value.

DlnSpiSlaveGetMinIdleEventTimeout()

Retrieves the minimum timeout value.

DlnSpiSlaveGetMaxIdleEventTimeout()

Retrieves the maximum timeout value.

The *dln_spi_slave.h* file declares the SPI Salve Interface functions.

DlnSpiSlaveGetPortCount() Function

The **DlnSpiSlaveGetPortCount()** function retrieves the total number of SPI slave ports available at your DLN-series adapter.

Syntax

```
DLN_RESULT DlnSpiSlaveGetPortCount(
   HDLN handle,
   uint8_t* count
);
```

Parameters

handle

A handle to the DLN-series adapter.

count

A pointer to an unsigned 8-bit integer that receives the number of the available SPI slave ports.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function retrieved the port count successfully.

Remarks

The **`DlnSpiSlaveGetPortCount()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveEnable() Function

The **`DlnSpiSlaveEnable()`** function activates the specified SPI slave port at your DLN-series adapter.

Syntax

```
DLN_RESULT DlnSpiSlaveEnable(
   HDLN handle,
   uint8_t port,
   uint16_t* conflict
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

conflict

A pointer to an unsigned 16-bit integer that receives a number of the conflicted pin, if any.

A conflict arises if a pin is already assigned to another module of the DLN-series adapter and cannot be used by the SPI slave module. To fix this, check which module uses the pin (call the **`DlnGetPinCfg()`** function), disconnect the pin from that module and call the **`DlnSpiSlaveEnable()`** function once again. If there is another conflicting pin, its number will be returned.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function enabled the SPI slave module successfully.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiSlaveGetPortCount()`** function to find the maximum possible port number.

DLN_RES_PIN_IN_USE (0xA5)

The port cannot be activated as the SPI slave port because one or more pins of the port are assigned to another module. The **conflict** parameter contains the number of a conflicting pin.

DLN_RES_SPI_SLAVE_SS_IDLE_TIMEOUT (0xCB)

The port cannot be activated as the SPI slave port because its SS line is busy transmitting some data and was not released for time defined by the **`DlnSpiSlaveSetSSIdleTimeout()`** function.

Remarks

The **`DlnSpiSlaveEnable()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveDisable() Function

The **`DlnSpiSlaveDisable()`** function deactivates the specified SPI slave port on your DLN adapter.

Syntax

```
DLN_RESULT DlnSpiSlaveDisable(
   HDLN handle,
   uint8_t port,
   uint8_t waitForTransferCompletion
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

waitForTransferCompletion

Used to choose whether the device should wait for current data transmissions complete before disabling as SPI slave. The following values are available:

- 1 or **`DLN_SPI_SLAVE_WAIT_FOR_TRANSFERS`** - wait until transmissions complete.
- 0 or **`DLN_SPI_SLAVE_CANCEL_TRANSFERS`** - cancel all pending data transmissions and release the port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function disabled the SPI slave module successfully.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiSlaveGetPortCount()`** function to find the maximum possible port number.

DLN_RES_TRANSFER_CANCELLED (0x20)

The pending transmissions were cancelled.

Remarks

The **`DlnSpiSlaveDisable()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveIsEnabled() Function

The **`DlnSpiSlaveIsEnabled()`** function retrieves information whether the specified SPI slave port is active or not.

Syntax

```
DLN_RESULT DlnSpiSlaveIsEnabled(
   HDLN handle,
   uint8_t port,
   uint8_t* enabled
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

enabled

A pointer to an unsigned 8-bit integer that receives information whether the specified SPI slave port is activated after the function execution. There are two possible values:

- 0 or **`DLN_SPI_SLAVE_DISABLED`** - the port is not configured as a SPI slave.
- 1 or **`DLN_SPI_SLAVE_ENABLED`** - the port is configured as a SPI slave.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved information about the port.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiSlaveGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiSlaveIsEnabled()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveSetSSIdleTimeout() Function

The **`DlnSpiSlaveSetSSIdleTimeout()`** function sets SS idle timeout. For details, read SS Idle Timeout.

Syntax

```
DLN_RESULT DlnSpiSlaveSetSSIdleTimeout(
    HDLN handle,
    uint8_t port,
    uint32_t timeout
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

timeout

The SS idle timeout value specified in milliseconds (ms). The minimum value is 1ms, the maximum value - 1000ms.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured SS idle timeout.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiSlaveGetPortCount()` function to find the maximum possible port number.

DLN_RES_INVALID_TIMEOUT_VALUE (0xCA)

The SS idle timeout value is not valid.

Remarks

The default SS idle timeout value is set to 100ms.

The **`DlnSpiSlaveSetSSIdleTimeout()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveGetSSIdleTimeout() Function

The **`DlnSpiSlaveGetSSIdleTimeout()`** function retrieves the current value of SS idle timeout.

Syntax

```
DLN_RESULT DlnSpiSlaveGetSSIdleTimeout(
    HDLN handle,
    uint8_t port,
    uint32_t *timeout
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

timeout

A pointer to an unsigned 32 bit integer that receives the current SS idle timeout value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the current SS idle timeout value.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiSlaveGetPortCount()` function to find the maximum possible port number.

Remarks

The **`DlnSpiSlaveGetSSIdleTimeout()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveSetFrameSize() Function

The **`DlnSpiSlaveSetFrameSize()`** function sets the size of a single SPI data frame.

Syntax

```
DLN_RESULT DlnSpiSlaveSetFrameSize(
   HDLN handle,
   uint8_t port,
   uint8_t frameSize
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

frameSize

A number of bits to be transmitted during the single frame. The DLN-series adapter supports 8 to 16 bits per frame.

The **frameSize** parameter does not limit the size of the buffer transmitted to/from the SPI slave device, it only defines the minimum portion of data in this buffer.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully specified the frame size.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiSlaveGetPortCount()` function to find the maximum possible port number.

DLN_RES_SPI_INVALID_FRAME_SIZE (0xB8)

The frame size is not valid. The DLN-series adapters support 8 to 16 bits per transmission.

DLN_RES_BUSY (0xB6)

The SPI slave is busy transmitting. The frame size cannot be changed.

Remarks

The **`DlnSpiSlaveSetFrameSize()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveGetFrameSize() Function

The **`DlnSpiSlaveGetFrameSize()`** function retrieves the current size setting for SPI data frames.

Syntax

```
DLN_RESULT DlnSpiSlaveGetFrameSize(
   HDLN handle,
   uint8_t port,
   uint8_t* frameSize
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

frameSize

A pointer to unsigned 8-bit integer that receives the number of bits transmitted in a single frame. The DLN-series adapter supports 8 to 16 bits per frame.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the current frame size.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiSlaveGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiSlaveGetFrameSize()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveGetSupportedFrameSizes() Function

The **`DlnSpiSlaveGetSupportedFrameSizes()`** function returns all supported frame size values.

Syntax

```
DLN_RESULT DlnSpiSlaveGetSupportedFrameSizes(
    HDLN handle,
    uint8_t port,
    DLN_SPI_SLAVE_FRAME_SIZES *supportedSizes
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

supportedSizes

The pointer to the `DLN_SPI_SLAVE_FRAME_SIZES` structure that receives supported frame size values.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved supported frame size values.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiSlaveGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiSlaveGetSupportedFrameSizes()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveSetMode() Function

The **`DlnSpiSlaveSetMode()`** function sets SPI transmission parameters (CPOL and CPHA). For more information, read Clock Phase and Polarity.

Syntax

```
DLN_RESULT DlnSpiSlaveSetMode(
    HDLN handle,
    uint8_t port,
    uint8_t mode
 );
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

mode

A bit field consisting of 8 bits. The bits 0 and 1 correspond to CPOL and CPHA parameters respectively and define the SPI mode. The rest of the bits are not used. You can also use special constants, defined in the *dln_spi_slave.h* file for each of the bits.

Bit	Value	Description	Constant
0	0	CPOL=0	DLN_SPI_SLAVE_CPOL_0
0	1	CPOL=1	DLN_SPI_SLAVE_CPOL_1
1	0	CPHA=0	DLN_SPI_SLAVE_CPHA_0
1	1	CPHA=1	DLN_SPI_SLAVE_CPHA_1

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured the slave port mode.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiSlaveGetPortCount()` function to find the maximum possible port number.

DLN_RES_INVALID_MODE (0xC7)

The mode value is not valid. Use the `DlnSpiSlaveGetSupportedModes()` function to check the supported modes for the current SPI slave port.

DLN_RES_MUST_BE_DISABLED (0x95)

These configuration changes are allowed only when the module is disabled. Use the `DlnSpiSlaveDisable()` function to disable the SPI module before configuring the slave port.

Remarks

The **`DlnSpiSlaveSetMode()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveGetMode() Function

The **`DlnSpiSlaveGetMode()`** function retrieves the current transmission parameters (CPOL and CPHA) for the specified SPI slave port. For details, read Clock Phase and Polarity.

Syntax

```
DLN_RESULT DlnSpiSlaveGetMode(
   HDLN handle,
   uint8_t port,
   uint8_t* mode
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

mode

A pointer to an unsigned 8 bit integer that receives the SPI mode value. The bits 0 and 1 of this value correspond to CPOL and CPHA parameters respectively. The rest of the bits are not used.

Bit	Value	Description	Constant
0	0	CPOL=0	DLN_SPI_SLAVE_CPOL_0
0	1	CPOL=1	DLN_SPI_SLAVE_CPOL_1
1	0	CPHA=0	DLN_SPI_SLAVE_CPHA_0
1	1	CPHA=1	DLN_SPI_SLAVE_CPHA_1

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured the slave port mode.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiSlaveGetPortCount()` function to find the maximum possible port number.

Remarks

The `DlnSpiSlaveGetMode()` function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveGetSupportedModes() Function

The `DlnSpiSlaveGetSupportedModes()` function retrieves all supported SPI slave modes.

Syntax

```
DLN_RESULT DlnSpiSlaveGetSupportedModes(
    HDLN handle,
    uint8_t port,
    DLN_SPI_SLAVE_MODE_VALUES *values
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

values

The pointer to `DLN_SPI_SLAVE_MODE_VALUES` structure that receives all supported SPI slave mode values.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the supported reply modes.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiSlaveGetPortCount()` function to find the maximum possible port number.

Remarks

The `DlnSpiSlaveGetSupportedModes()` function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveSetCpha() Function

The `DlnSpiSlaveSetCpha()` function allows to set the clock phase (CPHA) value.

Syntax

```
DLN_RESULT DlnSpiSlaveSetCpha(
    HDLN handle,
    uint8_t port,
    uint8_t cpha
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

cpha

The clock phase (CPHA) value. Can be 0 or 1.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured the slave port mode.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiSlaveGetPortCount()` function to find the maximum possible port number.

DLN_RES_MUST_BE_DISABLED (0x95)

These configuration changes are allowed only when the module is disabled. Use the `DlnSpiSlaveDisable()` function to disable the SPI module before configuring the slave port.

DLN_RES_INVALID_CPHA (0xC9)

The provided CPHA value is not valid. Use the

`DlnSpiSlaveGetSupportedCphaValues()` function to check the supported CPHA values for the current SPI slave port.

Remarks

The **`DlnSpiSlaveSetCpha()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveGetCpha() Function

The **`DlnSpiSlaveGetCpha()`** function retrieves the current value of clock phase (CPHA).

Syntax

```
DLN_RESULT DlnSpiSlaveGetCpha(
    HDLN handle,
    uint8_t port,
    uint8_t *cpha
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

cpha

The pointer to uint8_t variable that receives the current CPHA value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured the slave port mode.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiSlaveGetPortCount()` function to find the maximum possible port number.

Remarks

The **`DlnSpiSlaveGetCpha()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveGetSupportedCphaValues() Function

The **`DlnSpiSlaveGetSupportedCphaValues()`** function retrieves all supported CPHA (clock phase) values.

Syntax

```
DLN_RESULT DlnSpiSlaveGetSupportedCphaValues(
   HDLN handle,
   uint8_t port,
   DLN_SPI_SLAVE_CPHA_VALUES *values
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

values

The pointer to `DLN_SPI_SLAVE_CPHA_VALUES` structure that receives all supported SPI slave CPHA values.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured the slave port mode.

Remarks

The **`DlnSpiSlaveGetSupportedCphaValues()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveSetCpol() Function

The **`DlnSpiSlaveSetCpol()`** function allows to set the clock polarity (CPOL) value.

Syntax

```
DLN_RESULT DlnSpiSlaveSetCpol(
    HDLN handle,
    uint8_t port,
    uint8_t cpol
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

cpol

The clock polarity (CPOL) value. Can be 0 or 1.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured the slave port mode.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiSlaveGetPortCount()` function to find the maximum possible port number.

DLN_RES_MUST_BE_DISABLED (0x95)

These configuration changes are allowed only when the module is disabled. Use the `DlnSpiSlaveDisable()` function to disable the SPI module before configuring the slave port.

DLN_RES_INVALID_CPOL (0xC8)

The provided CPOL value is not valid. Use the `DlnSpiSlaveGetSupportedCpolValues()` function to check the supported CPOL values for the current SPI slave port.

Remarks

The **`DlnSpiSlaveSetCpol()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveGetCpol() Function

The **`DlnSpiSlaveGetCpol()`** function retrieves the current value of clock polarity (CPOL).

Syntax

```
DLN_RESULT DlnSpiSlaveGetCpol(
    HDLN handle,
    uint8_t port,
    uint8_t *cpol
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI-slave port.

cpol

The pointer to uint8_t variable that receives the current CPOL value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured the slave port mode.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiSlaveGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiSlaveGetCpol()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveGetSupportedCpolValues() Function

The **`DlnSpiSlaveGetSupportedCpolValues()`** function retrieves all supported CPOL (clock polarity) values.

Syntax

```
DLN_RESULT DlnSpiSlaveGetSupportedCpolValues(
    HDLN handle,
    uint8_t port,
    DLN_SPI_SLAVE_CPOL_VALUES *values
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

values

The pointer to **`DLN_SPI_SLAVE_CPOL_VALUES`** structure that receives all supported CPOL values.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured the slave port mode.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiSlaveGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiSlaveGetSupportedCpolValues()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveLoadReply() Function

The **`DlnSpiSlaveLoadReply()`** function loads data (loads a number of bytes to reply buffer) to be transmitted to an SPI-master device. This function clears the buffer and makes the current reply the first in the queue.

Syntax

```
DLN_RESULT DlnSpiSlaveLoadReply(
   HDLN handle,
   uint8_t port,
   uint16_t size,
   uint8_t* buffer
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

size

A size of the message buffer. This parameter is specified in bytes. The maximum value is 256 bytes.

buffer

A pointer to an array of unsigned 8-bit integers (max 256-elements array). The buffer must contain the data to send to the SPI bus master.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully loaded data to reply buffer.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiSlaveGetPortCount()`** function to find the maximum possible port number.

DLN_RES_INVALID_BUFFER_SIZE (0xAE)

The buffer size is not valid. The DLN-series adapters support up to 256 bytes of the buffer.

DLN_RES_BUSY (0xB6)

The SPI slave is busy transmitting. Data cannot be loaded to the buffer.

Remarks

The **`DlnSpiSlaveLoadReply()`** function removes all replies from the queue and makes the current reply the first in the queue. In contrast, the **`DlnSpiSlaveEnqueueReply()`** function adds the reply to the queue.

The type of reply is configurable: it can be based on the SS line or based on the byte count. For details, read Reply Modes. To configure the reply mode, use the **`DlnSpiSlaveSetReplyMode()`** function.

The **`DlnSpiSlaveLoadReply()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveEnqueueReply() Function

The **`DlnSpiSlaveEnqueueReply()`** function adds a buffer to queue.

Syntax

```
DLN_RESULT DlnSpiSlaveEnqueueReply(
    HDLN handle,
    uint8_t port,
    uint16_t size,
    uint8_t *buffer
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI-slave port.

size

A size of the message buffer. This parameter is specified in bytes. The maximum value is 256 bytes.

buffer

A pointer to an array of unsigned 8-bit integers (max 256-elements array). The buffer must contain the data to be sent to an SPI bus master.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully added data to reply queue.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiSlaveGetPortCount()`** function to find the maximum possible port number.

DLN_RES_INVALID_BUFFER_SIZE (0xAE)

The buffer size is not valid. The DLN-series adapters support up to 256 bytes of the buffer.

Remarks

If the queue is empty (after enabling), the **`DlnSpiSlaveEnqueueReply()`** function is equal to the **`DlnSpiSlaveLoadReply()`** function. If the queue is not empty, the **`DlnSpiSlaveEnqueueReply()`** function does not remove replies from the queue.

The **`DlnSpiSlaveEnqueueReply()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveSetReplyMode() Function

The **`DlnSpiSlaveSetReplyMode()`** function sets SPI slave reply mode.

Syntax

```
DLN_RESULT DlnSpiSlaveSetReplyMode(
    HDLN handle,
    uint8_t port,
    uint8_t replyMode
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

replyMode

Reply mode value. The following values are possible:

- 0 or **`DLN_SPI_SLAVE_REPLY_MODE_OFF`** - No reply, transmitter is disabled.
- 1 or **`DLN_SPI_SLAVE_REPLY_MODE_SS_BASED`** - Transmitting the reply stops when the SS line releases. If the reply was not transmitted completely, the rest part of the reply is lost. The next reply transmission starts after the new SS line assertion.
- 2 or **`DLN_SPI_SLAVE_REPLY_MODE_COUNT_BASED`** - Only after the reply transmission is completed, the next reply can start. Transmission occurs regardless of the SS line state.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured the reply mode.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiSlaveGetPortCount()`** function to find the maximum possible port number.

DLN_RES_INVALID_REPLY_TYPE (0xC5)

The specified reply type is not valid.

Remarks

The **`DLN_SPI_SLAVE_REPLY_MODE_COUNT_BASED`** reply mode allows to send data regardless of how many (or few) words are framed by the SS line. The **`DLN_SPI_SLAVE_REPLY_MODE_SS_BASED`** reply mode is set by default, it begins new data transmission/reply regardless of the data length. The **`DLN_SPI_SLAVE_REPLY_MODE_OFF`** mode disables transmission, it can be set for half-duplex write communication (for details, read SPI Data Transmission).

The **`DlnSpiSlaveSetReplyMode()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveGetReplyMode() Function

The **`DlnSpiSlaveGetReplyMode()`** function retrieves the current SPI slave reply mode.

Syntax

```
DLN_RESULT DlnSpiSlaveGetReplyMode(
    HDLN handle,
    uint8_t port,
    uint8_t *replyMode
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

replyMode

The pointer to an unsigned 8 bit integer that receives the SPI reply mode value after the function execution.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the current reply mode.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiSlaveGetPortCount()` function to find the maximum possible port number.

Remarks

The **`DlnSpiSlaveGetReplyMode()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveGetSupportedReplyModes() Function

The **`DlnSpiSlaveGetSupportedReplyModes()`** function retrieves the supported SPI slave reply modes.

Syntax

```
DLN_RESULT DlnSpiSlaveGetReplyMode(
    HDLN handle,
    uint8_t port,
    DLN_SPI_SLAVE_REPLY_MODES *supportedReplyModes
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

supportedReplyModes

The pointer to `DLN_SPI_SLAVE_REPLY_MODES` structure that receives all supported reply modes.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the supported reply mode.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiSlaveGetPortCount()` function to find the maximum possible port number.

Remarks

The `DlnSpiSlaveGetSupportedReplyModes()` function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveSetReplyShortageAction() Function

The `DlnSpiSlaveSetReplyShortageAction()` function sets the reply shortage action value. When the master sends a data block that is larger than the loaded reply, this function specifies how to fill empty bytes (repeat the reply bytes or send zeroes).

Syntax

```
DLN_RESULT DlnSpiSlaveSetReplyShortageAction(
    HDLN handle,
    uint8_t port,
    uint8_t action
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

action

Shortage action value to be set. There are 2 possible values:

- `DLN_SPI_SLAVE_REPLY_SHORTAGE_SEND_ZEROES` - To fill the missed bytes with 0.
- `DLN_SPI_SLAVE_REPLY_SHORTAGE_REUSE` - To fill the missed bytes repeating the reply.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured the shortage action.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiSlaveGetPortCount()`** function to find the maximum possible port number.

DLN_RES_BUSY (0xB6)

The SPI slave is busy transmitting. The reply shortage action cannot be changed. To change configuration, use the **`DlnSpiSlaveDisable()`** function.

Remarks

When the master sends a data block that is larger than the loaded reply, this function specifies how to fill empty bytes (repeat the reply bytes or send zeroes). For example, the loaded reply includes 3 bytes (`11 22 33`), but the master sends 5 bytes. If the function sets the **`DLN_SPI_SLAVE_REPLY_SHORTAGE_SEND_ZEROES`** value, the slave sends `11 22 33 00 00`. If the function sets the **`DLN_SPI_SLAVE_REPLY_SHORTAGE_REUSE`** value, the slave sends `11 22 33 11 22`. For details, read Reply Shortage Actions.

The **`DlnSpiSlaveSetReplyShortageAction()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveGetReplyShortageAction() Function

The **`DlnSpiSlaveGetReplyShortageAction()`** functions retrieves the current value of reply shortage action.

Syntax

```
DLN_RESULT DlnSpiSlaveGetReplyShortageAction(
    HDLN handle,
    uint8_t port,
    uint8_t *action
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

action

A pointer to unsigned 8-bit integer that receives the current reply shortage configuration.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the current shortage configuration.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiSlaveGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiSlaveGetReplyShortageAction()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveGetSupportedShortageActions() Function

The **`DlnSpiSlaveGetSupportedShortageActions()`** function retrieves supported shortage action values.

Syntax

```
DLN_RESULT DlnSpiSlaveGetSupportedShortageActions(
    HDLN handle,
    uint8_t port,
    DLN_SPI_SLAVE_SHORTAGE_ACTIONS *supportedSizes
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI-slave port.

supportedSizes

The pointer to **`DLN_SPI_SLAVE_SHORTAGE_ACTIONS`** structure that receives all the supported shortage action values.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the supported shortage action values.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiSlaveGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiSlaveGetSupportedShortageActions()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveEnableSSRiseEvent() Function

The **`DlnSpiSlaveEnableSSRiseEvent()`** function activates event on SS rising edge.

Syntax

```
DLN_RESULT DlnSpiSlaveEnableSSRiseEvent(
    HDLN handle,
    uint8_t port
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully activated the event on SS rising edge.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiSlaveGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiSlaveEnableSSRiseEvent()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveDisableSSRiseEvent() Function

The **`DlnSpiSlaveDisableSSRiseEvent()`** function deactivates SS rise events.

Syntax

```
DLN_RESULT DlnSpiSlaveDisableSSRiseEvent(
    HDLN handle,
    uint8_t port
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully deactivated the event on SS rising edge.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiSlaveGetPortCount()` function to find the maximum possible port number.

Remarks

The **`DlnSpiSlaveDisableSSRiseEvent()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveIsSSRiseEventEnabled() Function

The **`DlnSpiSlaveIsSSRiseEventEnabled()`** function retrieves information whether the SS rise events are active or not.

Syntax

```
DLN_RESULT DlnSpiSlaveIsSSRiseEventEnabled(
    HDLN handle,
    uint8_t port,
    uint8_t *enabled
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port to retrieve the information from.

enabled

A pointer to an unsigned 8-bit integer that receives information whether the SS rise events are enabled.

There are two possible values:

- 0 or **`DLN_SPI_SLAVE_EVENT_DISABLED`** - SS rise events are not enabled.
- 1 or **`DLN_SPI_SLAVE_EVENT_ENABLED`** - SS rise events are enabled.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the event on SS rising edge configuration.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiSlaveGetPortCount()` function to find the maximum possible port number.

Remarks

The **`DlnSpiSlaveIsSSRiseEventEnabled()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveEnableEvent() Function

The **`DlnSpiSlaveEnableEvent()`** function activates SPI slave buffer full event.

Syntax

```
DLN_RESULT DlnSpiSlaveEnableEvent(
    HDLN handle,
    uint8_t port
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully activated the SPI slave buffer full event.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiSlaveGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiSlaveEnableEvent()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveDisableEvent() Function

The **`DlnSpiSlaveDisableEvent()`** function deactivates SPI slave buffer full event.

Syntax

```
DLN_RESULT DlnSpiSlaveDisableEvent(
    HDLN handle,
    uint8_t port
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully deactivated the SPI slave buffer full event.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **DlnSpiSlaveGetPortCount()** function to find the maximum possible port number.

Remarks

The **DlnSpiSlaveDisableEvent()** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveIsEventEnabled() Function

The **DlnSpiSlaveIsEventEnabled()** function retrieves information whether the SPI slave events are active or not.

Syntax

```
DLN_RESULT DlnSpiSlaveIsEventEnabled(
    HDLN handle,
    uint8_t port,
    uint8_t *enabled
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port to retrieve the information from.

enabled

A pointer to an unsigned 8-bit integer that receives information whether the SPI slave port events are activated or not. There are two possible values:

- 0 or **DLN_SPI_SLAVE_EVENT_DISABLED** - SPI slave events are enabled.
- 1 or **DLN_SPI_SLAVE_EVENT_ENABLED** - SPI slave events are disabled.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieves the SPI slave buffer full event.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **DlnSpiSlaveGetPortCount()** function to find the maximum possible port number.

Remarks

The **DlnSpiSlaveIsEventEnabled()** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveSetEventSize() Function

The **DlnSpiSlaveSetEventSize()** function sets the event buffer size.

Syntax

```
DLN_RESULT DlnSpiSlaveSetEventSize(
    HDLN handle,
    uint8_t port,
    uint16_t size
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port to be configured.

size

Event buffer size parameter. Its value can vary from 1 to 256 bytes.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured the event buffer size.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiSlaveGetPortCount()`** function to find the maximum possible port number.

DLN_RES_INVALID_BUFFER_SIZE (0xAE)

The buffer size is not valid. The DLN-series adapters support up to 256 bytes of the buffer.

Remarks

The **`DlnSpiSlaveSetEventSize()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveGetEventSize() Function

The **`DlnSpiSlaveGetEventSize()`** function retrieves value of current event buffer size.

Syntax

```
DLN_RESULT DlnSpiSlaveGetEventSize(
    HDLN handle,
    uint8_t port,
    uint16_t *size
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port to retrieve the SPI buffer size value from.

size

A pointer to an unsigned 16 bit integer that receives the current event buffer size value. The value varies from 1 to 256.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the current event buffer size.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiSlaveGetPortCount()` function to find the maximum possible port number.

Remarks

The `DlnSpiSlaveGetEventSize()` function is defined in *dln_spi_slave.h* file.

DlnSpiSlaveEnableIdleEvent() Function

The `DlnSpiSlaveEnableIdleEvent()` function activates the SPI slave idle event.

Syntax

```
DLN_RESULT DlnSpiSlaveEnableIdleEvent(
    HDLN handle,
    uint8_t port
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully activated the SPI slave idle event.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiSlaveGetPortCount()` function to find the maximum possible port number.

DLN_RES_INVALID_EVENT_TYPE (0xA9)

The idle event conflicts with the SS rise event. To enable the idle event, disable the SS rise event by using the `DlnSpiSlaveDisableSSRiseEvent()` function.

Remarks

The `DlnSpiSlaveEnableIdleEvent()` function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveDisableIdleEvent() Function

The **`DlnSpiSlaveDisableIdleEvent()`** function deactivates the SPI slave idle event.

Syntax

```
DLN_RESULT DlnSpiSlaveDisableIdleEvent(
    HDLN handle,
    uint8_t port
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully deactivated the SPI slave idle event.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiSlaveGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiSlaveDisableIdleEvent()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveIsIdleEventEnabled() Function

The **`DlnSpiSlaveIsIdleEventEnabled()`** function activates the SPI slave idle event.

Syntax

```
DLN_RESULT DlnSpiSlaveIsIdleEventEnabled(
    HDLN handle,
    uint8_t port,
    uint8_t *enabled
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

enabled

A pointer to an unsigned 8-bit integer that receives information whether the idle events are activated or not.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully activated the SPI slave idle event.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiSlaveGetPortCount()` function to find the maximum possible port number.

Remarks

The **`DlnSpiSlaveIsIdleEventEnabled()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveSetIdleEventTimeout() Function

The **`DlnSpiSlaveSetIdleEventTimeout()`** function sets idle event timeout.

Syntax

```
DLN_RESULT DlnSpiSlaveSetIdleEventTimeout(
    HDLN handle,
    uint8_t port,
    uint32_t timeout
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

timeout

The idle event timeout value specified in milliseconds (ms).

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured idle event timeout.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiSlaveGetPortCount()` function to find the maximum possible port number.

DLN_RES_INVALID_TIMEOUT_VALUE (0xCA)

The idle event timeout value is not valid.

Remarks

The default idle event timeout value is set to 0ms.

The **`DlnSpiSlaveSetIdleEventTimeout()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveGetIdleEventTimeout() Function

The **`DlnSpiSlaveGetIdleEventTimeout()`** function retrieves the current value of idle event timeout.

Syntax

```
DLN_RESULT DlnSpiSlaveGetIdleEventTimeout(
    HDLN handle,
    uint8_t port,
    uint32_t *timeout
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

timeout

A pointer to an unsigned 32 bit integer that receives the current idle event timeout value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the minimum idle event timeout value.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnSpiSlaveGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnSpiSlaveGetIdleEventTimeout()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveGetMinIdleEventTimeout() Function

The **`DlnSpiSlaveGetMinIdleEventTimeout()`** function retrieves the minimum value of idle event timeout.

Syntax

```
DLN_RESULT DlnSpiSlaveGetMinIdleEventTimeout(
    HDLN handle,
    uint8_t port,
    uint32_t *timeout
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

timeout

A pointer to an unsigned 32 bit integer that receives the minimum idle event timeout value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the maximum idle event timeout value.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiSlaveGetPortCount()` function to find the maximum possible port number.

Remarks

The **`DlnSpiSlaveGetMinIdleEventTimeout()`** function is defined in the *dln_spi_slave.h* file.

DlnSpiSlaveGetMaxIdleEventTimeout() Function

The **`DlnSpiSlaveGetMaxIdleEventTimeout()`** function retrieves the maximum value of idle event timeout.

Syntax

```
DLN_RESULT DlnSpiSlaveGetMaxIdleEventTimeout(
    HDLN handle,
    uint8_t port,
    uint32_t *timeout
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI slave port.

timeout

A pointer to an unsigned 32 bit integer that receives the minimum idle event timeout value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the maximum idle event timeout value.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnSpiSlaveGetPortCount()` function to find the maximum possible port number.

Remarks

The `DlnSpiSlaveGetMaxIdleEventTimeout()` function is defined in the *dln_spi_slave.h* file.

6. GPIO Interface

DLN series interface adapters contain a large number of pins that you can use as general-purpose inputs and outputs. Some of these pins are shared between GPIO (general-purpose input/output) and other modules (I2C, SPI). If you do not the corresponding pins to other modules, you can configure them as digital inputs or digital outputs and control them.

If you configure pins as inputs, your application reads the digital value set by the external device. If you define pins as outputs, your application can control the digital value on these pins.

To use a pin as a digital input or a digital output, the pin must be assigned to the GPIO module by using the `DlnGpioPinEnable()` function. You can use the `DlnGpioPinIsEnabled()` function, to check whether a pin is already assigned to the GPIO module.

You cannot assign a pin to the GPIO module if another module uses it. Call the `DlnGetPinCfg()` function to check which module uses a pin.

Most of the DLN-series adapters provide additional functionality for GPIO pins. You can configure the following features for any input and/or output pin individually:

- **Event-driven interface.** You can configure a digital input pin to raise events when the value on this pin changes. There are several configuration options - what kind of events and when should be generated. For more details, read Digital Input Events.
- **Debounce filter.** If you do not want to treat casual pulses as value changes, you can enable the debounce filter, which rejects pulses less that the predefined debounce period. For details, read Debounce Filter.
- **Open drain mode.** If you interconnect several outputs on a single I/O line, they may settle different values simultaneously and cause hardware damage. The open drain mode helps you to avoid this. For details, read Open Drain Mode.
- **Pull-up and pull-down resistors.** When no external hardware is connected to an input line, the logic level on this line is undefined. Use pull-up or pull-down resistors to ensure that inputs to I/O lines are settled at expected levels. For details, read Pull-up/Pull-down Resistors.

6.1 Digital Outputs

You can configure most of the DLN-series adapter pins as general-purpose digital outputs. Call the `DlnGpioPinSetDirection()` function and pass 1 for the **direction** parameter to configure the GPIO pin as a digital output. You can call this function either before or after the pin is assigned to the GPIO module by using the `DlnGpioPinEnable()` function.

To set the value on the pin, use the `DlnGpioPinSetOutVal()` function. Call the `DlnGpioPinGetOutVal()` function to retrieve the output value configured for this pin. The actual value of the pin may differ from the configured one (for example, when the pin is in the open drain mode or it is not assigned to the GPIO module). The `DlnGpioPinGetVal()` function retrieves the actual value present on the pin.

For digital outputs, you can use the Open Drain Mode.

Use the following GPIO Interface functions to control and monitor output pins:

`DlnGpioPinSetDirection()`
: Configures a pin as a digital output. Pass 1 for the **direction** parameter.

`DlnGpioPinSetOutVal()`
: Sets the output value for the pin.

`DlnGpioPinGetOutVal()`
: Retrieves the output value configured for the pin.

`DlnGpioPinGetVal()`
: Retrieves the actual value on the I/O line.

6.2 Digital Inputs

You can configure most of the DLN-series adapter pins as general-purpose digital inputs. Call the `DlnGpioPinSetDirection()` function and pass 0 for the **direction** parameter to configure the GPIO pin as a digital input. You can call this function either before or after the pin is assigned to the GPIO module by using the `DlnGpioPinEnable()` function.

You can check a pin direction by using the `DlnGpioPinGetDirection()` function.

The default pin direction is set to input (value 0).

To check the value on the pin, use the `DlnGpioPinGetVal()` function.

For digital inputs, you can use the following features:

- Digital Input Events;
- Debounce Filter;
- Pull-up/Pull-down Resistors.

Use the following GPIO Interface functions to control and monitor input pins:

`DlnGpioPinSetDirection()`
: Configures a pin as a digital input. Pass 0 for the **direction** parameter.

`DlnGpioPinGetVal()`

Retrieves the value on the I/O line.

6.3 Digital Input Events

There are two ways of monitoring changes on the digital input line:

- To poll the input pin periodically. In this case, you can potentially miss an input if your application reads the value at the wrong time.
- To receive events when the input value changes. You can configure event generation for specific changes, for example, when the level on a pin rises. You can also configure the adapter to generate events periodically.

To configure events, use the `DlnGpioPinSetEventCfg()` function. You need to specify the **eventType** and the **eventPeriod** parameters. The `DlnGpioPinGetSupportedEventTypes()` function returns the list of event types available for a pin.

The **eventType** parameter can have one of the following values:

Event Type	Description
DLN_GPIO_EVENT_NONE	A DLN adapter does not generate any events.
DLN_GPIO_EVENT_CHANGE	A DLN adapter generates events when the level on the digital input line changes. For details, read DLN_GPIO_EVENT_CHANGE Events
DLN_GPIO_EVENT_LEVEL_HIGH	A DLN adapter generates events when the high level is present on the digital input line or after transition from low to high level. For details, read DLN_GPIO_EVENT_LEVEL_HIGH Events
DLN_GPIO_EVENT_LEVEL_LOW	A DLN adapter generates events when the low level is present on the digital input line or after transition from high to low level. For details, read DLN_GPIO_EVENT_LEVEL_LOW Events
DLN_GPIO_EVENT_ALWAYS	A DLN adapter generates events continuously, regardless of the level. For details, read DLN_GPIO_EVENT_ALWAYS Events

The **eventPeriod** parameter defines the interval between repeated events in milliseconds (ms). This parameter must be non-zero for the `DLN_GPIO_EVENT_ALWAYS` events. For other events, this parameter is optional.

If the Debounce Filter in enabled on a pin, the level change event is generated after the debounce filter accepts a new value.

You can configure events for each input pin individually. If the pin is configured as a digital input, the new settings are applied immediately. If you configure events when a pin is defined as an output or is not assigned to the GPIO module at all, this configuration is saved in the internal memory and is applied when the pin becomes a GPIO input.

When an event occurs, your application is notified (see the Notifications section). Call the `DlnGetMessage()` function to obtain the event details. The **DLN_GPIO_CONDITION_MET_EV** structure describes this information.

By default, event generation is disabled for all pins (the **eventType** parameter is set to DLN_GPIO_EVENT_NONE).

Note: The DLN-1 adapters do not support events.

The following GPIO Interface functions can be used to control and monitor events:

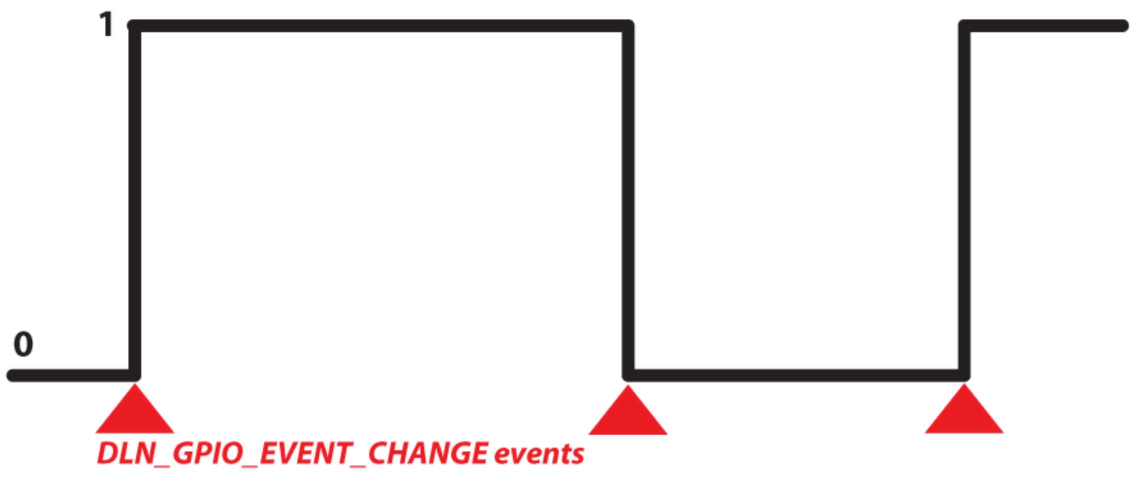
`DlnGpioPinSetEventCfg()`
: Configures event generation for a pin.

`DlnGpioPinGetEventCfg()`
: Retrieves event generation configuration for a pin.

`DlnGpioPinGetSupportedEventTypes()`
: Returns the list of the event types available for a pin.

6.3.1 DLN_GPIO_EVENT_CHANGE Events

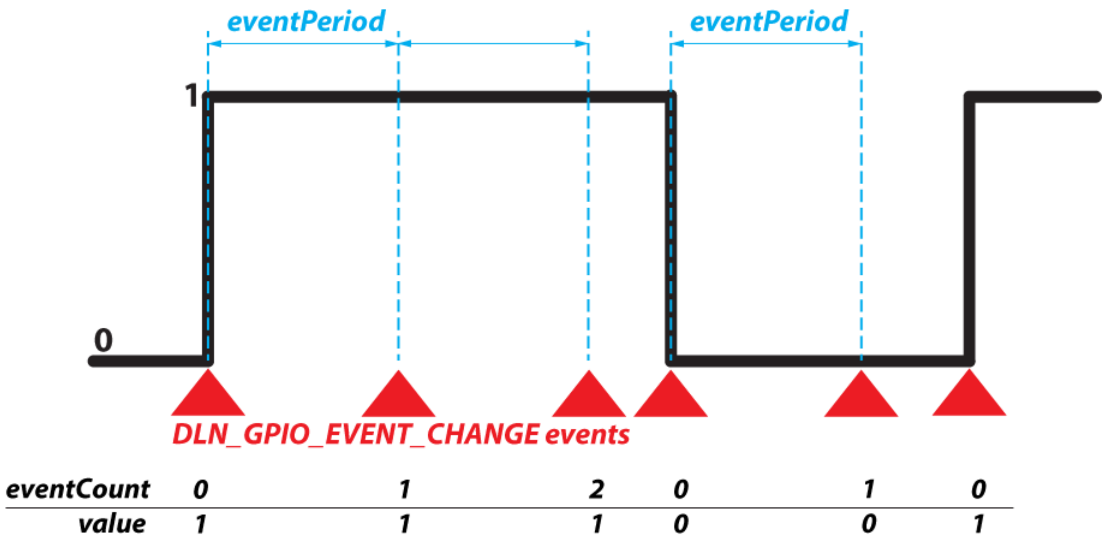
A DLN adapter generates the `DLN_GPIO_EVENT_CHANGE` events each time the level on the input pin changes.



Use the `DlnGpioPinSetEventCfg()` function to configure events. Pass `DLN_GPIO_EVENT_CHANGE` for the **eventType** parameter.

If the **eventPeriod** parameter is zero, the DLN adapter generates single events when the level on the input pin changes.

If the **eventPeriod** is non-zero, the DLN adapter resents the events periodically with the **eventPeriod** interval while the level on the pin remains unchanged (a series of events).

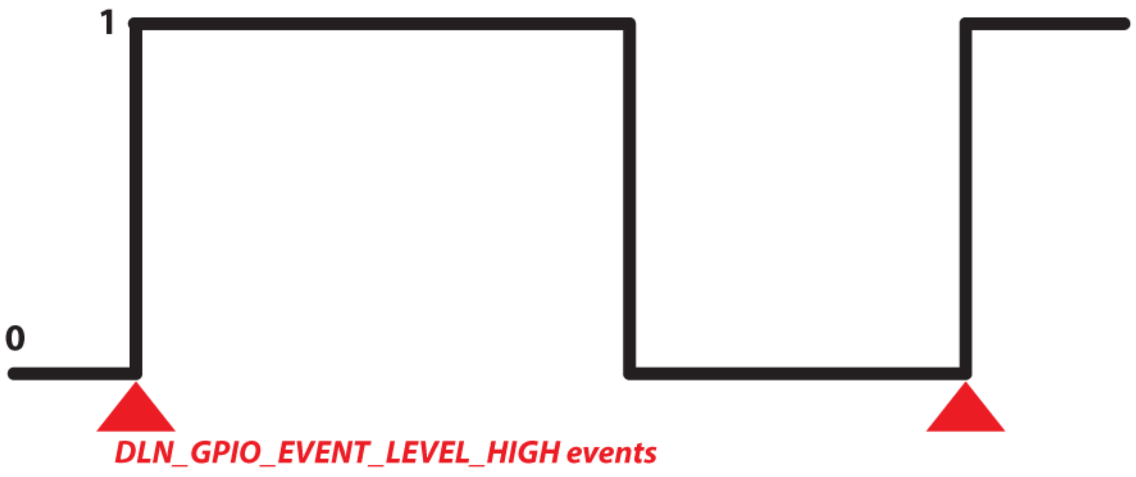


A series of events begins when the level on the GPIO line changes. At this moment, the DLN adapter sends the first event in the series. The **eventCount** field of the first event is set to zero and the **value** field contains the actual value on the GPIO line. Then, while the level on the GPIO line remains unchanged, the DLN adapter repeatedly sends events with the same value in the **value** field. The interval between recurring events is equal to **eventPeriod** milliseconds. The **eventCount** field increments for each event in the series. You can use this field to calculate the time passed from the last level change (**time = eventCount * eventPeriod**).

When the level on the GPIO line changes, a new series of events begins. The **eventCount** field is reset to zero and the **value** field is set to the actual value present on the line after the change.

6.3.2 DLN_GPIO_EVENT_LEVEL_HIGH Events

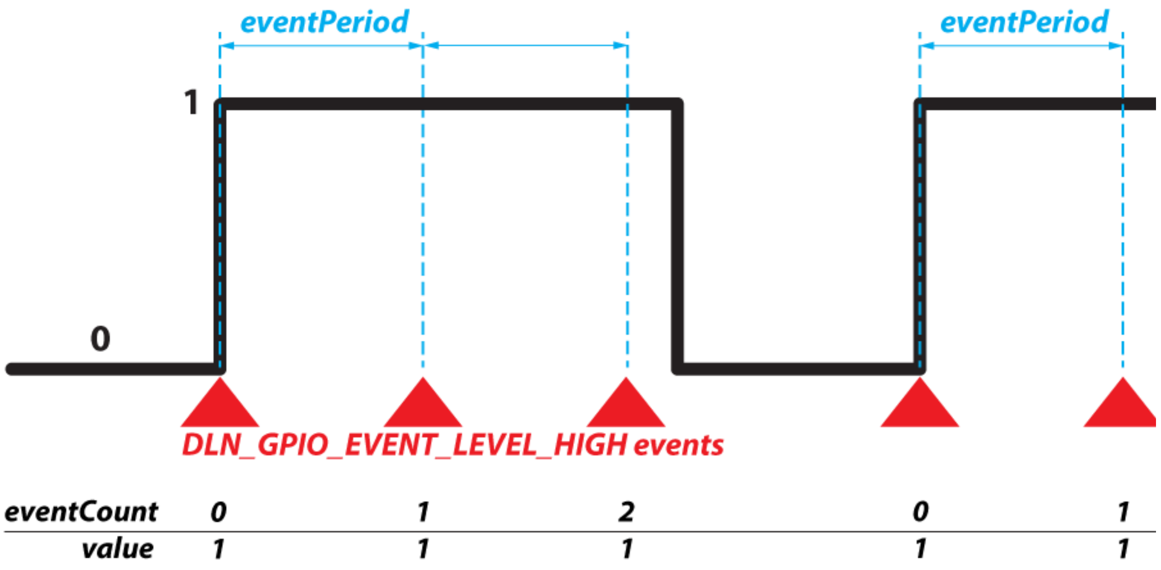
A DLN adapter generates the `DLN_GPIO_EVENT_LEVEL_HIGH` events each time the level on the input pin rises.



Use the `DlnGpioPinSetEventCfg()` function to configure events. Pass `DLN_GPIO_EVENT_LEVEL_HIGH` for the **eventType** parameter.

If the **eventPeriod** is zero, the DLN adapter generates single events when the level on the input pin rises (after transition from low to high level).

If the **eventPeriod** is non-zero , the DLN adapter resents events periodically with the **eventPeriod** interval while the level is high:

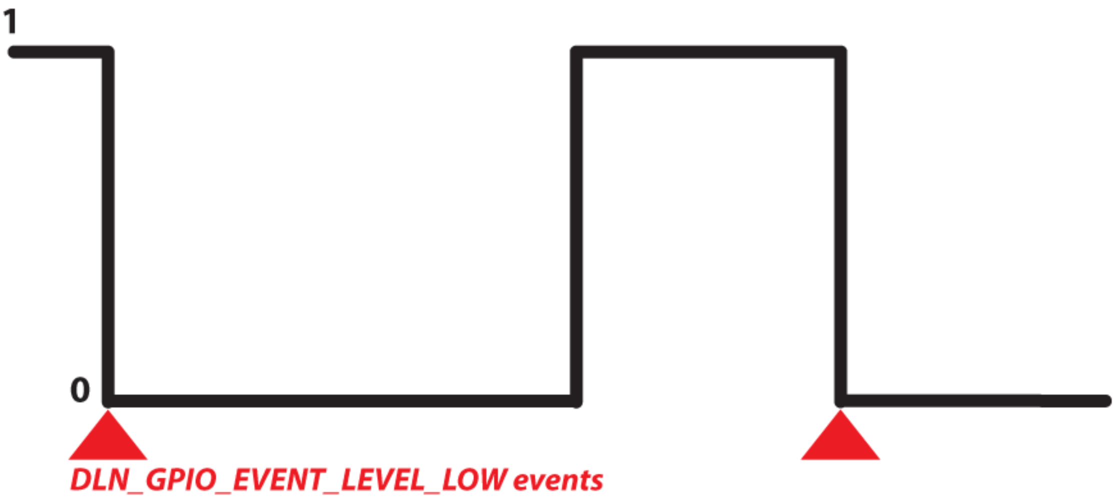


A series of events begins when the level on the GPIO line rises. At this moment, the DLN adapter sends the first event in the series. The **eventCount** field of the first event is set to zero and the **value** field is set to 1 to reflect the actual value on the GPIO line. Then, while the level on the GPIO line remains high, the DLN adapter repeatedly sends events. The same value in the **value** field stays the same. The **eventCount** field increments for every new event. The interval between recurring events is equal to **eventPeriod** milliseconds. You can calculate the time passed from the last level rise (**time = eventCount * eventPeriod**).

When the level on the GPIO line drops, the series of events ends. When the level on the line rises again, a new series of events begins. The **eventCount** field is reset to zero.

6.3.3 DLN_GPIO_EVENT_LEVEL_LOW Events

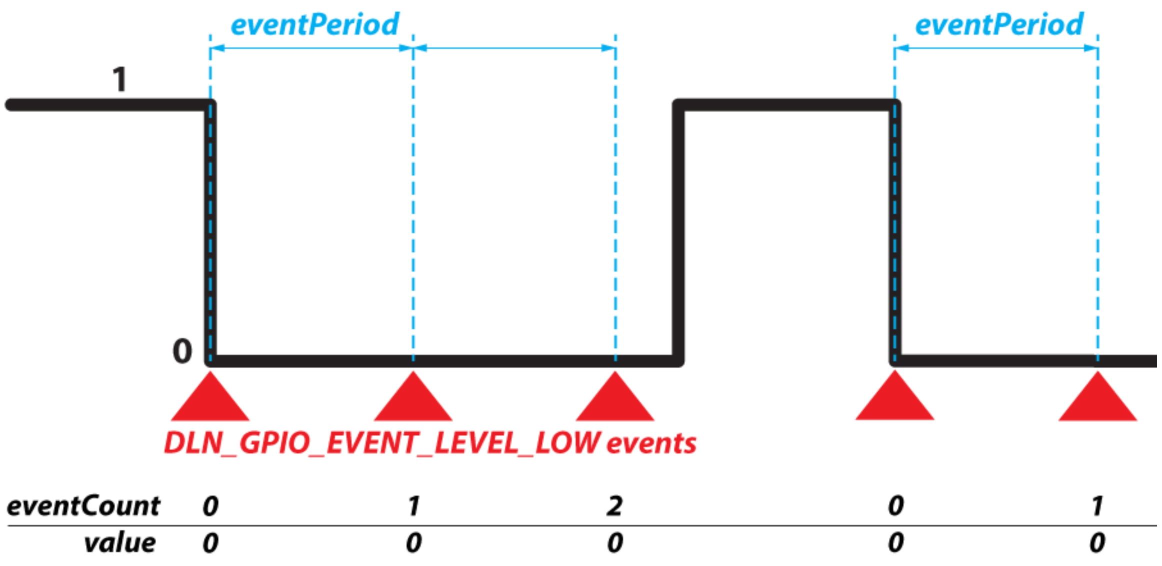
A DLN adapter generates the `DLN_GPIO_EVENT_LEVEL_LOW` events each time the level on the input pin drops.



Use the `DlnGpioPinSetEventCfg()` function to configure events. Pass `DLN_GPIO_EVENT_LEVEL_LOW` for the **eventType** parameter.

If the **eventPeriod** is zero, the DLN adapter generates single events when the level on the input pin drops (after transition from high to low level).

If the **eventPeriod** is non-zero, the DLN adapter resents events periodically with the **eventPeriod** interval while the level is low.

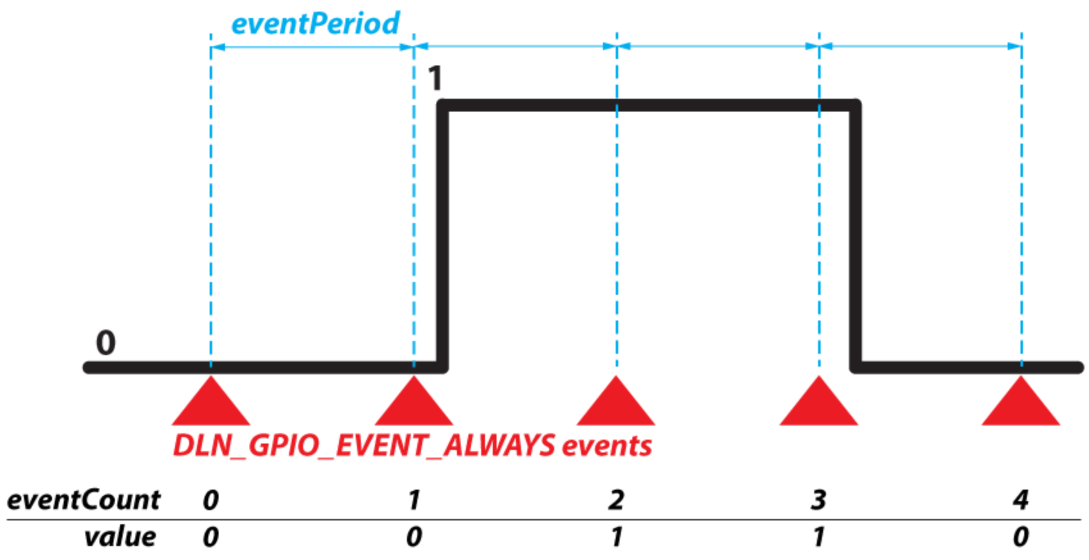


A series of events begins when the level on the GPIO line drops. At this moment, the DLN adapter sends the first event in the series. The **eventCount** field of the first event is set to zero and the **value** field contains 0 to reflect the actual value on the GPIO line. Then, while the level on the GPIO line remains low, the DLN adapter repeatedly sends events. The value in the **value** field stays the same. The **eventCount** field increments for every new event. The interval between recurring events is equal to **eventPeriod** milliseconds. You can calculate the time passed from the last level lowering (**time = eventCount * eventPeriod**).

When the level on the GPIO line rises, the series of events ends. When the level on the line drops again, a new series of events begins. The **eventCount** field is reset to zero.

6.3.4 DLN_GPIO_EVENT_ALWAYS Events

A DLN adapter generates the `DLN_GPIO_EVENT_ALWAYS` events periodically with the **eventPeriod** interval regardless of the level on the pin and its changes. This event notifies the current signal level on a pin.



Call the `DlnGpioPinSetEventCfg()` function to configure events. Specify `DLN_GPIO_EVENT_ALWAYS` as the **eventType** parameter.

For the `DLN_GPIO_EVENT_ALWAYS` events, the **eventPeriod** parameter is required to be non-zero.

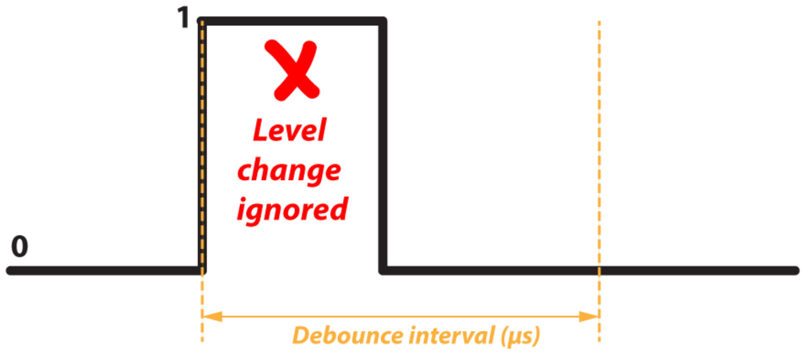
Immediately after you configure the `DLN_GPIO_EVENT_ALWAYS` event, the DLN adapter sends the first event. The **eventCount** field is set to zero and the **value** field contains the actual value on the GPIO line. Then, the DLN adapter repeatedly sends events with the actual value on the line in the **value** field. The interval between recurring events is equal to **eventPeriod** milliseconds. The **eventCount** field increments for each event in the series.

The changes of the level on the pin do not affect the **eventCount** field. You can only reset it if you change the event configuration.

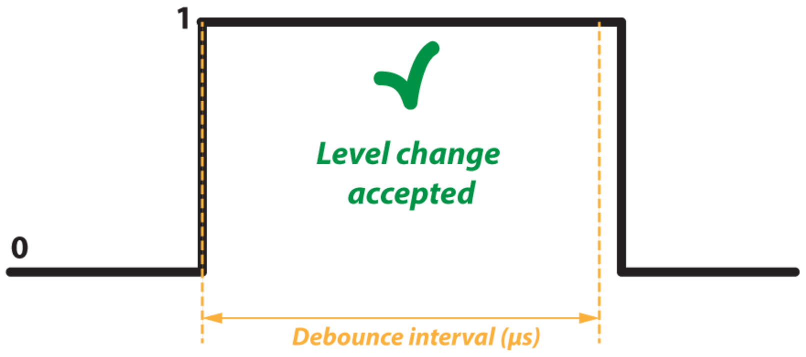
6.4 Debounce Filter

Contact bounce may cause faulty level changes and sending numerous events. To avoid it, the DLN-series adapters support a debounce filter. The debounce filter accepts a new signal level only if it is stable for a predefined period of time (Debounce interval).

A pulse shorter than the specified debounce interval is automatically rejected:



A pulse longer than the debounce interval is accepted:



If events are configured for the corresponding input pin, they will only be generated after the debounce filter accepts the level change.

The Debounce interval is the same for all pins.You can change it by using the `DlnGpioSetDebounce()` function. The Debounce interval is specified in microseconds (µs). If a DLN-series adapter does not support the specified value, it rounds it up to the nearest supported value.

You can switch the debounce filter on/off for each pin individually. To switch it on, use the `DlnGpioPinDebounceEnable()` function. To switch it off, use the `DlnGpioPinDebounceDisable()` function. If a pin is not assigned to the GPIO module, your settings are saved in the internal memory and will be applied when you assign the pin to the GPIO module.

By default, the debounce filters are disabled on all pins.

Note: The debounce Filter feature is not available for the DLN-1, DLN-2 adapters.

Use the following GPIO Interface functions to control and monitor the debounce filter:

`DlnGpioPinDebounceEnable()`
Enables the Debounce Filter on a pin.

`DlnGpioPinDebounceDisable()`

Disables the Debounce Filter on a pin

`DlnGpioPinDebounceIsEnabled()`
: Determines whether the Debounce Filter is enabled on a pin.

`DlnGpioSetDebounce()`
: Configures the debounce interval (the minimum duration of pulses to be accepted).

`DlnGpioGetDebounce()`
: Retrieves the debounce interval value.

6.5 Open Drain Mode

Your application may require connecting several outputs together on a single I/O line. In this case, you should use the Open Drain feature to avoid the situation when different outputs set different signal levels on the line. To avoid hardware damage all interconnected outputs should be in open drain mode.

In open drain mode, pin cannot output high level (logical 1) on the line. It can either output low level (logical 0) or be in high-impedance state (logically disconnected). In high-impedance, pin does not affect the signal level on the line. A pull-up resistor pulls the line to high voltage level when all outputs are in high-impedance.

The Open Drain Mode can be enabled/disabled on each output pin individually. To activate this feature, use the `DlnGpioPinOpendrainEnable()` function. To disable it, use the `DlnGpioPinOpendrainDisable()` function. If a pin is not assigned to the GPIO module, your settings are saved in the internal memory and will be applied when you assign the pin to the GPIO module.

By default, the Open Drain Mode is disabled on all pins.

Note: The DLN-1, DLN-2 adapters do not support the Open Drain mode.

Use the following GPIO Interface functions to control and monitor the Open Drain Mode:

`DlnGpioPinOpendrainEnable()`
: Enables the Open Drain mode on a pin.

`DlnGpioPinOpendrainDisable()`
: Disables the Open Drain mode on a pin.

`DlnGpioPinOpendrainIsEnabled()`
: Determines whether the Open Drain mode in enabled on a pin.

6.6 Pull-up/Pull-down Resistors

Most I/O lines are equipped with an embedded pull-up resistor. A pull-up resistor has one end connected to the positive voltage and the other end connected to an input. The pull-up resistor ensures that the level on the input is high (logic 1) when no external device is connected. If an external device is connected and outputs 0, the input detects logic 0.

A pull-down resistor works the similar way, but it is connected to ground and the logic 0 is present on the line when no external device is connected.

Pull-up and pull-down resistors can be enabled/disabled on each input pin individually.

Call the `DlnGpioPinPullupEnable()` function to enable the pull-up resistor. It can be disabled by using the `DlnGpioPinPullupDisable()` function.

To enable a pull-down resistor, use the `DlnGpioPinPulldownEnable()` function. To disable it, use the `DlnGpioPinPulldownDisable()` function.

You cannot simultaneously enable pull-up and pull-down resistors for the same pin.

If a pin is not assigned to the GPIO module, your settings are saved in the internal memory and will be applied when you assign the pin to the GPIO module.

By default, pull-up resistors are enabled for all pins. The pull-down resisters are disabled by default.

Note: DLN-4 adapters do not support pull-down resistors.

Use the following GPIO Interface functions to control and monitor pull-up resistors:

`DlnGpioPinPullupEnable()`
: Enables the pull-up resistor on a pin.

`DlnGpioPinPullupDisable()`
: Disables the pull-up resistor on a pin.

`DlnGpioPinPullupIsEnabled()`
: Determines whether the pull-up resistor is enabled on a pin.

Use the following GPIO Interface functions to control and monitor pull-down resistors:

`DlnGpioPinPulldownEnable()`
: Enables the pull-down resistor on a pin.

`DlnGpioPinPulldownDisable()`
: Disables the pull-down resistor on a pin.

`DlnGpioPinPulldownIsEnabled()`
: Determines whether the pull-down resistor is enabled on a pin.

6.7 Default Configuration

When you assign a pin to the GPIO module, it has the following default configuration:

- The pin is configured as an input.
- Events are disabled for the pin.
- Debounce filtering is turned off.
- An Open Drain mode is not active.
- An embedded pull-up resistor is active.

- An embedded pull-down resistor is not active.

You can change the configuration before or after you assign a pin to the GPIO module. If a pin is not currently assigned to the GPIO module, your settings are saved in the internal memory and will be applied when you assign the pin to the GPIO module.

6.8 Simple GPIO Module Example

The following example shows how to enable/disable and set GPIO pin parameters, such as direction, output value and read GPIO pin value, when its direction set to input. You can find the complete example in the “*..\Program Files\Diolan\DLN\examples\c_cpp\examples\simple*” folder after DLN setup package installation.

```
1   #include "..\..\..\common\dln_generic.h"
2   #include "..\..\..\common\dln_gpio.h"
3   #pragma comment(lib, "..\\..\\..\\bin\\dln.lib")
4
5
6   int _tmain(int argc, _TCHAR* argv[])
7   {
8   // Open device
9   HDLN device;
10  DlnOpenUsbDevice(&device);
11
12  // Configure pin 0 to input with pullup
13  DlnGpioPinPullupEnable(device, 0);
14  DlnGpioPinSetDirection(device, 0, 0);
15  DlnGpioPinEnable(device, 0);
16
17  // Configure pin 1 to output
18  DlnGpioPinSetDirection(device, 1, 1);
19  DlnGpioPinEnable(device, 1);
20
21  // Read pin 0 and set its iverted value on pin 1
22  uint8_t value;
23  DlnGpioPinGetVal(device, 0, &value);
24  DlnGpioPinSetOutVal(device, 1, value ^ 1);
25
26  // Disable buttons
27  DlnGpioPinDisable(device, 0);
28  DlnGpioPinDisable(device, 0);
29  // Close device
30  DlnCloseHandle(device);
31  return 0;
32  }
```

Line 1: `#include "..\..\..\common\dln_generic.h"`

The *dln_generic..h* header file declares functions and data structures for the generic interface

Line 2: `#include "..\..\..\common\dln_gpio.h"`

The dln_gpio.h header file declares functions and data structures for GPIO interface.

Line 3: `#pragma comment(lib, "..\\..\\..\\bin\\dln.lib")`

Use *dln.lib* library while project linking.

Line 10: `DlnOpenUsbDevice(&device);`

The function establishes the connection with the DLN adapter. This application uses the USB connectivity of the adapter. For additional options, refer to the Device Opening & Identification section.

Line 13: `DlnGpioPinPullupEnable(device, 0);`

Enable pullup for GPIO pin 0. You can read more about pullups at Pull-up/Pull-down Resistors section.

Line 14: `DlnGpioPinSetDirection(device, 0, 0);`

Set GPIO pin 0 to input. For more information read Digital Inputs section.

Line 15: `DlnGpioPinEnable(device, 0);`

Enable GPIO pin 0.

Line 18: `DlnGpioPinSetDirection(device, 1, 1);`

Set GPIO pin 1 direction to output. For more information read Digital Outputs section.

Line 19: `DlnGpioPinEnable(device, 1);`

Enable GPIO pin 1.

Line 23: `DlnGpioPinGetVal(device, 0, &value);`

Get value on GPIO pin 0.

Line 24: `DlnGpioPinSetOutVal(device, 1, value ^ 1);`

Set output value on the GPIO pin 1, which is opposite to the value variable.

Line 27: `DlnGpioPinDisable(device, 0);`

Disable GPIO pin 0.

Line 28: `DlnGpioPinDisable(device, 1);`

Disable GPIO pin 1.

Line 30: `DlnCloseHandle(device);`

The application closes the handle to the connected DLN-series adapter.

6.9 GPIO Interface Functions

Use the GPIO Interface functions to control and monitor the GPIO module of a DLN-series adapter.

General pin information:

`DlnGpioGetPinCount()`
Retrieves the number of pins that can be assigned to the GPIO module.

`DlnGpioPinEnable()`

Assigns a pin to the GPIO module.

`DlnGpioPinDisable()`

Unassigns a pin from the GPIO module.

`DlnGpioPinIsEnabled()`

Retrieves whether a pin is connected to the GPIO module.

`DlnGpioPinSetDirection()`

Configures a pin as a digital input or output.

`DlnGpioPinGetDirection()`

Retrieves whether a pin is configured as a digital input or output.

`DlnGpioPinSetOutVal()`

Defines a value on the output pin.

`DlnGpioPinGetOutVal()`

Retrieves the output value configured to the pin.

`DlnGpioPinGetVal()`

Retrieves the actual value on the I/O line.

Event functions:

`DlnGpioPinSetEventCfg()`

Configures event generation for a pin.

`DlnGpioPinGetEventCfg()`

Retrieves event generation configuration for a pin.

`DlnGpioPinGetSupportedEventTypes()`

Returns the list of the event types available for a pin.

Debounce filter functions:

`DlnGpioPinDebounceEnable()`

Enables the Debounce Filter on a pin.

`DlnGpioPinDebounceDisable()`

Disables the Debounce Filter on a pin

`DlnGpioPinDebounceIsEnabled()`

Determines whether the Debounce Filter is enabled on a pin.

`DlnGpioSetDebounce()`

Configures the debounce interval (the minimum duration of pulses to be registered).

`DlnGpioGetDebounce()`

Retrieves the debounce interval value.

Open Drain mode functions:

`DlnGpioPinOpendrainEnable()`

Enables the Open Drain mode on a pin.

`DlnGpioPinOpendrainDisable()`

Disables the Open Drain mode on a pin.

`DlnGpioPinOpendrainIsEnabled()`

Determines whether the Open Drain mode in enabled on a pin.

Pull-up/Pull-down resistors functions:

`DlnGpioPinPullupEnable()`

Enables the pull-up resistor on a pin.

`DlnGpioPinPulldownEnable()`

Enables the pull-down resistor on a pin.

`DlnGpioPinPullupDisable()`

Disables the pull-up resistor on a pin.

`DlnGpioPinPulldownDisable()`

Disables the pull-down resistor on a pin.

`DlnGpioPinPullupIsEnabled()`

Determines whether the pull-up resistor is enabled on a pin.

`DlnGpioPinPulldownIsEnabled()`

Determines whether the pull-down resistor is enabled on a pin.

The *dln_gpio.h* file declares the GPIO Interface functions.

DlnGpioGetPinCount() Function

The **`DlnGpioGetPinCount()`** function retrieves the total number of GPIO pins available in the DLN-series adapter.

Syntax

```
DLN_RESULT DlnGpioGetPinCount(
    HDLN handle,
    uint16_t* count
 );
```

Parameters

handle

A handle to a DLN-series adapter.

count

A pointer to an unsigned 16-bit integer that receives the number of available GPIO pins.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the GPIO pin count.

Remarks

The **`DlnGpioGetPinCount()`** function retrieves the number of pins that can be configured as GPIO. Even the pins that are assigned for other modules (but can be configured as GPIO) will be counted. To check whether a pin is connected to the GPIO module, use the **`DlnGpioPinIsEnabled()`** function.

The GPIO functions that accept a pin number as a parameter will return the **`DLN_RES_INVALID_PIN_NUMBER`** error code if the specified pin number exceeds the value returned by the **`DlnGpioGetPinCount()`** function.

The **`DlnGpioGetPinCount()`** function is declared in the *dln_gpio.h* file.

DlnGpioPinEnable() Function

The **`DlnGpioPinEnable()`** function connects a pin to the GPIO module.

Syntax

```
DLN_RESULT DlnGpioPinEnable(
    HDLN handle,
    uint8_t pin
);
```

Parameters

handle

A handle to a DLN-series adapter.

pin

A number of the pin that you want to configure.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully connected the pin to the GPIO module.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the **`DlnGpioGetPinCount()`** function to find the maximum possible pin number.

DLN_RES_PIN_IN_USE (0xA5)

Another module uses the pin and the GPIO module cannot use it. Use the **`DlnGetPinCfg()`** function to get the name of module using the pin.

Remarks

If you have not changed the configuration settings for the pin which you want to configure as a GPIO, it will have the default configuration. See Digital Input Events for details.

The **`DlnGpioPinEnable()`** function is declared in the *dln_gpio.h* file.

DlnGpioPinDisable() Function

The **`DlnGpioPinDisable()`** function disconnects a pin from the GPIO module. Then, another module can use the pin.

Syntax

```
DLN_RESULT DlnGpioPinDisable(
   HDLN handle,
   uint8_t pin
);
```

Parameters

handle

A handle to a DLN-series adapter.

pin

A number of the pin that you want to configure.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully disconnected the pin from the GPIO module.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the **`DlnGpioGetPinCount()`** function to find the maximum possible pin number.

DLN_RES_PIN_NOT_CONNECTED_TO_MODULE (0xAA)

The GPIO module does not use the pin, so the pin cannot be disconnected from the module.

Remarks

When you use the **`DlnGpioPinDisable()`** function, the pin configuration settings are not lost. They are saved in the internal memory. Later, if you reassign this pin to the GPIO module by using the **`DlnGpioPinEnable()`** function, the pin configuration is restored.

The **`DlnGpioPinDisable()`** function is declared in the *dln_gpio.h* file.

DlnGpioPinIsEnabled() Function

The **`DlnGpioPinIsEnabled()`** function informs whether the GPIO module currently uses the pin.

Syntax

```
DLN_RESULT DlnGpioPinIsEnabled(
   HDLN handle,
   uint8_t pin,
   uint8_t* enabled
);
```

Parameters

handle

A handle to a DLN-series adapter.

pin

A number of the pin that you want to check.

enabled

A pointer to an unsigned 8-bit integer. After the function execution, this integer is set to one of the following values:

Value	Description
1	The pin is configured as GPIO.
0	The pin is NOT configured as GPIO.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the pin information.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the `DlnGpioGetPinCount()` function to find the maximum possible pin number.

Remarks

To connect the pin with the GPIO module, use the `DlnGpioPinEnable()` function. To disconnect the pin assigned to the GPIO module, call the `DlnGpioPinDisable()` function.

To check which module uses the pin, use the `DlnGetPinCfg()` function.

The **`DlnGpioPinIsEnabled()`** function is declared in the *dln_gpio.h* file.

DlnGpioPinSetDirection() Function

The **`DlnGpioPinSetDirection()`** function configures a pin as an input or as an output.

Syntax

```
DLN_RESULT DlnGpioPinSetDirection(
    HDLN handle,
    uint8_t pin,
    uint8_t direction
 );
```

Parameters

handle

A handle to the DLN-series adapter.

pin

A number of the GPIO pin that you want to configure.

direction

Direction of a pin. Set to 0 for input or 1 for output.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The pin direction has been successfully retrieved.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the `DlnGpioGetPinCount()` function to find the maximum possible pin number.

Remarks

By default, The GPIO pin is defined as an input. You can check the current pin direction by using the `DlnGpioPinGetDirection()` function.

If the pin is not currently assigned to the GPIO module, the pin direction setting is not lost. It is saved in internal memory. Later, when you assign this pin to the GPIO module by using the `DlnGpioPinEnable()` function, the pin direction will be applied.

The `DlnGpioPinSetDirection()` function is declared in *dln_gpio.h* file.

DlnGpioPinGetDirection() Function

The `DlnGpioPinGetDirection()` function retrieves current direction of a GPIO pin.

Syntax

```
DLN_RESULT DlnGpioPinGetDirection(
   HDLN handle,
   uint8_t pin,
   uint8_t* direction
);
```

Parameters

handle

A handle to the DLN-series adapter.

pin

A number of the GPIO pin that you want to check.

direction

A pointer to an unsigned 8-bit integer. After the function execution, the integer will be filled with the pin current direction. Possible values:

Value	Description
1	The pin is an output.
0	The pin is an input.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the pin direction.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the `DlnGpioGetPinCount()` function to find the maximum possible pin number.

Remarks

If the pin is not currently assigned to the GPIO module, the pin direction setting is got from internal memory where it is saved when you define it. To change the current pin direction, use the `DlnGpioPinSetDirection()` function.

The **`DlnGpioPinGetDirection()`** function is declared in the *dln_gpio.h* file.

DlnGpioPinGetVal() Function

The **`DlnGpioPinGetVal()`** function retrieves the current value on the specified GPIO pin.

Syntax

```
DLN_RESULT DlnGpioPinGetVal(
   HDLN handle,
   uint8_t pin,
   uint8_t* value
);
```

Parameters

handle

A handle to the DLN-series adapter.

pin

A number of the pin which value you want to know.

value

A pointer to an unsigned 8-bit integer. After the function execution, this integer will be set to the pin value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function retrieved the current pin value successfully.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the **`DlnGpioGetPinCount()`** function to find the maximum possible pin number.

Remarks

The **`DlnGpioPinGetVal()`** function retrieves the value on the pin regardless of its direction. To check the GPIO pin direction, use the **`DlnGpioPinGetDirection()`** function. You can change the pin output value by using the **`DlnGpioPinSetOutVal()`** function.

The **`DlnGpioPinGetVal()`** function is declared in the *dln_gpio.h* file.

DlnGpioPinSetOutVal() Function

The **`DlnGpioPinSetOutVal()`** function sets the output value for the specified GPIO pin.

Syntax

```
DLN_RESULT DlnGpioPinSetOutVal(
   HDLN handle,
   uint8_t pin,
   uint8_t value
);
```

Parameters

handle

A handle to the DLN-series adapter.

pin

A number of the pin that you want to configure.

value

A pin output value to be set.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The pin output value was successfully applied.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the `DlnGpioGetPinCount()` function to find the maximum possible pin number.

Remarks

If the specified pin is an output, the value is applied immediately. If the pins is an input or is not assigned to the GPIO module, the value is stored in the internal memory. The stored value will be applied when the pin becomes output.

The **`DlnGpioPinSetOutVal()`** function is declared in the *dln_gpio.h* file.

DlnGpioPinGetOutVal() Function

The **`DlnGpioPinGetOutVal()`** function retrieves the pin output value.

Syntax

```
DLN_RESULT DlnGpioPinGetOutVal(
    HDLN handle,
    uint8_t pin,
    uint8_t* value
 );
```

Parameters

handle

A handle to the DLN-series adapter.

pin

A number of the pin which output value you want to know.

value

A pointer to an unsigned 8-bit integer. After the function execution, this integer is filled with the pin output value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function retrieved the pin output value successfully.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the **DlnGpioGetPinCount()** function to find the maximum possible pin number.

Remarks

The default pin output value is 0. By using the **DlnGpioPinSetOutVal()** function you can change the output value. If the specified pin is not assigned to the GPIO module or is not an output, the **DlnGpioPinGetOutVal()** function takes the output value from the internal memory, where it is saved.

The **DlnGpioPinGetOutVal()** function is declared in the *dln_gpio.h* file.

DlnGpioPinSetEventCfg() Function

The **DlnGpioPinSetEventCfg()** function configures when and which events should be generated for the specified pin. For more information, read Digital Input Events.

Syntax

```
DLN_RESULT DlnGpioPinSetEventCfg(
   HDLN handle,
   uint8_t pin,
   uint8_t eventType,
   uint16_t eventPeriod
);
```

Parameters

handle

A handle to the DLN-series adapter.

pin

A number of the pin for which you want to configure the event generation.

eventType

Defines when the DLN adapter should generate pin events. The following values are available:

Value	Description
0 or DLN_GPIO_EVENT_NONE	The DLN adapter does not generate any events.
1 or DLN_GPIO_EVENT_CHANGE	The DLN adapter generates events when the level on the digital input line changes. For details, read DLN_GPIO_EVENT_CHANGE Events

Value	Description
2 or DLN_GPIO_EVENT_LEVEL_HIGH	The DLN adapter generates events when the high level is present on the digital input line or after transition from low to high level. For details, read DLN_GPIO_EVENT_LEVEL_HIGH Events
3 or DLN_GPIO_EVENT_LEVEL_LOW	The DLN adapter generates events when the low level is present on the digital input line or after transition from high to low level. For details, read DLN_GPIO_EVENT_LEVEL_LOW Events
4 or DLN_GPIO_EVENT_ALWAYS	The DLN adapter generates events continuously, regardless of the level. For details, read DLN_GPIO_EVENT_ALWAYS Events

eventPeriod

Defines the interval in milliseconds (ms) at which the events occur.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function configured event settings successfully.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the `DlnGpioGetPinCount()` function to find the maximum possible pin number.

DLN_RES_INVALID_EVENT_TYPE (0xA9)

The specified event type is not valid. Use the `DlnGpioPinGetSupportedEventTypes()` function to check the event types supported for the pin.

DLN_RES_INVALID_EVENT_PERIOD (0xAC)

The specified event period is not valid.

Remarks

Note: The DLN-1 adapters do not support GPIO events.

The `DlnGpioPinSetEventCfg()` function is declared in the *dln_gpio.h* file.

DlnGpioPinGetEventCfg() Function

The **`DlnGpioPinGetEventCfg()`** function retrieves the current event configuration for the specified pin. For more information, read Digital Input Events.

Syntax

```
DLN_RESULT DlnGpioPinGetEventCfg(
   HDLN handle,
   uint8_t pin,
   uint8_t* eventType,
   uint16_t* eventPeriod
);
```

Parameters

handle

A handle to the DLN-series adapter.

pin

A number of the pin that you want to check.

eventType

A pointer to an unsigned 8-bit integer. After the function execution, this integer is set to the current event configuration of the pin.

eventPeriod

A pointer to an unsigned 16-bit integer. After the function execution, this integer is set to the interval at which the specified events occur.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function retrieved the event configuration successfully.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the **`DlnGpioGetPinCount()`** function to find the maximum possible pin number.

Remarks

Note: GPIO events are by the DLN-1 adapters.

The **`DlnGpioPinGetEventCfg()`** function is declared in the *dln_gpio.h* file.

DlnGpioPinGetSupportedEventTypes() Function

The **`DlnGpioPinGetSupportedEventTypes()`** function returns all event types supported for the specified pin.

Syntax

```
DLN_RESULT DlnGpioPinGetSupportedEventTypes(
    HDLN handle,
    uint16_t pin,
    DLN_GPIO_PIN_EVENT_TYPES* supportedEventTypes
);
```

Parameters

handle

A handle to the DLN-series adapter.

pin

A number of the pin for which you want to get the information.

supportedEventTypes

The pointer to the **DLN_GPIO_PIN_EVENT_TYPES** structure that is filled by the supported event types. After the function execution, the structure fields include the following information:

Field	Description
count	The number of available event types.
eventTypes[8]	An 8-element array. Each element corresponds to one of the supported event types.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function was successful.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the `DlnGpioGetPinCount()` function to find the maximum possible pin number.

Remarks

Note: The DLN-1 adapters do not support GPIO events.

The **`DlnGpioPinGetSupportedEventTypes()`** function is declared in the *dln_gpio.h* file.

DlnGpioPinDebounceEnable() Function

The **`DlnGpioPinDebounceEnable()`** function enables Debounce Filter for the specified pin.

Syntax

```
DLN_RESULT DlnGpioPinDebounceEnable(
   HDLN handle,
   uint8_t pin
);
```

Parameters

handle

A handle to the DLN-series adapter.

pin

A number of the pin that you want to configure.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function activated the debounce filter successfully.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the `DlnGpioGetPinCount()` function to find the maximum possible pin number.

DLN_RES_COMMAND_NOT_SUPPORTED (0x91)

The adapter does not support debounce filtering.

Remarks

You can use the **`DlnGpioPinDebounceEnable()`** function regardless of whether or not the specified pin is connected to the GPIO module. For a disconnected pin, the debounce filter configuration is saved in the internal memory. This value will be applied when the pin is connected to the GPIO module.

Note: The debounce filter feature is not supported by DLN-1 and DLN-2 adapters.

The **`DlnGpioPinDebounceEnable()`** function is declared in the *dln_gpio.h* file.

DlnGpioPinDebounceDisable() Function

The **`DlnGpioPinDebounceDisable()`** disables Debounce Filter for the specified pin.

Syntax

```
DLN_RESULT DlnGpioPinDebounceDisable(
    HDLN handle,
    uint8_t pin
 );
```

Parameters

handle

A handle to the DLN-series adapter.

pin

A number of the pin that you want to configure.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function deactivated debounce filter successfully.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the `DlnGpioGetPinCount()` function to find the maximum possible pin number.

DLN_RES_COMMAND_NOT_SUPPORTED (0x91)

The adapter does not support debounce filtering.

Remarks

Note: The DLN-1 and DLN-2 adapters do not support the debounce filter feature.

The **`DlnGpioPinDebounceDisable()`** function is declared in the *dln_gpio.h* file.

DlnGpioPinDebounceIsEnabled() Function

The **`DlnGpioPinDebounceIsEnabled()`** function informs whether the Debounce Filter is currently enabled for the specified pin.

Syntax

```
DLN_RESULT DlnGpioPinOpendrainIsEnabled(
   HDLN handle,
   uint8_t pin,
   uint8_t* enabled
);
```

Parameters

handle

A handle to the DLN-series adapter.

pin

A number of the pin that you want to check.

enabled

A pointer to an unsigned 8-bit integer. After the function execution, this parameter is set to one of the following values:

Value	Description
1	Debounce filtering is active.
0	Debounce filtering is not active.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function retrieves the debounce filter settings successfully.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the `DlnGpioGetPinCount()` function to find the maximum possible pin number.

DLN_RES_COMMAND_NOT_SUPPORTED (0x91)

The adapter does not support debounce filtering.

Remarks

The `DlnGpioPinDebounceIsEnabled()` function returns whether or not the debounce filtering feature is enabled for the specified pin. To enable this feature, use the `DlnGpioPinDebounceEnable()` function. To disable this feature, use the `DlnGpioPinDebounceDisable()` function. To specify the debounce interval, use the `DlnGpioSetDebounce()` function (See Debounce Filter for details).

Note: DLN-1, DLN-2 adapters do not support the debounce filter feature.

The `DlnGpioPinDebounceIsEnabled()` function is declared in the *dln_gpio.h* file.

DlnGpioSetDebounce() Function

The `DlnGpioSetDebounce()` function specifies the debounce interval (the minimum duration of pulses to be registered). See Debounce Filter for details.

Syntax

```
DLN_RESULT DlnGpioSetDebounce(
   HDLN handle,
   uint32_t* duration
);
```

Parameters

handle

A handle to the DLN-series adapter.

duration

A pointer to an unsigned 32-bit integer. After the function execution, this integer is set to the debounce duration, rounded up to the nearest value supported by the adapter. See Debounce Filter for details.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function specified the debounce interval successfully.

DLN_RES_COMMAND_NOT_SUPPORTED (0x91)

The adapter does not support debounce filtering.

Remarks

The specified debounce interval is actual for all pins where the debounce filter is enabled. This value should be specified in microseconds (µs): from 1µs up to 4,294,967,295µs (~1h 10m). If a DLN-series adapter does not support the specified value, it rounds it up to the nearest supported value.

The default value of the debounce interval depends on the DLN adapter. Once you call the **`DlnGpioSetDebounce()`** function and specify the debounce interval value, you change the default value. If the debounce filter is disabled for a pin or a pin is not assigned to the GPIO module, this value will be applied when the pin is assigned to the GPIO module with the debounce filter enabled.

Note: DLN-1 and DLN-2 adapters do not support the debounce filter feature.

The **`DlnGpioSetDebounce()`** function is declared in the *dln_gpio.h* file.

DlnGpioGetDebounce() Function

The **`DlnGpioGetDebounce()`** function retrieves the current value of the debounce interval (the minimum duration of the pulse to be registered). See Debounce Filter for details.

Syntax

```
DLN_RESULT DlnGpioGetDebounce(
   HDLN handle,
   uint32_t* duration
);
```

Parameters

handle

A handle to the DLN-series adapter.

duration

A pointer to an unsigned 32-bit integer. After the function executed, this integer is set to the debounce interval value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function retrieved the debounce interval successfully.

DLN_RES_COMMAND_NOT_SUPPORTED (0x91)

The adapter does not support debounce filtering.

Remarks

The debounce interval is actual for all GPIO pins with the enabled debounce filter. To change the debounce interval value, use the `DlnGpioSetDebounce()` function.

Note: DLN-1 and DLN-2 adapters do not support the debounce filter feature.

The **`DlnGpioGetDebounce()`** function is declared in the *dln_gpio.h* file.

DlnGpioPinOpendrainEnable() Function

The **`DlnGpioPinOpendrainEnable()`** function enables Open Drain Mode for the specified pin.

Syntax

```
DLN_RESULT DlnGpioPinOpendrainEnable(
   HDLN handle,
   uint8_t pin
);
```

Parameters

handle

A handle to the DLN-series adapter.

pin

A number of the pin that you want to configure.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function enabled the open drain mode successfully.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the **`DlnGpioGetPinCount()`** function to find the maximum possible pin number.

Remarks

By default, the Open Drain Mode is disabled for all GPIO pins. The **`DlnGpioPinOpendrainEnable()`** function enables the Open Drain Mode for the specified pin. If the pin is not assigned to the GPIO module, this configuration will be saved in the internal memory. When you assign the pin to the GPIO module, the saved configuration will be applied.

Note: DLN-1 and DLN-2 adapters do not support the Open Drain mode.

The **`DlnGpioPinOpendrainEnable()`** function is defined in the *dln_gpio.h* file.

DlnGpioPinOpendrainDisable() Function

The **`DlnGpioPinOpendrainDisable()`** disables Open Drain Mode for the specified pin.

Syntax

```
DLN_RESULT DlnGpioPinOpendrainDisable(
   HDLN handle,
   uint8_t pin
);
```

Parameters

handle

A handle to the DLN-series adapter.

pin

A number of the pin that you want to configure.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function disabled the open drain mode successfully.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the **`DlnGpioGetPinCount()`** function to find the maximum possible pin number.

Remarks

The **`DlnGpioPinOpendrainDisable()`** function disables the Open Drain Mode for the specified pin. If the pin is not assigned to the GPIO module, this configuration will be saved in the internal memory. When you assign the pin to the GPIO module, the saved configuration will be applied.

Note: Most pins of DLN-1 and DLN-2 adapters do not support the Open Drain mode.

The **`DlnGpioPinOpendrainDisable()`** function is defined in the *dln_gpio.h* file.

DlnGpioPinOpendrainIsEnabled() Function

The **`DlnGpioPinOpendrainIsEnabled()`** function informs whether the pin output is currently configured as push-pull or Open Drain.

Syntax

```
DLN_RESULT DlnGpioPinOpendrainIsEnabled(
   HDLN handle,
   uint8_t pin,
   uint8_t* enabled
);
```

Parameters

handle

A handle to the DLN-series adapter.

pin

A number of the pin that you want to check.

enabled

A pointer to an unsigned 8-bit integer. After the function execution, this integer is set to the current pin configuration. The following values are available:

Value	Description
1	The output is Open Drain.
0	The output is push-pull.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function retrieved the open drain mode configuration successfully.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the **`DlnGpioGetPinCount()`** function to find the maximum possible pin number.

Remarks

Note: Most pins of DLN-1 and DLN-2 adapters do not support the Open Drain mode.

The **`DlnGpioPinOpendrainIsEnabled()`** function is declared in the *dln_gpio.h* file.

DlnGpioPinPullupEnable() Function

The **`DlnGpioPinPullupEnable()`** function activates an embedded pull-up resistor for the specified pin. For more information, read Pull-up/Pull-down Resistors.

Syntax

```
DLN_RESULT DlnGpioPinPullupEnable(
   HDLN handle,
   uint8_t pin
);
```

Parameters

handle

A handle to the DLN-series adapter.

pin

A number of the pin for which you want to enable a pull-up resistor.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function activated the pull-up resistor successfully.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the **`DlnGpioGetPinCount()`** function to find the maximum possible pin number.

Remarks

By default, the pull-up resistors are active for GPIO pins. You can change this configuration by using the **`DlnGpioPinPullupDisable()`** function. To check the current configuration setting for a pin, use the **`DlnGpioPinPullupIsEnabled()`** function.

The **`DlnGpioPinPullupEnable()`** function is declared in the *dln_gpio.h* file.

DlnGpioPinPullupDisable() Function

The **`DlnGpioPinPullupDisable()`** deactivates an embedded pull-up resistor for the specified pin. For more information, read Pull-up/Pull-down Resistors.

Syntax

```
DLN_RESULT DlnGpioPinPullupDisable(
    HDLN handle,
    uint8_t pin
 );
```

Parameters

handle

A handle to the DLN-series adapter.

pin

A number of the pin for which you want to disable a pull-up resistor.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function deactivated the pull-up resistor successfully.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the `DlnGpioGetPinCount()` function to find the maximum possible pin number.

Remarks

By default, pull-up resistors are active for GPIO pins. You can activate it manually by using the `DlnGpioPinPullupEnable()` function. To check the current configuration set for a pin, use the `DlnGpioPinPullupIsEnabled()` function.

The **`DlnGpioPinPullupDisable()`** function is declared in the *dln_gpio.h* file.

DlnGpioPinPullupIsEnabled() Function

The **`DlnGpioPinPullupIsEnabled()`** function informs whether an embedded pull-up resistor is currently enabled for the specified pin.

Syntax

```
DLN_RESULT DlnGpioPinPullupIsEnabled(
    HDLN handle,
    uint8_t pin,
    uint8_t* enabled
 );
```

Parameters

handle

A handle to the DLN-series adapter.

pin

A number of the pin from which you want to retrieve information.

enabled

A pointer to an unsigned 8-bit integer. After the function execution, this integer is filled with current state of a pull-up resistor. The following values are available:

Value	Description
1	The pull-up resistor is active.
0	The pull-up resistor is not active.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the current state of the pull-up resistor.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the `DlnGpioGetPinCount()` function to find the maximum possible pin number.

Remarks

By default, the pull-up resistors are active for GPIO pins. You can activate it manually by using the `DlnGpioPinPullupEnable()` function. To deactivate the Pull-up/Pull-down Resistors for GPIO pins, use the `DlnGpioPinPullupDisable()` function.

The `DlnGpioPinPullupIsEnabled()` function is declared in the *dln_gpio.h* file.

DlnGpioPinPulldownEnable() Function

The `DlnGpioPinPulldownEnable()` function activates an embedded pull-down resistor for the specified pin. For more information, read Pull-up/Pull-down Resistors.

Syntax

```
DLN_RESULT DlnGpioPinPulldownEnable(
   HDLN handle,
   uint16_t pin
);
```

Parameters

handle

A handle to the DLN-series adapter.

pin

The number of the pin that you want to configure.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The function successfully activated the pull-down resistor.

DLN_RES_INVALID_PIN_NUMBER

The pin number is not valid. Use the `DlnGpioGetPinCount()` function to find the maximum possible pin number.

DLN_RES_COMMAND_NOT_SUPPORTED

The DLN adapter does not support pull-down resistors.

Remarks

Note: DLN-1 and DLN-2 adapters do not support pull-down resistors.

The **`DlnGpioPinPulldownEnable()`** function is declared in the *dln_gpio.h* file.

DlnGpioPinPulldownDisable() Function

The **`DlnGpioPinPulldownDisable()`** function deactivates an embedded pull-down resistor for the specified pin.

Syntax

```
DLN_RESULT DlnGpioPinPulldownDisable(
   HDLN handle,
   uint16_t pin
);
```

Parameters

handle

A handle to the DLN-series adapter.

pin

The number of the pin that you want to configure.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The function successfully deactivated the pull-down resistor.

DLN_RES_INVALID_PIN_NUMBER

The pin number is not valid. Use the `DlnGpioGetPinCount()` function to find the maximum possible pin number.

DLN_RES_COMMAND_NOT_SUPPORTED

The DLN adapter does not support pull-down resistors.

Remarks

Note: DLN-1 and DLN-2 adapters do not support pull-down resistors.

The **`DlnGpioPinPulldownDisable()`** function is declared in the *dln_gpio.h* file.

DlnGpioPinPulldownIsEnabled() Function

The **`DlnGpioPinPulldownIsEnabled()`** function informs whether an embedded pull-down resistor is active for the specified pin.

Syntax

```
DLN_RESULT DlnGpioPinPulldownIsEnabled(
   HDLN handle,
   uint16_t pin,
   uint8_t* enabled
);
```

Parameters

handle

A handle to the DLN-series adapter.

pin

The number of the pin from which you want to retrieve information.

enabled

A pointer to an unsigned 8-bit integer. After the function execution, the integer is set to the current state of a pull-down resistor.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The function successfully retrieved the current state of the pull-down resistor.

DLN_RES_INVALID_PIN_NUMBER

The pin number is not valid. Use the **`DlnGpioGetPinCount()`** function to find the maximum possible pin number.

DLN_RES_COMMAND_NOT_SUPPORTED

The DLN adapter does not support pull-down resistors.

Remarks

Note: DLN-1 and DLN-2 adapters do not support pull-down resistors.

The **`DlnGpioPinPulldownIsEnable()`** function is declared in the *dln_gpio.h* file.

DlnGpioPinSetCfg() Function

The **`DlnGpioPinSetCfg()`** function changes the configuration of a single GPIO pin. It allows customizing the following:

- Define the pin direction (input or output).
- Enable or disable the open drain, pull up resistor and/or debounce filter.
- Define the output value.

- Define input event parameters.

With this function, you can either reconfigure the pin entirely or change only some of its parameters.

Syntax

```
DLN_RESULT DlnGpioPinSetCfg(
   HDLN handle,
   uint16_t pin,
   uint16_t validFields,
   DLN_GPIO_PIN_CONFIG config
);
```

Parameters

handle

A handle to the DLN-series adapter.

pin

A pin that you want to configure.

validFields

A bit field that defines the configuration parameters that you want to update. Each of the 16 bits of **`validFields`** corresponds to a specific parameter in the **DLN_GPIO_PIN_CONFIG** structure. If you set the bit to 1, the new configuration parameter will be applied. If you set the bit to 0, the configuration parameter will remain unchanged regardless of its value in the **DLN_GPIO_PIN_CONFIG** structure. You can also configure the pin parameters, using the constants declared in the *dln_gpio.h* file. If several constants are used, separate them with "| " (binary "or").

Several bits are reserved for future use and must be set to 0.

Bits	Corresponds to	Constant
0	Bit 0 of DLN_GPIO_PIN_CONFIG::cfg	DLN_GPIO_ENABLE_BIT
1	Bit 1 of DLN_GPIO_PIN_CONFIG::cfg	DLN_GPIO_OUTPUT_BIT
2	Bit 2 of DLN_GPIO_PIN_CONFIG::cfg	DLN_GPIO_OUTPUT_VAL_BIT
3	Bit 3 of DLN_GPIO_PIN_CONFIG::cfg	DLN_GPIO_OPEN_DRAIN_BIT
4	Bit 4 of DLN_GPIO_PIN_CONFIG::cfg	DLN_GPIO_PULL_UP_BIT
5	Bit 5 of DLN_GPIO_PIN_CONFIG::cfg	DLN_GPIO_DEBOUNCE_BIT
6	Reserved	
7	Reserved	
8	DLN_GPIO_PIN_CONFIG::eventType	DLN_GPIO_EVENT_TYPE_BIT
9	DLN_GPIO_PIN_CONFIG::eventPeriod	DLN_GPIO_EVENT_PERIOD_BIT
10	Reserved	
11	Reserved	
12	Reserved	

Bits	Corresponds to	Constant
13	Reserved	
14	Reserved	
15	Reserved	

In order to include a configuration field in the operation, set the corresponding bit to 1. If you set a bit to 0, the field will be ignored.

For example, if you only need to change the **isOutput** and **eventType** settings, the **validFields** byte should look like this: 0000000100000010.

You can configure the pin by using the constants declared in the *dln_gpio.h* file. In this case, the **validFields** byte should look like this:

```
validFields = DLN_GPIO_OUTPUT_BIT | DLN_GPIO_EVENT_TYPE_BIT;
```

config

A new configuration. See the **DLN_GPIO_PIN_CONFIG** structure for details.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The new configuration was successfully applied.

DLN_RES_INVALID_HANDLE (0x8F)

The specified handle is not valid.

DLN_RES_CONNECTION_LOST (0xA0)

The connection to the DLN server was interrupted.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the `DlnGpioGetPinCount()` function to find the maximum possible pin number.

DLN_RES_NON_ZERO_RESERVED_BIT (0xAD)

One or more of the reserved bits **validFields** or **config** parameters are set to 1.

Remarks

The `DlnGpioPinSetCfg()` function is declared in the *dln_gpio.h* file.

DlnGpioPinGetCfg() Function

The `DlnGpioPinGetCfg()` function retrieves the current configuration of the specified GPIO pin.

Syntax

```
DLN_RESULT DlnGpioPinGetCfg(
   HDLN handle,
   uint16_t pin,
   DLN_GPIO_PIN_CONFIG* config
);
```

Parameters

handle

A handle to the DLN-series adapter.

pin

A number of the pin which configuration you want to know.

config

A pointer to the **DLN_GPIO_PIN_CONFIG** structure which is set to the configuration (after the function execution). This structure contains the following fields:

Field	Description
cfg	A bit field that defines the pin configuration.
eventType	A type of the event.
eventPeriod	An interval for repeating events.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the pin configuration.

DLN_RES_INVALID_HANDLE (0x8F)

The specified handle is not valid.

DLN_RES_CONNECTION_LOST (0xA0)

Connection to the DLN server was interrupted.

DLN_RES_INVALID_PIN_NUMBER (0xAB)

The pin number is not valid. Use the `DlnGpioGetPinCount()` function to find the maximum possible pin number.

Remarks

The **`DlnGpioPinGetCfg()`** function is declared in the *dln_gpio.h* file.

6.10 GPIO Event Structure

This section describes the structure used for GPIO events. This structure is declared in the *dln_gpio.h* file.

DLN_GPIO_CONDITION_MET_EV Structure

The `DLN_GPIO_CONDITION_MET_EV` structure contains information about the current GPIO event on a pin.

Syntax

```
typedef struct
{
        DLN_MSG_HEADER header;
        uint16_t eventCount;
        uint8_t eventType;
        uint16_t pin;
        uint8_t value;
} __PACKED_ATTR DLN_GPIO_CONDITION_MET_EV;
```

Members

header

Defines the DLN message header. It should have the following value:
`DLN_MSG_ID_GPIO_CONDITION_MET_EV (0x010F)`

eventCount

The number of events generated in the series. For the `DLN_GPIO_EVENT_ALWAYS` events, this field contains the number of events generated after the event configuration changed. For other events, this field contains the number of events generated since the previous level change.

eventType

The type of generating events. The following values are available:

Value	Constant	Description
0	DLN_GPIO_EVENT_NONE	No events are generated.
1	DLN_GPIO_EVENT_CHANGE	Events are generated when the input value changes.
2	DLN_GPIO_EVENT_LEVEL_HIGH	Events are generated when the high level (logic 1) is detected.
3	DLN_GPIO_EVENT_LEVEL_LOW	Events are generated when the low level (logic 0) is detected.
4	DLN_GPIO_EVENT_ALWAYS	The events are generated periodically with the predefined interval.

For more information, read Digital Input Events.

pin

The number of the pin where the event is generated.

value

The current value of the pin.

Remarks

The **eventCount** parameter contains information about the number of events generated in one event series. A new event series can start if:

- event generation is enabled;
- the event type is changed;
- the level on the pin changed (except for `DLN_GPIO_EVENT_ALWAYS` events).

When a new series of events starts, the **eventCount** is reset to zero. This parameter increases by 1 when a new event in the series is generated.

Find examples of the **eventCount** values in the Digital Input Events section.

6.11 GPIO Structures

This section describes the structures used for the GPIO module. These structures are declared in the *dln_gpio.h* file.

DLN_GPIO_PIN_EVENT_TYPES Structure

The `DLN_GPIO_PIN_EVENT_TYPES` structure contains information about the event types supported for a pin.

Syntax

```
typedef struct
{
    uint8_t count;
    uint8_t eventTypes[8];
} __PACKED_ATTR DLN_GPIO_PIN_EVENT_TYPES;
```

Members

count

The number of available event types.

eventTypes

An 8-element array. Each element corresponds to one of the supported event types.

DLN_GPIO_PIN_CONFIG Structure

The `DLN_GPIO_PIN_CONFIG` structure contains information about the pin configuration.

Syntax

```
typedef struct
{
    uint16_t cfg;
    uint8_t eventType;
    uint16_t eventPeriod;
} __PACKED_ATTR DLN_GPIO_PIN_CONFIG;
```

Members

cfg

A bit field that defines the pin configuration and consists of 16 bits. Each of the bits 0-6, 8-9 corresponds to a specific parameter, that defines the pin configuration. The bits 7, 10-15 are reserved.

Bit	Value	Description	Constant
0	0	The pin is not configured as a GPIO.	DLN_GPIO_DISABLED
0	1	The pin is configured as a GPIO.	DLN_GPIO_ENABLED
1	0	The pin is an input.	DLN_GPIO_INPUT
1	1	The pin is an output.	DLN_GPIO_OUTPUT
2	0	The output value is 0.	DLN_GPIO_OUTPUT_VAL_0
2	1	The output value is 1.	DLN_GPIO_OUTPUT_VAL_1
3	0	The output is open drain.	DLN_GPIO_OPEN_DRAIN_DISABLED
3	1	The output is push-pull.	DLN_GPIO_OPEN_DRAIN_ENABLED
4	0	The pull-up resistor is not active.	DLN_GPIO_PULLUP_DISABLED
4	1	The pull-up resistor is active.	DLN_GPIO_PULLUP_ENABLED
5	0	The debounce filter is not active.	DLN_GPIO_DEBOUNCE_DISABLED
5	1	The debounce filter is active.	DLN_GPIO_DEBOUNCE_ENABLED
6	0	The pull-down resistor is not active.	DLN_GPIO_PULLDOWN_DISABLED
6	1	The pull-down resistor is active.	DLN_GPIO_PULLDOWN_ENABLED
7		Reserved.	
8	0	No events are generated.	DLN_GPIO_EVENT_DISABLED
8	1	Events are generated.	DLN_GPIO_EVENT_ENABLED
9	0	Single events are generated.	DLN_GPIO_EVENT_SINGLE
9	1	Periodical events are generated.	DLN_GPIO_EVENT_PERIODIC
10		Reserved.	
11		Reserved.	
12		Reserved.	

Bit	Value	Description	Constant
13		Reserved.	
14		Reserved.	
15		Reserved.	

eventType

A type of the event. The following values are available:

Value	Constant	Description
0	DLN_GPIO_EVENT_NONE	No events are generated.
1	DLN_GPIO_EVENT_CHANGE	Events are generated when the input value changes.
2	DLN_GPIO_EVENT_LEVEL_HIGH	Events are generated when the high level (logic 1) is detected.
3	DLN_GPIO_EVENT_LEVEL_LOW	Events are generated when the low level (logic 0) is detected.
4	DLN_GPIO_EVENT_ALWAYS	The events are generated periodically with the predefined interval.

For detailed information, read Digital Input Events.

eventPeriod

An interval in milliseconds (ms) at which the events are generated. If the interval is zero, the DLN adapter generates a single event when the level change meets the specified conditions.

7. LED Interface

All DLN-series adapters are fitted with a single status LED and several user-controlled LEDs. The number of user-controlled LEDs depends on the DLN adapter model.

The status LED shows the current state of your device.

Each of the user-controlled LEDs can be adjusted to a specific state by using the `DlnLedSetState()` function. For details, read LED States.

You can find out the number of available user-controlled LEDs by using the `DlnLedGetCount()` function.

Attention: You can purchase DLN-4 devices with an optional enclosure. Such enclosures have no openings for LEDs. Therefore, you will not be able to see the LEDs through the enclosure.

7.1 LED States

DLN-series adapters allow you to control LEDs' behavior by defining the value of the DLN_LED_STATE type defined in the *dln_led.h* file.

The list of its possible LED states is given in the following table:

Led States

State	LED Behavior
DLN_LED_STATE_OFF	Off
DLN_LED_STATE_ON	On
DLN_LED_SLOW_BLINK	Blinking slowly
DLN_LED_FAST_BLINK	Blinking rapidly
DLN_LED_DOUBLE_BLINK	Blinking in a double-blink pattern
DLN_LED_TRIPLE_BLINK	Blinking in a triple-blink pattern

To specify a LED state, use the **`DlnLedSetState()`** function. You can check the current state of a LED by using the **`DlnLedGetState()`** function.

7.2 Simple LED Module Example

This example shows how to manipulate LEDs on the DLN-series adapters. You can find the complete example in the "*..\Program Files\Diolan\DLN\examples\c_cpp\examples\simple*" folder after DLN setup package installation.

```
1   #include "..\..\..\common\dln_generic.h"
2   #include "..\..\..\common\dln_led.h"
3   #pragma comment(lib, "..\\..\\..\\bin\\dln.lib")
4
5
6   int _tmain(int argc, _TCHAR* argv[])
7   {
8   // Open device
9   HDLN device;
10  DlnOpenUsbDevice(&device);
11
12  // Set LED1
13  DlnLedSetState(device, 0, DLN_LED_STATE_FAST_BLINK);
14  // Set LED2
15  DlnLedSetState(device, 1, DLN_LED_STATE_TRIPLE_BLINK);
16
17  // Close device
18  DlnCloseHandle(device);
19  return 0;
20  }
```

Line 1:

The *dln_generic..h* header file declares functions and data structures for the generic interface. In current example this header is used to call DlnOpenUsbDevice() and **`DlnCloseHandle()`** functions.

Line 2:

The dln_led.h header file declares functions and data structures for the LED interface. In current example this header is used to to call **`DlnLedSetState()`** function.

Line 3:

Use *dln.lib* library while project linking.

Line 10: `DlnOpenUsbDevice(&device);`

The function establishes the connection with the DLN adapter. This application uses the USB connectivity of the adapter. For additional options, refer to the Device Opening & Identification section.

Line 13: `DlnLedSetState(device, 0, DLN_LED_STATE_FAST_BLINK);`

Set LED 0 to fast blink state. You can read more about available LED states at LED States section.

Line 15: `DlnLedSetState(device, 1, DLN_LED_STATE_TRIPLE_BLINK);`

Set LED 1 to triple blink state.

Line 18: `DlnCloseHandle(device);`

The application closes the handle to the connected DLN-series adapter.

7.3 LED Functions

This section describes the LED functions. They are used to control and monitor the LED module of a DLN-series adapter.

Actual control of the device is due to use of commands and responses. Each function utilizes respective commands and responses. You can send such commands directly if necessary.

`DlnLedGetCount()`
: Retrieves the total number of user-controlled LEDs

`DlnLedSetState()`
: Configures the LED state.

`DlnLedGetState()`
: Retrieves the current LED state.

All the functions are declared in the *dln_led.h* file.

DlnLedGetCount() Function

The **`DlnLedGetCount()`** function retrieves the total number of user-controlled LEDs available in the device.

Syntax

```
DLN_RESULT DlnLedGetCount(
   HDLN handle,
   uint8_t* count
);
```

Parameters

handle

A handle to the DLN-series adapter.

count

A pointer to an unsigned 8-bit integer. After the function execution, this integer stores the number of user-controlled LEDs.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function was successful.

Remarks

The **`DlnLedGetCount()`** function is declared in the *dln_led.h* file.

DlnLedSetState() Function

The **`DlnLedSetState()`** function sets a new state of the specified user-controlled LED.

Syntax

```
DLN_RESULT DlnLedSetState(
   HDLN handle,
   uint8_t ledNumber,
   DLN_LED_STATE state
);
```

Parameters

handle

A handle to the DLN-series adapter.

ledNumber

The id of the LED which state you want to change.

state

The state of a LED that you want to set.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function was successful.

DLN_RES_INVALID_LED_NUMBER (0xA6)

The specified LED number is not valid. Use the `DlnLedGetCount()` function to find the maximum possible LED number.

Remarks

The `DlnLedSetState()` function is declared in the *dln_led.h* file.

DlnLedGetState() Function

The `DlnLedGetState()` function retrieves the current state of the LED.

Syntax

```
DLN_RESULT DlnLedGetState(
   HDLN handle,
   uint8_t ledNumber
   DLN_LED_STATE* state
);
```

Parameters

handle

A handle to the DLN-series adapter.

ledNumber

A number of the LED which state you want to know.

state

A pointer to the `DLN_LED_STATE` type, where the LED state is stored.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function was successful.

DLN_RES_INVALID_LED_NUMBER (0xA6)

The specified LED number is not valid. Use the `DlnLedGetCount()` function to find the maximum possible LED number.

Remarks

The `DlnLedGetState()` function is declared in the *dln_led.h* file.

8. PWM (Pulse Width Modulation) Interface

Pulse-width modulation (PWM) is the technique that allows to produce variable analog signals using digital means. The average value of the output signal is controlled by switching the digital signal between HIGH (1) and LOW (0) states at a fast rate. The longer the digital signal is in the HIGH state compared to LOW state periods, the higher the output analog signal voltage.

PWM signals are used in a wide variety of control applications. You can use them to control the power supplied to electrical devices like DC motors, valves, pumps, hydraulics and other mechanical parts.

8.1 How PWM Works

A DLN-series adapter generates digital pulses with a certain **frequency**. Each cycle includes the signal in a HIGH (1) and the following LOW (0) states.

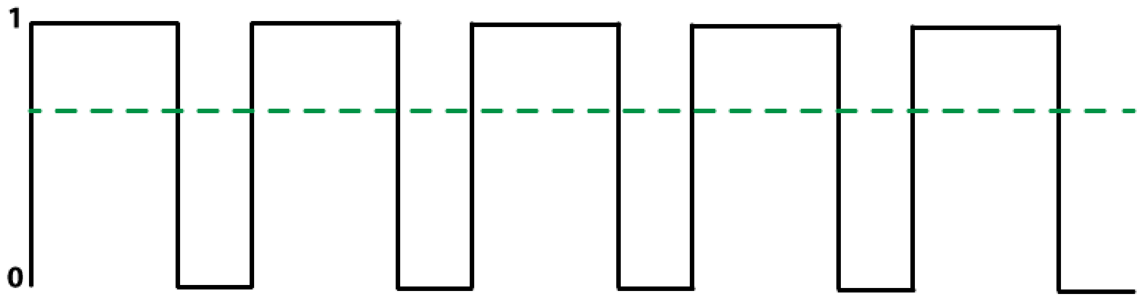
The amount of time the signal is in a HIGH state as a percentage of the total time of one cycle describes the **duty cycle**.

A duty cycle and a frequency are two main parameters that define a PWM signal.

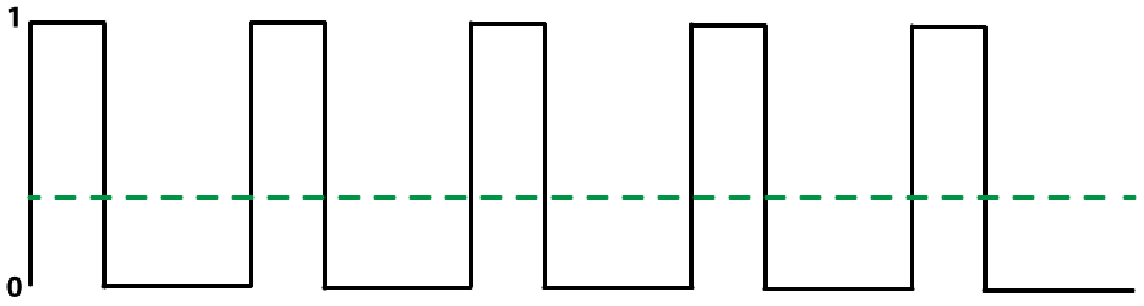
By cycling a digital signal HIGH and LOW at a fast rate and with a certain duty cycle, the PWM output behaves like a constant voltage analog signal when providing power to devices.

Example

Having a DLN adapter that can generate a digital signal either HIGH (3.3V) or LOW (0V), you can create a 2.3V signal with a PWM module specifying a duty cycle as 61%. The PWM module outputs 3.3V 70% of the time. If the PWM frequency is fast enough, then the voltage seen at the output appears to be average voltage and can be calculated by taking the digital high voltage multiplied by the duty cycle: 3.3V x 0.7 = 2.31V.



Selecting a duty cycle 30% would produce 0.99V signal:



The PWM Interface is present in all DLN-series adapters. Some DLN adapters can have more than one PWM ports. Each PWM port includes several channels (Read PWM Channels). The number of ports and channels depends on the type of your DLN adapter.

8.2 Configuring PWM Interface

To start using the PWM Interface, you need to configure and activate the PWM port:

1. For each channel that you suppose to use:

- Configure PWM frequency. The higher frequency, the smoother the output signal becomes. Read PWM Frequency.
- Configure the duty cycle. The level of the output signal depends on the duty cycle value. Read PWM Duty Cycle.

2. Enable all channels that you suppose to use. All channels are equal, but every channel can have specific parameters (frequency and duty cycle). Read PWM Channels.

3. Enable the PWM port. When the PWM port is enabled, you can change configuration for each channel but you cannot activate or release channels.

8.3 PWM Frequency

Your DLN adapter outputs a PWM signal that must be accepted by the device receiving it. The device connected to the DLN adapter requires a particular frequency of the PWM signal.

You can specify the frequency of the PWM calling the `DlnPwmSetFrequency()` function. The frequency value should be within the range supported by your DLN adapter. To check the maximum and minimum PWM frequency values possible for DLN adapters, use the `DlnPwmGetMaxFrequency()` and `DlnPwmGetMinFrequency()` functions. If you enter an incompatible value, the DLN adapter approximates the frequency to the closest lower value supported by the DLN adapter.

The default frequency value is set to 1000Hz. To check the current frequency, use the `DlnPwmGetFrequency()` function.

You can specify individual frequency for each PWM channel.

8.4 PWM Duty Cycle

The duty cycle describes the pulse width as a percentage of the period. By switching the signal level with the specified duty cycle, the output signal will be approximated to the desired level. The 100% duty cycle means that the output level is HIGH. The 0% duty cycle means that the output level is LOW. The 50% duty cycle means that the output level is in the middle between HIGH and LOW.

You can define the duty cycle of the PWM calling the `DlnPwmSetDutyCycle()` function. To check the current duty cycle, use the `DlnPwmGetDutyCycle()` functions.

The default duty cycle value depends on the DLN adapter.

8.5 PWM Channels

Each DLN adapter has several PWM channels. All channels are independent from each other and have similar functionality. You can define frequency and duty cycle for each channel separately.

To know the number of available PWM channels for a specified PWM port, use the **`DlnPwmGetChannelCount()`** function.

To activate the specified channel, call the **`DlnPwmChannelEnable()`** function. You can change frequency and duty cycle values regardless whether the channel is activated or not.

To release the channel, use the **`DlnPwmChannelDisable()`** function.

You cannot activate or release the channel if the appropriate PWM port is enabled. To disable the active port, use the **`DlnPwmDisable()`** function. To enable the PWM port, use the **`DlnPwmEnable()`** function.

8.6 Simple PWM Module Example

The following application shows how to perform an output analog signal using the PWM module. For brevity, this application does not include error detection. You can find the complete example in the "*..\Program Files\Diolan\DLN\examples\c_cpp\examples\simple*" folder after DLN setup package installation.

```
1   #include "..\..\..\common\dln_generic.h"
2   #include "..\..\..\common\dln_pwm.h"
3   #pragma comment(lib, "..\\..\\..\\bin\\dln.lib")
4
5   int _tmain(int argc, _TCHAR* argv[])
6   {
7   // Open device
8   HDLN device;
9   DlnOpenUsbDevice(&device);
10
11  // Set PWM duty cycle
12  double duty;
13  DlnPwmSetDutyCycle(device, 0, 0, 50.0, &duty);
14
15  // Set PWM frequency
16  uint32_t frequency;
17  DlnPwmSetFrequency(device, 0, 0, 1000, &frequency);
18
19  // Enable PWM channel
20  DlnPwmChannelEnable(device, 0, 0);
21  // Enable PWM port
22  uint16_t conflict;
23  DlnPwmEnable(device, 0, &conflict);
24
25  // Wait
26  getchar();
27
28  // Disable PWM port
29  DlnPwmDisable(device, 0);
30  // Disable PWM channel
31  DlnPwmChannelDisable(device, 0, 0);
32  // Close device
33  DlnCloseHandle(device);
34  return 0;
35  }
```

Line 1: `#include "..\..\..\common\dln_generic.h"`

The *dln_generic..h* header file declares functions and data structures for the generic interface. In current example this header is used to call DlnOpenUsbDevice() and **`DlnCloseHandle()`** functions.

Line 2: `#include "..\..\..\common\dln_pwm.h"`

The dln_pwm.h header file declares functions and data structures for the PWM interface. In current example this header is used to call **`DlnPwmSetDutyCycle()`**, **`DlnPwmSetFrequency()`**, **`DlnPwmChannelEnable()`**, **`DlnPwmEnable()`**, **`DlnPwmDisable()`**, **`DlnPwmChannelDisable()`** functions.

Line 3: `#pragma comment(lib, "..\\..\\..\\bin\\dln.lib")`

Use *dln.lib* library while project linking.

Line 9: `DlnOpenUsbDevice(&device);`

The application establishes the connection with the DLN adapter. This application uses the USB connectivity of the adapter. For additional options, refer to the Device Opening & Identification section

Line 13: DlnPwmSetDutyCycle(device, 0, 0, 50.0, &duty);

The application configures the duty cycle of the PWM port 0 channel 0. A duty cycle is the percentage of one period in which a signal is active. You can read more about duty cycle parameter at PWM Duty Cycle section.

Line 17: DlnPwmSetFrequency(device, 0, 0, 1000, &frequency);

The application configures the frequency of the PWM port 0 channel 0. The frequency influences the smoothness of the output signal.

Line 20: `DlnPwmChannelEnable(device, 0, 0);`

The application enables the channel 0 of the PWM port 0. PWM ports can have several channels, each channel can have different configuration. See PWM Channels for details.

Line 23: `DlnPwmEnable(device, 0, &conflict);`

The application enables the PWM port 0. The **`DlnPwmEnable()`** function assigns the corresponding pins to the SPI master module and configures them. If some other module uses a pin required for the SPI bus interface, the **`DlnPwmEnable()`** function returns the **`DLN_RES_PIN_IN_USE`** error code. The **conflictPin** parameter receives the pin's number.

Line 26: `getchar();`

Wait for pressing “Enter” button. The PWM signal will be active before you press “Enter” button.

Line 29: `DlnPwmDisable(device, 0);`

The application releases (disables) the PWM port.

Line 31: `DlnPwmChannelDisable(device, 0, 0);`

The application releases the PWM channel 0 of PWM port 0. The channel cannot be released while the PWM port is enabled.

Line 33: `DlnCloseHandle(device);`

The application closes handle to the DLN adapter.

8.7 PWM Functions

This section describes the PWM Interface functions. They are used to control and monitor the PWM module of a DLN-series adapter.

General information:

`DlnPwmGetPortCount()`
Retrieves the number of ports that can be assigned to the PWM module.

`DlnPwmEnable()`
Assigns a port to the PWM module.

`DlnPwmDisable()`
Unassigns a port to the PWM module.

`DlnPwmIsEnabled()`
Retrieves whether a port is assigned to the PWM module.

`DlnPwmGetChannelCount()`
Retrieves the number of channels available to the PWM port.

`DlnPwmChannelEnable()`
Activates a channel of the PWM port.

`DlnPwmChannelDisable()`
Releases a channel of the PWM port.

`DlnPwmChannelIsEnabled()`
Retrieves whether a channel is activated.

Configuration functions:

`DlnPwmSetDutyCycle()`
Configures the duty cycle value for the specified PWM port.

`DlnPwmGetDutyCycle()`
Retrieves the current duty cycle value for the specified PWM port.

`DlnPwmSetFrequency()`
Configures the frequency value for the specified PWM port.

`DlnPwmGetFrequency()`
Retrieves the current frequency value for the specified PWM port.

`DlnPwmGetMaxFrequency()`
Retrieves the maximum frequency value for the specified PWM port.

`DlnPwmGetMinFrequency()`

Retrieves the minimum frequency value for the specified PWM port.

The *dln_pwm.h* file declares the PWM Interface functions.

DlnPwmGetPortCount() Function

The **`DlnPwmGetPortCount()`** function retrieves the number of PWM ports available in your DLN-series adapter.

Syntax

```
DLN_RESULT DlnPwmGetPortCount(
    HDLN handle,
    uint8_t* count
);
```

Parameters

handle

A handle to the DLN-series adapter.

count

A pointer to an unsigned 8-bit integer that receives the number of available PWM ports.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the number of PWM ports.

Remarks

The **`DlnPwmGetPortCount()`** function is defined in the *dln_pwm.h* file.

DlnPwmEnable() Function

The **`DlnPwmEnable()`** function activates the corresponding PWM port of your DLN-series adapter.

Syntax

```
DLN_RESULT DlnPwmEnable(
    HDLN handle,
    uint8_t port,
    uint16_t* conflict
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the PWM port.

conflict

A pointer to an unsigned 16-bit integer that receives the number of the conflicted pin, if any.

A conflict arises if a pin is already assigned to another module of the DLN-series adapter and cannot be used for the PWM module. To fix this, check which module uses the pin (call the `DlnGetPinCfg()` function), disconnect the pin from that module and call the **`DlnPwmEnable()`** function once again. In case there still are conflicted pins, only the number of the next one will be returned.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully activated the PWM port.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnPwmGetPortCount()` function to find the maximum possible port number.

DLN_RES_PIN_IN_USE (0xA5)

The port cannot be activated as the PWM port because one or more pins of the port are assigned to another module. The **conflict** parameter contains the number of a conflicting pin.

Remarks

The **`DlnPwmEnable()`** function is defined in the *dln_pwm.h* file.

DlnPwmDisable() Function

The **`DlnPwmDisable()`** function releases the specified PWM port of your DLN-series adapter.

Syntax

```
DLN_RESULT DlnPwmDisable(
    HDLN handle,
    uint8_t port
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the PWM port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully released the PWM port.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnPwmGetPortCount()` function to find the

maximum possible port number.

Remarks

The **`DlnPwmDisable()`** function is defined in the *dln_pwm.h* file.

DlnPwmIsEnabled() Function

The **`DlnPwmIsEnabled()`** function retrieves information, whether the specified port is assigned to the PWM module.

Syntax

```
DLN_RESULT DlnPwmIsEnabled(
    HDLN handle,
    uint8_t port,
    uint8_t* enabled
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the PWM port.

enabled

A pointer to an unsigned 8-bit integer. The integer will be filled with the information whether the specified PWM port is activated. There are two possible values:

- 0 or `DLN_PWM_DISABLED` - PWM port is deactivated.
- 1 or `DLN_PWM_ENABLED` - PWM port is activated.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the PWM port state.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnPwmGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnPwmIsEnabled()`** function is defined in the *dln_pwm.h* file.

DlnPwmGetChannelCount() Function

The **`DlnPwmGetChannelCount()`** function retrieves the number of PWM channels available in the specified PWM-port of your DLN-series adapter.

Syntax

```
DLN_RESULT DlnPwmGetChannelCount(
    HDLN handle,
    uint8_t port,
    uint8_t* count
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A PWM port to retrieve the number of channels from.

count

A pointer to an unsigned 8-bit integer that receives the available number of channels in the specified PWM port of the DLN-series adapter.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the number of channels.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnPwmGetPortCount()` function to find the maximum possible port number.

Remarks

The `DlnPwmGetChannelCount()` function is defined in the *dln_pwm.h* file.

DlnPwmChannelEnable() Function

The `DlnPwmChannelEnable()` function activates the specified channel from the corresponding PWM port of your DLN-series adapter.

Syntax

```
DLN_RESULT DlnPwmChannelEnable(
    HDLN handle,
    uint8_t port,
    uint8_t channel
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the PWM port.

channel

A number of the channel.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully activated the channel.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnPwmGetPortCount()` function to find the maximum possible port number.

DLN_RES_INVALID_CHANNEL_NUMBER (0xC0)

The channel number is not valid. Use the `DlnPwmGetChannelCount()` function to find the maximum possible port number.

Remarks

The `DlnPwmChannelEnable()` function is defined in the *dln_pwm.h* file.

DlnPwmChannelDisable() Function

The `DlnPwmChannelDisable()` function releases the specified channel from the corresponding PWM port of your DLN-series adapter.

Syntax

```
DLN_RESULT DlnPwmChannelDisable(
    HDLN handle,
    uint8_t port,
    uint8_t channel
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the PWM port.

channel

A number of the channel.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully released the channel.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnPwmGetPortCount()` function to find the maximum possible port number.

DLN_RES_INVALID_CHANNEL_NUMBER (0xC0)

The channel number is not valid. Use the `DlnPwmGetChannelCount()` function to find the maximum possible port number.

Remarks

The **`DlnPwmChannelDisable()`** function is defined in the *dln_pwm.h* file.

DlnPwmChannelIsEnabled() Function

The **`DlnPwmChannelIsEnabled()`** retrieves information, whether the specified PWM channel is activated.

Syntax

```
DLN_RESULT DlnPwmChannelIsEnabled(
    HDLN handle,
    uint8_t port,
    uint8_t channel,
    uint8_t* enabled
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the PWM port.

channel

A number of the PWM channel.

enabled

A pointer to an unsigned 8-bit integer that receives the information about the specified PWM channel. There are two possible values:

- 0 or **`DLN_PWM_DISABLED`** - The PWM channel is released.
- 1 or **`DLN_PWM_ENABLED`** - The PWM channel is activated.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved information about the channel.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnPwmGetPortCount()` function to find the maximum possible port number.

DLN_RES_INVALID_CHANNEL_NUMBER (0xC0)

The channel number is not valid. Use the `DlnPwmGetChannelCount()` function to find the maximum possible port number.

Remarks

The `DlnPwmChannelIsEnabled()` function is defined in the *dln_pwm.h* file.

DlnPwmSetDutyCycle() Function

The `DlnPwmSetDutyCycle()` function defines a PWM duty cycle value, which is the ratio of the high time to the PWM period.

Syntax

```
DLN_RESULT DlnPwmSetDutyCycle(
   HDLN handle,
   uint8_t port,
   uint8_t channel,
   double dutyCycle,
   double* actualDutyCycle
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the PWM port.

channel

A number of the PWM channel.

dutyCycle

A double precision floating point number. Must contain a duty cycle that should to be set. The value is specified in percents.

actualDutyCycle

A pointer to a double precision floating point number that receives the actual duty cycle value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully defined the duty cycle value.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnPwmGetPortCount()` function to find the maximum possible port number.

DLN_RES_INVALID_CHANNEL_NUMBER (0xC0)

The channel number is not valid. Use the `DlnPwmGetChannelCount()` function to find the maximum possible port number.

DLN_RES_PWM_INVALID_DUTY_CYCLE (0xC4)

The provided value is not valid.

Remarks

The **`DlnPwmSetDutyCycle()`** function is defined in the *dln_pwm.h* file.

DlnPwmGetDutyCycle() Function

The **`DlnPwmGetDutyCycle()`** function retrieves the current PWM duty cycle value.

Syntax

```
DLN_RESULT DlnPwmGetDutyCycle(
   HDLN handle,
   uint8_t port,
   uint8_t channel,
   double* dutyCycle
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the PWM port.

channel

A number of the PWM channel.

dutyCycle

A pointer to the double precision floating point number that receives the current duty cycle.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the current duty cycle value.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnPwmGetPortCount()`** function to find the maximum possible port number.

DLN_RES_INVALID_CHANNEL_NUMBER (0xC0)

The channel number is not valid. Use the **`DlnPwmGetChannelCount()`** function to find the maximum possible port number.

Remarks

The **`DlnPwmGetDutyCycle()`** function is defined in the *dln_pwm.h* file.

DlnPwmSetFrequency() Function

The **`DlnPwmSetFrequency()`** function defines the PWM frequency.

Syntax

```
DLN_RESULT DlnPwmSetFrequency(
   HDLN handle,
   uint8_t port,
   uint8_t channel,
   uint32_t frequency,
   uint32_t* actualFrequency
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the PWM port.

channel

A number of the PWM channel.

frequency

The frequency value, specified in Hz. You may specify any value within the range, supported by the DLN-series adapter. This range can be retrieved using the respective function. If you enter an incompatible value, it will be approximated as the closest lower frequency value, supported by the adapter.

actualFrequency

A pointer to an unsigned 32-bit integer that receives the actually set frequency approximated as the closest to user-defined lower value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully defined the frequency value.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnPwmGetPortCount()` function to find the maximum possible port number.

DLN_RES_INVALID_CHANNEL_NUMBER (0xC0)

The channel number is not valid. Use the `DlnPwmGetChannelCount()` function to find the maximum possible port number.

DLN_RES_VALUE_ROUNDED (0x21)

The function approximated the frequency value to the closest supported value.

Remarks

The `DlnPwmSetFrequency()` function is defined in the *dln_pwm.h* file.

DlnPwmGetFrequency() Function

The `DlnPwmGetFrequency()` function retrieves the current PWM frequency.

Syntax

```
DLN_RESULT DlnPwmGetFrequency(
  HDLN handle,
  uint8_t port,
  uint8_t channel,
  uint32_t* frequency,
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the PWM port.

channel

A number of the PWM channel.

frequency

A pointer to an unsigned 32-bit integer that receives the current frequency setting for the DLN-series adapter.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the current frequency value.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnPwmGetPortCount()` function to find the maximum possible port number.

DLN_RES_INVALID_CHANNEL_NUMBER (0xC0)

The channel number is not valid. Use the `DlnPwmGetChannelCount()` function to find the maximum possible port number.

Remarks

The **`DlnPwmGetFrequency()`** function is defined in the *dln_pwm.h* file.

DlnPwmGetMaxFrequency() Function

The **`DlnPwmGetMaxFrequency()`** function retrieves the maximum possible value of the PWM frequency.

Syntax

```
DLN_RESULT DlnPwmGetMaxFrequency(
    HDLN handle,
    uint8_t port,
    uint32_t *frequency
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the PWM port.

frequency

A pointer to an unsigned 32-bit integer that receives the maximum frequency value for the DLN-series adapter.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the maximum possible frequency value.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnPwmGetPortCount()` function to find the maximum possible port number.

DLN_RES_INVALID_CHANNEL_NUMBER (0xC0)

The channel number is not valid. Use the `DlnPwmGetChannelCount()` function to find the maximum possible port number.

Remarks

The **`DlnPwmGetMaxFrequency()`** function is defined in the *dln_pwm.h* file.

DlnPwmGetMinFrequency() Function

The **`DlnPwmGetMinFrequency()`** function retrieves the minimum possible value of the PWM frequency.

Syntax

```
DLN_RESULT DlnPwmGetMinFrequency(
    HDLN handle,
    uint8_t port,
    uint32_t *frequency
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the PWM port.

frequency

A pointer to an unsigned 32-bit integer that receives the minimum frequency value for the DLN-series adapter.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the minimum possible frequency value.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnPwmGetPortCount()` function to find the maximum possible port number.

DLN_RES_INVALID_CHANNEL_NUMBER (0xC0)

The channel number is not valid. Use the `DlnPwmGetChannelCount()` function to find the maximum possible port number.

Remarks

The `DlnPwmGetMinFrequency()` function is defined in the *dln_pwm.h* file.

9. Pulse Counter Interface

Pulse counter interface allows to count with pulses and set timers.

To start using Pulse counter module you need to enable Pulse counter port. It can be done with the `DlnPlsCntEnable()` function. Also pulse counter module mode need to be set with `DlnPlsCntSetMode()` function.

At any moment pulse counter or timer value can be read with function `DlnPlsCntGetValue()`.

You can also enable the events to get alert when condition is met. One of the few conditions can be selected: pulse counter/timer overflow, when it matches selected value or repeat event for selected period. Use `DlnPlsCntSetEventCfg()` to setup events.

9.1 Pulse Counter Modes

There are 3 available modes: free run mode, time-based mode and pulse-based mode.

In "**Free Run Mode**" pulses are counted continuously. You can suspend, resume or reset the counter and get the number of pulses at any time.

In "**Time-Based Mode**" pulses are counted during the user-defined time period. When the predefined time period (limit) is exceeded, the counting starts again from 0. The pulse counter can send a match event to PC if activated. The event contains the number of pulses detected during this period.

In "**Pulse-Based Mode**" Pulses are counted until the number of pulses reaches the user-defined value (limit). Then the counting starts again from 0. The counter can send an event to PC if activated. The event contains time elapsed from the moment you started the counter.

Pulse counter module mode can be set or changes by calling **`DlnPlsCntSetMode()`** function. You can also retrieve the current mode by calling **`DlnPlsCntGetMode()`** function.

There are also constants available for each mode in the *dln_pls_cnt.h* header file: **`DLN_PLS_CNT_MODE_FREE_RUN(0)`**, **`DLN_PLS_CNT_MODE_TIME_BASED(1)`**, **`DLN_PLS_CNT_MODE_PULSE_BASED(2)`**.

9.2 Simple Pulse Counter Module Example

This example shows how to setup pulse counter module, read timer and pulse counter values. You can find the complete example in the "*..\Program Files\Diolan\DLN\examples\c_cpp\examples\simple*" folder after DLN setup package installation.

```
1   #include "..\..\..\common\dln_generic.h"
2   #include "..\..\..\common\dln_pls_cnt.h"
3   #pragma comment(lib, "..\\..\\..\\bin\\dln.lib")
4
5
6   int _tmain(int argc, _TCHAR* argv[])
7   {
8   // Open device
9   HDLN device;
10  DlnOpenUsbDevice(&device);
11
12  // Set Free Run mode
13  DlnPlsCntSetMode(device, 0, DLN_PLS_CNT_MODE_FREE_RUN, 0);
14  // Enable counter
15  uint16_t conflict;
16  DlnPlsCntEnable(device, 0, &conflict);
17
18  // Do some delay
19  for (int i = 0; i < 10000; i++);
20
21  // Read counter values
22  uint32_t timerValue, counterValue;
23  DlnPlsCntGetValue(device, 0, &timerValue, &counterValue);
24  printf("Timer Value = %u, Counter Value = %u\n", timerValue,
counterValue);
25
26  // Close device
27  DlnCloseHandle(device);
28  return 0;
29  }
```

Line 1: `#include "..\..\..\common\dln_generic.h"`

The *dln_generic..h* header file declares functions and data structures for the generic interface. In current example this header is used to call DlnOpenUsbDevice() and **`DlnCloseHandle()`** functions.

Line 2: `#include "..\..\..\common\dln_pls_cnt.h"`

The dln_pwm.h header file declares functions and data structures for the PWM interface. In current example this header is used to call **`DlnPlsCntSetMode()`**, **`DlnPlsCntEnable()`**, **`DlnPlsCntGetValue()`** functions.

Line 3: `#pragma comment(lib, "..\\..\\..\\bin\\dln.lib")`

Use *dln.lib* library while project linking.

Line 10: `DlnOpenUsbDevice(&device);`

The function establishes the connection with the DLN adapter. This application uses the USB connectivity of the adapter. For additional options, refer to the Device Opening & Identification section.

Line 13: `DlnPlsCntSetMode(device, 0, DLN_PLS_CNT_MODE_FREE_RUN, 0);`

The function sets pulse counter mode to “Free run mode”. You can read more about pulse counter modes at Pulse Counter Modes section.

Line 16: `DlnPlsCntEnable(device, 0, &conflict);`

The function enables pulse counter module.

Line 19: `for (int i = 0; i < 10000; i++);`

Performing delay for more continuous mode of counter.

Line 23: `DlnPlsCntGetValue(device, 0, &timerValue, &counterValue);`

The function retrieves timer and pulse counter values.

Line 24: `printf("Timer Value = %u, Counter Value = %u\n", timerValue, counterValue);`

Printing retrieved timer and pulse counter values to console.

Line 27: `DlnCloseHandle(device);`

The application closes handle to the DLN adapter.

9.3 Pulse Counter Functions

This section describes the Pulse Counter module functions. They are used to control Pulse Counter interface and modify its settings.

Actual control of the device is performed by use of commands and responses. Each function utilizes respective commands and responses. You can send such commands directly if necessary.

DlnPlsCntGetPortCount() Function

The **`DlnPlsCntGetPortCount()`** function retrieves the number of pulse counter ports available in your DLN-series adapter.

Syntax

```
DLN_RESULT DlnPlsCntGetPortCount(
  HDLN handle,
  uint8_t *count
);
```

Parameters

handle

A handle to the DLN-series adapter.

count

A pointer to an unsigned 8-bit integer that receives the number of available Pulse counter ports.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the number of available pulse counter ports.

Remarks

The **`DlnPlsCntGetPortCount()`** function is defined in the *dln_pls_cnt.h* file.

DlnPlsCntEnable() Function

The **`DlnPlsCntEnable()`** function configures a port as Pulse counter.

Syntax

```
DLN_RESULT DlnPlsCntEnable(
  HDLN handle,
  uint8_t port,
  uint16_t *conflict
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the port.

conflict

A pointer to an unsigned 16-bit integer that receives the conflicted pin information.

A conflict arises if any pin of the port is already assigned to another module of the DLN adapter and cannot be used for the Pulse counter module. To fix this, check which module uses the pin (call the **`DlnGetPinCfg()`** function), disconnect the pin from that module and call the **`DlnPlsCntEnable()`** function once again. In case there still are conflicted pins, the number of the next one will be returned.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured the port as Pulse counter.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnPlsCntGetPortCount()`** function to find the maximum possible port number.

DLN_RES_PIN_IN_USE (0xA5)

The port cannot be activated as Pulse counter because one or more pins of the port are assigned to another module. The **conflict** parameter contains the number of a conflicting pin.

DLN_RES_NO_FREE_TIMER (0xD0)

Remarks

The **`DlnPlsCntEnable()`** function is defined in the *dln_pls_cnt.h* file.

DlnPlsCntDisable() Function

The **`DlnPlsCntDisable()`** function releases a port from Pulse counter.

Syntax

```
DLN_RESULT DlnPlsCntDisable(
  HDLN handle,
  uint8_t port
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully released the port from Pulse counter.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnPlsCntGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnPlsCntDisable()`** function is defined in the *dln_pls_cnt.h* file.

DlnPlsCntIsEnabled() Function

The **`DlnPlsCntIsEnabled()`** function informs whether a port is currently configured as Pulse counter.

Syntax

```
DLN_RESULT DlnPlsCntIsEnabled(
  HDLN handle,
  uint8_t port,
  uint8_t *enabled
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the port.

enabled

A pointer to an unsigned 8-bit integer that receives the current port configuration.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved information about the Pulse counter port.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnPlsCntGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnPlsCntIsEnabled()`** function is defined in the *dln_pls_cnt.h* file.

DlnPlsCntSetMode() Function

The **`DlnPlsCntSetMode()`** function is used to set mode and limit parameters of specified Pulse Counter port.

Syntax

```
DLN_RESULT DlnPlsCntSetMode(
  HDLN handle,
  uint8_t port,
  uint8_t mode,
  uint32_t limit
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the port to retrieve information about.

mode

Defines the mode parameter of the Pulse Counter port.

Pulse Counter Module Modes

Mode Id	Mode Title	Mode Description
0	Free Run Mode	Pulses are counted continuously. You can suspend, resume or reset the counter and get the number of pulses at any time.
1	Time Based Mode	Pulses are counted during the user-defined time period. When the predefined time period (limit) is exceeded, the counting starts again from 0. The pulse counter can send a match event to PC if activated. The event contains the number of pulses detected during this period.
2	Pulse Based Mode	Pulses are counted until the number of pulses reaches the user-defined value (limit). Then the counting starts again from 0. The counter can send an event to PC if activated. The event contains time elapsed from the moment you started the counter.

limit

Defines the limit parameter of the Pulse Counter port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured the Pulse counter port.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnPlsCntGetPortCount()` function to find the maximum possible port number.

DLN_RES_OVERFLOW (0xB5)

DLN_RES_BUSY (0xB6)

The function cannot configure the Pulse counter port while it is busy.

DLN_RES_INVALID_MODE (0xC7)

The mode value is not valid.

Remarks

The **`DlnPlsCntSetMode()`** function is defined in the *dln_pls_cnt.h* file.

DlnPlsCntGetMode() Function

The **`DlnPlsCntGetMode()`** function is used to get mode and limit parameters of the specified Pulse Counter port.

Syntax

```
DLN_RESULT DlnPlsCntGetMode(
  HDLN handle,
  uint8_t port,
  uint8_t *mode,
  uint32_t *limit
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the port.

mode

A pointer to an unsigned 8-bit integer that receives the current mode parameter of the Pulse Counter port.

limit

A pointer to an unsigned 32-bit integer that receives the current limit parameter of the Pulse Counter port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the Pulse counter port configuration.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnPlsCntGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnPlsCntGetMode()`** function is defined in the *dln_pls_cnt.h* file.

DlnPlsCntGetResolution() Function

The **`DlnPlsCntGetResolution()`** function retrieves the current Pulse Counter Module resolution.

Syntax

```
DLN_RESULT DlnPlsCntGetResolution(
  HDLN handle,
  uint8_t port,
  uint8_t *resolution
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the port to be configured.

resolution

A pointer to an unsigned 8-bit integer that receives the resolution value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the Pulse counter module resolution.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnPlsCntGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnPlsCntGetResolution()`** function is defined in the *dln_pls_cnt.h* file.

DlnPlsCntGetValue() Function

The **`DlnPlsCntGetValue()`** function retrieves the current timer and counter values of the specified Pulse Counter port.

Syntax

```
DLN_RESULT DlnPlsCntGetValue(
  HDLN handle,
  uint8_t port,
  uint32_t *timerValue,
  uint32_t *counterValue
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the port.

timerValue

A pointer to an unsigned 32-bit integer that receives the timer value.

counterValue

A pointer to an unsigned 32-bit integer that receives the counter value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the timer and counter values.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnPlsCntGetPortCount()` function to find the maximum possible port number.

Remarks

The **`DlnPlsCntGetValue()`** function is defined in the **`dln_pls_cnt.h`** file.

DlnPlsCntReset() Function

The **`DlnPlsCntReset()`** function resets the timer and/or counter on the Pulse Counter port.

Syntax

```
DLN_RESULT DlnPlsCntReset(
  HDLN handle,
  uint8_t port,
  uint8_t resetTimer,
  uint8_t resetCounter
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the port.

resetTimer

Defines whether the timer needs to be reset. To reset the timer, specify any value greater than 0 to this parameter.

resetCounter

Defines whether counter need to be reset. To reset the counter, specify any value greater than 0 to this parameter.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully reset the timer and/or counter values.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnPlsCntGetPortCount()` function to find the maximum possible port number.

Remarks

The **`DlnPlsCntReset()`** function is defined in the *dln_pls_cnt.h* file.

DlnPlsCntResume() Function

The **`DlnPlsCntResume()`** function resumes the pulse counter operation state on the Pulse Counter port.

Syntax

```
DLN_RESULT DlnPlsCntResume(
  HDLN handle,
  uint8_t port
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully resumed the pulse counter operation state.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnPlsCntGetPortCount()` function to find the maximum possible port number.

Remarks

The **`DlnPlsCntResume()`** function is defined in the *dln_pls_cnt.h* file.

DlnPlsCntSuspend() Function

The **`DlnPlsCntSuspend()`** function suspends pulse counter operation state on the Pulse Counter port.

Syntax

```
DLN_RESULT DlnPlsCntSuspend(
  HDLN handle,
  uint8_t port
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully suspended the pulse counter operation state.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the **`DlnPlsCntGetPortCount()`** function to find the maximum possible port number.

Remarks

The **`DlnPlsCntSuspend()`** function is defined in the *dln_pls_cnt.h* file.

DlnPlsCntIsSuspended() Function

The **`DlnPlsCntIsSuspended()`** function informs whether a port is currently suspended.

Syntax

```
DLN_RESULT DlnPlsCntIsSuspended(
  HDLN handle,
  uint8_t port,
  uint8_t *suspended
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the port.

suspended

A pointer to an unsigned 8-bit integer that receives the current port configuration.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the current port configuration.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnPlsCntGetPortCount()` function to find the maximum possible port number.

Remarks

The `DlnPlsCntIsSuspended()` function is defined in the *dln_pls_cnt.h* file.

DlnPlsCntSetEventCfg() Function

The `DlnPlsCntSetEventCfg()` function defines the event configuration for the specified Pulse Counter port.

Syntax

```
DLN_RESULT DlnPlsCntSetEventCfg(
  HDLN handle,
  uint8_t port,
  uint8_t eventType,
  uint32_t repeatInterval
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the port.

eventType

Defines the event type parameter. There are 4 possible event types.

- DLN_PLS_CNT_EVENT_NONE - 0
- DLN_PLS_CNT_EVENT_OVERFLOW - 1
- DLN_PLS_CNT_EVENT_MATCH - 2
- DLN_PLS_CNT_EVENT_REPEAT - 4

repeatInterval

Defines the repeat interval parameter.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully configured events for the specified port.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnPlsCntGetPortCount()` function to find the maximum possible port number.

DLN_RES_INVALID_EVENT_TYPE (0xA9)

The specified event type is not valid.

DLN_RES_INVALID_EVENT_PERIOD (0xAC)

The specified event period is not valid.

Remarks

The `DlnPlsCntSetEventCfg()` function is defined in the *dln_pls_cnt.h* file.

DlnPlsCntGetEventCfg() Function

The `DlnPlsCntGetEventCfg()` function is used to get event configuration of the specified Pulse Counter port.

Syntax

```
DLN_RESULT DlnPlsCntGetEventCfg(
  HDLN handle,
  uint8_t port,
  uint8_t *eventType,
  uint32_t *repeatInterval
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the port to get configuration.

eventType

A pointer to an unsigned 8-bit integer. Defines the current event type parameter.

repeatInterval

A pointer to an unsigned 32-bit integer. Defines the current repeat interval parameter

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS (0x00)

The function successfully retrieved the current event configuration.

DLN_RES_INVALID_PORT_NUMBER (0xA8)

The port number is not valid. Use the `DlnPlsCntGetPortCount()` function to find the maximum possible port number.

Remarks

The **`DlnPlsCntGetEventCfg()`** function is defined in the *dln_pls_cnt.h* file.

10. UART Interface

UART (A Universal Asynchronous Receiver/Transmitter) interface is supported only by DLN-4M and DLN-4S adapters.

UART is a interface which is included in micro-controller, that translates data between parallel and serial forms. UART is commonly used in conjunction with communication standards such as EIA, RS-232, RS-422 or RS-485, so that it can “talk” to and exchange data with modems and other serial devices.

10.1 Simple UART Module Example

This example shows how to setup and send data with UART module. You can find the complete example in the “*..\Program Files\Diolan\DLN\examples\c_cpp\examples\simple*” folder after DLN setup package installation.

```
#include "..\..\..\common\dln_generic.h"
#include "..\..\..\common\dln_uart.h"
#pragma comment(lib, "..\\..\\..\\bin\\dln.lib")

int _tmain(int argc, _TCHAR* argv[])
{
// Open device
HDLN device;
DlnOpenUsbDevice(&device);

// Configure UART to 19200 8N1
uint32_t baud;
DlnUartSetBaudrate(device, 0, 19200, &baud);
DlnUartSetCharacterLength(device, 0, DLN_UART_CHARACTER_LENGTH_8);
DlnUartSetParity(device, 0, DLN_UART_PARITY_NONE);
DlnUartSetStopbits(device, 0, DLN_UART_STOPBITS_1);
// Enable UART
uint16_t conflict;
DlnUartEnable(device, 0, &conflict);

// Write data
uint8_t output[10] = "123456789";
DlnUartWrite(device, 0, 10, output);

// Read data
uint8_t input[10];
uint16_t total;
DlnUartRead(device, 0, 10, &input, &total);
// Print it
for (int i = 0; i < total; i++) putchar(input[i]);

// Disable UART
33 DlnUartDisable(device, 0);
34
35 // Close device
36 DlnCloseHandle(device);
37 return 0;
38 }
```

Line 1: `#include "..\..\..\common\dln_generic.h"`

The *dln_generic..h* header file declares functions and data structures for the generic interface. In current example this header is used to call DlnOpenUsbDevice() and **`DlnCloseHandle()`** functions.

Line 2: `#include "..\..\..\common\dln_uart.h"`

The dln_uart.h header file declares functions and data structure specific for UART interface. By including this header file you are able to call **`DlnUartSetBaudrate()`**, **`DlnUartSetCharacterLength()`**, **`DlnUartSetParity()`**, **`DlnUartSetStopbits()`**, **`DlnUartEnable()`**, **`DlnUartWrite()`**, **`DlnUartRead()`**, **`DlnUartDisable()`** and other UART interface functions.

Line 3: `#pragma comment(lib, "..\\..\\..\\bin\\dln.lib")`

Use *dln.lib* library while project linking.

Line 9: `DlnOpenUsbDevice(&device);`

The function establishes the connection with the DLN adapter. This application uses the USB connectivity of the adapter. For additional options, refer to the Device Opening & Identification section.

Line 13: `DlnUartSetBaudrate(device, 0, 19200, &baud);`

The function sets the baud rate value equal to 10200 bits.

Line 14: `DlnUartSetCharacterLength(device, 0, DLN_UART_CHARACTER_LENGTH_8);`

The function sets the character length equal to 8 bits.

Line 15: `DlnUartSetParity(device, 0, DLN_UART_PARITY_NONE);`

The function sets parity to none.

Line 16: `DlnUartSetStopbits(device, 0, DLN_UART_STOPBITS_1);`

The function sets the number of stop bits to be sent with each character.

line 19: `DlnUartEnable(device, 0, &conflict);`

The function enables the UART interface.

Line 22: `uint8_t output[10] = "123456789";`

Define the array of 10 8-bits unsigned integer values (10 bytes) and set it to “123456789”.

Line 23: `DlnUartWrite(device, 0, 10, output);`

The function sends 10 bytes data buffer via UART interface.

Line 28: `DlnUartRead(device, 0, 10, &input, &total);`

The function reads 10 bytes data buffer to input variable and returns the value of real read bytes to total variable.

Line 30: `for (int i = 0; i < total; i++) putchar(input[i]);`

Output input array values in loop.

Line 33: `DlnUartDisable(device, 0);`

Disable UART interface port 0.

Line 36: `DlnCloseHandle(device);`

The application closes handle to the DLN adapter.

10.2 UART Functions

This section describes the UART functions. They are used to control UART interface and modify its settings.

Actual control of the device is performed by use of commands and responses. Each function utilizes respective commands and responses. You can send such commands directly if necessary.

DlnUartGetPortCount() Function

The **`DlnUartGetPortCount()`** function retrieves the total number of the UART ports available in your DLN-series adapter.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartGetPortCount(
    HDLN handle,
    uint8_t *count
);
```

Parameters

handle

A handle to the DLN-series adapter.

count

A pointer to an unsigned 8-bit integer. This integer will be filled with the number of available ports after function execution.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The UART port count has been successfully retrieved.

DlnUartEnable() Function

The **`DlnUartEnable()`** function enables UART port.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartEnable(
    HDLN handle,
    uint8_t port,
    uint16_t *conflict
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the UART port to be enabled.

conflict

A pointer to an unsigned 16-bit integer. This integer can be filled with a number of the conflicted pin, if any.

A conflict arises if a pin is already assigned to another module of the DLN-series adapter and cannot be used by the UART module. To fix this a user has to disconnect a pin from a module that it has been assigned to and call the **`DlnUartEnable()`** function once again. If there are any more conflicting pins, the next conflicted pin number will be returned.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and UART port was enabled.

DlnUartDisable() Function

The **`DlnUartDisable()`** function deactivates corresponding UART port on your DLN-Series adapter.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartDisable(
    HDLN handle,
    uint8_t port
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the SPI master port to be enabled.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and UART port was disabled.

DlnUartIsEnabled() Function

The **`DlnUartIsEnabled()`** function retrieves information whether the specified SPI master port is activated.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartIsEnabled(
    HDLN handle,
    uint8_t port,
    uint8_t *enabled
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the UART port to retrieve the information from.

enabled

A pointer to an unsigned 8-bit integer. The integer will be filled with information whether the specified UART port is activated after the function execution. There are two possible values:

- 0 or DLN_UART_DISABLED - the port is not configured as UART.
- 1 or DLN_UART_ENABLED - the port is configured as UART.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and UART port state was retrieved.

DlnUartSetBaudrate() Function

The **`DlnUartSetBaudrate()`** function sets the baud rate, which represents the number of bits that are actually being sent.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartSetBaudrate(
    HDLN handle,
    uint8_t port,
    uint32_t baudrate,
    uint32_t *actualBaudrate
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the UART port to be configured.

baudrate

The Baud rate value. The minimal baud rate value can be achieved by **`DlnUartGetMinBaudrate()`** function. The maximum baud rate value can be achieved by using **`DlnUartGetMaxBaudrate()`** function.

actualBaudrate

The pointer to uint32_t type value, which will be filled with actual Baud rate value to be set. This value is rounded value of **baudrate** parameter.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and baud rate has been set.

DlnUartGetBaudrate() Function

The **`DlnUartGetBaudrate()`** function retrieves the baud rate value, which represents the number of bits that are actually being sent.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartGetBaudrate(
    HDLN handle,
    uint8_t port,
    uint32_t *baudrate
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the UART port.

baudrate

The current baud rate value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and baud rate has been retrieved.

DlnUartSetCharacterLength() Function

The **`DlnUartSetCharacterLength()`** function selects the data character length of 5, 6, 7, 8 and 9 bits per character.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartSetCharacterLength(
    HDLN handle,
    uint8_t port,
    uint8_t characterLength
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the UART port to be configured.

characterLength

Set the data character length value of 5, 6, 7, 8 and 9 bits per character. You can use predefined constants to set required value.

- 5 or DLN_UART_CHARACTER_LENGTH_5
- 6 or DLN_UART_CHARACTER_LENGTH_6

- 7 or DLN_UART_CHARACTER_LENGTH_7
- 8 or DLN_UART_CHARACTER_LENGTH_8
- 9 or DLN_UART_CHARACTER_LENGTH_9

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and character length has been set.

DlnUartGetCharacterLength() Function

The **`DlnUartGetCharacterLength()`** function retrieves the current data character length.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartGetCharacterLength(
    HDLN handle,
    uint8_t port,
    uint8_t *characterLength
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the UART port.

characterLength

The pointer to uint8_t type variable. Retrieve the current data character length value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and character length has been retrieved successfully.

DlnUartSetParity() Function

The **`DlnUartSetParity()`** function selects even, odd, mark (when the parity bit is always 1), space (the bit is always 0) or none parity.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartSetParity(
    HDLN handle,
    uint8_t port,
    uint8_t parity
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the UART port to be configured.

parity

The parity value parameter. You can use predefined constants to set required value.

- 0 or DLN_UART_PARITY_EVEN
- 1 or DLN_UART_PARITY_ODD
- 2 or DLN_UART_PARITY_SPACE
- 3 or DLN_UART_PARITY_MARK
- 4 or NONE

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and parity has been set.

DlnUartGetParity() Function

The **`DlnUartGetParity()`** function retrieves the current parity bit value.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartGetParity(
    HDLN handle,
    uint8_t port,
    uint8_t *parity
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the UART.

parity

The pointer to uint8_t type variable. Variable will be filled with current parity value after successful function execution.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and parity value has been retrieved.

DlnUartSetStopbits() Function

The **`DlnUartSetStopbits()`** function selects the number of stop bits to be sent with each character.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartSetStopbits(
    HDLN handle,
    uint8_t port,
    uint8_t stopbits
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the UART port to be configured.

stopbits

The stop bit value parameter. You can use predefined constants to set required value.

- 0 or DLN_UART_STOPBITS_1
- 1 or DLN_UART_STOPBITS_1_5
- 2 or DLN_UART_STOPBITS_2

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and stop bit value has been set.

DlnUartGetStopbits() Function

The **`DlnUartGetStopbits()`** function retrieves the current number of stop bits to be sent with each character.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartGetStopbits(
    HDLN handle,
    uint8_t port,
    uint8_t *stopbits
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the UART port to be configured.

stopbits

The pointer to uint8_t type variable. Variable will be filled with current stop bit value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and stop bit value has been retrieved.

DlnUartWrite() Function

The **`DlnUartWrite()`** function sends data via UART interface.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartWrite(
    HDLN handle,
    uint8_t port,
    uint16_t size,
    uint8_t *buffer
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the UART port.

size

The size of the message buffer. This parameter is specified in bytes. The maximum value is **`DLN_UART_MAX_TRANSFER_SIZE`** (256 bytes).

buffer

A pointer to an array of unsigned 8-bit integers. This array must be filled with data to be sent during the function execution.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and data has been transmitted.

DlnUartRead() Function

The **`DlnUartRead()`** function receives data via UART interface.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartRead(
    HDLN handle,
    uint8_t port,
    uint16_t size,
    uint8_t *buffer,
    uint16_t *actualSize
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the UART interface port.

size

The size of the message buffer. This parameter is specified in bytes. The maximum value is **`DLN_UART_MAX_TRANSFER_SIZE`** (256 bytes).

buffer

A pointer to an array of unsigned 8-bit integers. This array will be filled with received data during the function execution.

actualSize

A pointer to an array of unsigned 16-bit integers. This variable will be filled with actual data size value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and data has been read.

DlnUartEnableEvent() Function

The **`DlnUartEnableEvent()`** function enables UART event for specified UART port.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartEnableEvent(
    HDLN handle,
    uint8_t port
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the UART port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and events for UART port were enabled.

DlnUartDisableEvent() Function

The **`DlnUartDisableEvent()`** function disables UART events for specified UART port.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartDisableEvent(
    HDLN handle,
    uint8_t port
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the UART port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and events for UART port were disabled.

DlnUartIsEventEnabled() Function

The **`DlnUartIsEventEnabled()`** function informs whether the UART port is currently configured for monitoring events.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartIsEventEnabled(
    HDLN handle,
    uint8_t port,
    uint8_t *enabled
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the UART port.

enabled

A pointer to an unsigned 8-bit integer. The integer will be filled with current port configuration after the function execution: 0 if events are disabled and 1 if events are enabled.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The UART port information has been successfully retrieved.

DlnUartSetEventSize() Function

The **`DlnUartSetEventSize()`** function configures the event size.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartSetEventSize(
    HDLN handle,
    uint8_t port,
    uint16_t size
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the UART port.

size

Event size value.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and event size has been set.

DlnUartGetEventSize() Function

The **`DlnUartGetEventSize()`** function retrieves the current event size value.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartGetEventSize(
    HDLN handle,
    uint8_t port,
    uint16_t *size
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the UART port.

size

The pointer to uint16_t type variable. Variable will be filled with the current event size value after function execution.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and event size has been retrieved.

DlnUartSetEventPeriod() Function

The DlnUartSetEventPeriod() function configures the event period for specified UART port.

This function is defined in the dln_uart.h file.

Syntax

```
DLN_RESULT DlnUartSetEventPeriod(
    HDLN handle,
    uint8_t port,
    uint32_t period
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the UART port to be configured.

period

Defines the event period for event in ms.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and event period has been set.

DlnUartGetEventPeriod() Function

The DlnUartGetEventPeriod() function retrieves the current event period for specified UART port.

This function is defined in the dln_uart.h file.

Syntax

```
DLN_RESULT DlnUartGetEventPeriod(
    HDLN handle,
    uint8_t port,
    uint32_t *period
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the UART port.

period

A pointer to an unsigned 32-bit integer. The integer will be filled with current event repeat interval after the function execution.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and the current event period has been retrieved.

DlnUartGetMinBaudrate() Function

The **`DlnUartGetMinBaudrate()`** function retrieves the minimum possible baud rate value.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartGetMinBaudrate(
    HDLN handle,
    uint8_t port,
    uint32_t *minBaudrate
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the UART port.

minBaudrate

The pointer to uint32_t type variable. Variable will contain minimum possible baud rate value after function execution.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and minimum baud rate value has been retrieved.

DlnUartGetMaxBaudrate() Function

The **`DlnUartGetMaxBaudrate()`** function retrieves the maximum possible baud rate value.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartGetMaxBaudrate(
    HDLN handle,
    uint8_t port,
    uint32_t *maxBaudrate
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the UART port.

maxBaudrate

The pointer to uint32_t type variable. Variable will contain maximum possible baud rate value after function execution.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and maximum baud rate value has been retrieved.

DlnUartGetSupportedCharacterLengths() Function

The **`DlnUartGetSupportedCharacterLengths()`** function returns all supported character length types.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartGetSupportedCharacterLengths(
    HDLN handle,
    uint8_t port,
    DLN_UART_CHARACTER_LENGTHS *supportedLengths
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

UART port number.

supportedLengths

The pointer to **`DLN_UART_CHARACTER_LENGTHS`** structure which will be filled by supported character length values.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

DlnUartGetSupportedParities() Function

The **`DlnUartGetSupportedParities()`** function returns all supported parities values.

This function is defined in the *dln_uart.h* file.

Syntax

```
DLN_RESULT DlnUartGetSupportedParities(
    HDLN handle,
    uint8_t port,
    DLN_UART_PARITIES *supportedParities
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

UART port number.

supportedParities

The pointer to **`DLN_UART_PARITIES`** structure which will be filled by supported parities values.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

DlnUartGetSupportedStopbits() Function

The **`DlnUartGetSupportedStopbits()`** function returns all supported stop bits values.

This function is defined in the **`dln_uart.h`** file.

Syntax

```
DLN_RESULT DlnUartGetSupportedStopbits(
    HDLN handle,
    uint8_t port,
    DLN_UART_STOPBITS *supportedStopbits
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

UART port number.

supportedStopbits

The pointer to **`DLN_UART_STOPBITS`** structure which will be filled by supported stop bits values.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

11. Analog to Digital Converter (ADC) Interface

Some USB-GPIO Interface Adapters have several analog inputs connected to an Analog to Digital Converter (ADC). Analog to digital conversion has many applications. You can acquire data from various analog sensors and save it to your PC for further processing. You can also implement a real time analog data processing in your software. Combined with other USB-GPIO adapter modules you can implement feedback control over your hardware. Analog to digital conversion is well utilized for external analog signal reading such as current, voltage, temperature, distance, pressure, or even color information.

Analog to Digital Converter inputs can generate events. If you adjust low or high threshold value, the adapter will send events once this threshold is crossed.

DLN-4M and DLN-4S each have 2 ADC modules. The first one has 4 10-bit channels and the second one has 4 12-bit channels. DLN-4M or DLN-4S USB-GPIO adapter can be used to measure voltage from 0 V to VDD (positive supply voltage). Since VDD can be configured as 3.3 V, the ADC can be used to measure voltage from 0 V to 3.3 V. You can use either VDD or external supply as the reference voltage.

11.1 Simple ADC Module Example

This example shows how to enable ADC module, setup its resolution and measure voltage from the ADC channel. You can find the complete example in the “*..\Program Files\Diolan\DLN\examples\c_cpp\examples\simple*” folder after DLN setup package installation.

```
#include "..\..\..\common\dln_generic.h"
#include "..\..\..\common\dln_adc.h"
#pragma comment(lib, "..\\..\\..\\bin\\dln.lib")

int _tmain(int argc, _TCHAR* argv[])
{
// Open device
HDLN device;
DlnOpenUsbDevice(&device);

// Set ADC resolution
DlnAdcSetResolution(device, 0, DLN_ADC_RESOLUTION_10BIT);
// Enable ADC channel 0
DlnAdcChannelEnable(device, 0, 0);
// Enable ADC port 0
uint16_t conflict;
DlnAdcEnable(device, 0, &conflict);

// Read ADC value
uint16_t value;
DlnAdcGetValue(device, 0, 0, &value);
printf("ADC value = %d\n", value);

24  // Disable ADC
25  DlnAdcDisable(device, 0);
26  DlnAdcChannelDisable(device, 0, 0);
27  // Close device
28  DlnCloseHandle(device);
29  return 0;
30  }
```

Line 1: `#include "..\..\..\common\dln_generic.h"`

The *dln_generic..h* header file declares functions and data structures for the generic interface. In current example this header is used to call DlnOpenUsbDevice() and **`DlnCloseHandle()`** functions.

Line 2: `#include "..\..\..\common\dln_adc.h"`

The dln_uart.h header file declares functions and data structure specific for ADC interface. By including this header file you are able to call **`DlnAdcSetResolution()`**, **`DlnAdcChannelEnable()`**, **`DlnAdcEnable()`**, **`DlnAdcGetValue()`**, **`DlnAdcDisable()`**, **`DlnAdcChannelDisable()`**.

Line 3: `#pragma comment(lib, "..\\..\\..\\bin\\dln.lib")`

Use *dln.lib* library while project linking.

Line 9: `DlnOpenUsbDevice(&device);`

The function establishes the connection with the DLN adapter. This application uses the USB connectivity of the adapter. For additional options, refer to the Device Opening & Identification section.

Line 12: `DlnAdcSetResolution(device, 0, DLN_ADC_RESOLUTION_10BIT);`

This functions sets ADC resolution to 10 bit.

Line 14: `DlnAdcChannelEnable(device, 0, 0);`

This function enable ADC channel.

Line 17: `DlnAdcEnable(device, 0, &conflict);`

This function enables ADC port 0.

Line 21: `DlnAdcGetValue(device, 0, 0, &value);`

This function gets ADC value of port 0 channel 0.

Line 22: `printf("ADC value = %d\n", value);`

Print retrieved ADC value to console.

Line 25: `DlnAdcDisable(device, 0);`

This function disables ADC port 0.

Line 26: `DlnAdcChannelDisable(device, 0, 0);`

This function disables ADC channel 0 of port 0.

Line 28: `DlnCloseHandle(device);`

The application closes handle to the DLN adapter.

11.2 ADC Functions

This section describes the ADC functions. They are used to control and monitor the ADC module of a DLN-series adapter.

Actual control of the device is performed by use of commands and responses. Each function utilizes respective commands and responses. You can send such commands directly if necessary.

DlnAdcEnable() Function

The **`DlnAdcEnable()`** function activates the corresponding ADC port of your DLN-series adapter. This function is defined in the *dln_adc.h* file.

Syntax

```
DLN_RESULT DlnAdcEnable(
    HDLN handle,
    uint8_t port,
    uint16_t* conflict
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the ADC port to be activated.

conflict

A pointer to an unsigned 16-bit integer. The integer will be filled with the number of the conflicted pin, if any.

A conflict arises if a pin is already assigned to another module of the DLN-series adapter and cannot be used for the ADC module. To fix this a user has to disconnect a pin from a module that it has been assigned to and call the **`DlnAdcEnable()`** function once again. In case there still are conflicted pins, only the number of the next one will be returned.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The operation completed successfully and I2C master port was successfully disabled.

DlnAdcDisable() Function

The **`DlnAdcDisable()`** function deactivates the specified ADC port of your DLN-series adapter. This function is defined in the *dln_adc.h* file.

Syntax

```
DLN_RESULT DlnAdcDisable(
    HDLN handle,
    uint8_t port
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the ADC port to be deactivated.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

DlnAdcIsEnabled() Function

The **`DlnAdcIsEnabled()`** function retrieve information, whether the specified ADC port is activated. This function is defined in the *dln_adc.h* file.

Syntax

```
DLN_RESULT DlnAdcIsEnabled(
    HDLN handle,
    uint8_t port,
    uint8_t* enabled
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the ADC port to retrieve the information from.

enabled

A pointer to an unsigned 8-bit integer. The integer will be filled with the information whether the specified ADC port is activated. There are two possible values:

- 0 or DLN_ADC_DISABLED - ADC port is deactivated.
- 1 or DLN_ADC_ENABLED - ADC port is activated.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

DlnAdcChannelEnable() Function

The **`DlnAdcChannelEnable()`** function activates the specified channel from the corresponding ADC port of your DLN-series adapter. This function is defined in the *dln_adc.h* file.

Syntax

```
DLN_RESULT DlnAdcChannelEnable(
    HDLN handle,
    uint8_t port,
    uint8_t channel
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the ADC port to be used.

channel

A number of the channel to be enabled.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

DlnAdcChannelDisable() Function

The **`DlnAdcChannelEnable()`** function deactivates the specified channel from the corresponding ADC port of your DLN-series adapter.

This function is defined in the *dln_adc.h* file.

Syntax

```
DLN_RESULT DlnAdcChannelDisable(
    HDLN handle,
    uint8_t port,
    uint8_t channel
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the ADC port to be used.

channel

A number of the channel to be disabled.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

DlnAdcChannelIsEnabled() Function

The **`DlnAdcChannelIsEnabled()`** retrieves information, whether the specified ADC channel is activated. This function is defined in the *dln_adc.h* file.

Syntax

```
DLN_RESULT DlnAdcChannelIsEnabled(
   HDLN handle,
   uint8_t port,
   uint8_t channel,
   uint8_t* enabled
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the ADC port to retrieve the information from.

channel

A number of the ADC channel to retrieve the information from.

enabled

A pointer to an unsigned 8-bit integer. The integer will be filled with the information whether the specified ADC channel is activated. There are two possible values:

- 0 or DLN_ADC_DISABLED - ADC channel is deactivated.
- 1 or DLN_ADC_ENABLED - ADC channel is activated.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

DlnAdcGetPortCount() Function

The **`DlnAdcGetPortCount()`** function retrieves the number of ADC ports available in your DLN-series adapter. This function is defined in the *dln_adc.h* file.

Syntax

```
DLN_RESULT DlnAdcGetPortCount(
    HDLN handle,
    uint8_t* count
);
```

Parameters

handle

A handle to the DLN-series adapter.

count

A pointer to an unsigned 8-bit integer. The integer will be filled with the number of available ADC ports after the function execution.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

DlnAdcGetChannelCount() Function

The **`DlnAdcGetChannelCount()`** function retrieves the number of ADC channels, available in the specified ADC-port of your DLN-series adapter.

This function is defined in the *dln_adc.h* file.

Syntax

```
DLN_RESULT DlnAdcGetChannelCount(
    HDLN handle,
    uint8_t port,
    uint8_t* count
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

An ADC port to retrieve the number of channels from.

count

A pointer to an unsigned 8-bit integer. The integer will be filled with the available number of channels in the specified ADC port of the DLN-series adapter.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

DlnAdcChannelSetCfg() Function

The **`DlnAdcChannelSetCfg()`** function changes the configuration of a single GPIO pin and set the conditions of DLN_ADC_CONDITION_MET_EV event generation. This function is defined in the *dln_adc.h* file.

Syntax

```
DLN_RESULT DlnAdcChannelSetCfg(
   HDLN handle,
   uint8_t port,
   uint8_t channel,
   uint8_t eventType,
   uint16_t eventPeriod,
   uint16_t thresholdLow,
   uint16_t thresholdHigh
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the ADC port to be configured.

channel

A number of the ADC channel to be configured.

eventType

Must contain the event generation condition for the ADC channel. The following values are available:

Value	Description
0 or DLN_ADC_EVENT_NONE	No events are generated for the current channel.
1 or DLN_ADC_EVENT_BELOW	Events are generated when voltage level on the ADC channel crosses the lower threshold.
2 or DLN_ADC_EVENT_LEVEL_ABOVE	Events are generated when voltage level on the ADC channel crosses the higher threshold.
3 or DLN_ADC_EVENT_OUTSIDE	Events are generated when voltage level on the ADC channel falls outside of the specified range between thresholds.
4 or DLN_ADC_EVENT_INSIDE	Events are generated when voltage level on the ADC channel falls within the specified range between thresholds.
5 or DLN_ADC_EVENT_ALWAYS	Events are sent periodically with predefined repeat interval. The non-zero interval must be specified for this event type.

eventPeriod

Must contain the repeat interval for DLN_ADC_CONDITION_MET_EV event generation on the pin. The repeat interval is set in ms (1 to 65,535ms). If the repeat interval is set to 0, the DLN-series adapter will send a single event when the level on the line changes to meet the specified conditions.

thresholdLow

The lower threshold value, specified in bits.

thresholdHigh

The higher threshold value specified in bits.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

DlnAdcChannelGetCfg() Function

The **`DlnAdcChannelGetCfg()`** function retrieves the current configuration settings of a single ADC channel.

This function is defined in the *dln_adc.h* file.

Syntax

```
DLN_RESULT DlnAdcChannelGetCfg(
    HDLN handle,
    uint8_t port,
    uint8_t channel,
    uint8_t* eventType,
    uint16_t* eventPeriod,
    uint16_t* thresholdLow,
    uint16_t* thresholdHigh
 );
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the ADC port to retrieve the information from.

channel

A number of the ADC channel to retrieve the information from.

eventType

A pointer to an unsigned 8-bit integer. The integer will be filled the currently set event generation condition for the ADC channel after the function execution. The following values are available:

Value	Description
0 or DLN_ADC_EVENT_NONE	No events are generated for the current channel.
1 or DLN_ADC_EVENT_BELOW	Events are generated when voltage level on the ADC channel crosses the lower threshold.
2 or DLN_ADC_EVENT_LEVEL_ABOVE	Events are generated when voltage level on the ADC channel crosses the higher threshold.
3 or DLN_ADC_EVENT_OUTSIDE	Events are generated when voltage level on the ADC channel falls outside of the specified range between thresholds.
4 or DLN_ADC_EVENT_INSIDE	Events are generated when voltage level on the ADC channel falls within the specified range between thresholds.
5 or DLN_ADC_EVENT_ALWAYS	Events are sent periodically with predefined repeat interval. The non-zero interval must be specified for this event type.

eventPeriod

A pointer to an unsigned 16-bit integer. The integer will be filled with the repeat interval for `DLN_ADC_CONDITION_MET_EV` event generation on the pin after the function execution. The repeat interval is set in ms (1 to 65,535ms). If the repeat interval is set to 0, the DLN-series adapter will send a single event when the level on the line changes to meet the specified conditions.

thresholdLow

A pointer to an unsigned 16-bit integer. The integer will be filled with the lower voltage threshold value, specified in bits, after the function execution.

thresholdHigh

A pointer to an unsigned 16-bit integer. The integer will be filled with the higher voltage threshold value, specified in bits, after the function execution.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

DlnAdcSetResolution() Function

The **`DlnAdcSetResolution()`** function sets the ADC resolution value of your DLN-series adapter. This function is defined in the *dln_adc.h* file.

Syntax

```
DLN_RESULT DlnAdcSetResolution(
    HDLN handle,
    uint8_t port,
    uint8_t resolution
 );
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the ADC port to be configured.

resolution

The new resolution in bits. For the list of compatible values see table below.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

DlnAdcGetResolution() Function

The **`DlnAdcGetResolution()`** function retrieves the currently set ADC resolution of your DLN-series adapter.

This function is defined in the *dln_adc.h* file.

Syntax

```
 DLN_RESULT DlnAdcGetResolution(
   HDLN handle,
   uint8_t port,
   uint8_t* resolution
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the ADC port to retrieve the information from.

resolution

The current ADC resolution in bits.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

DlnAdcGetSupportedResolutions() Function

The **`DlnAdcGetSupportedResolutions()`** function returns all supported resolution values for connected DLN-series adapter.

This function is defined in the *dln_adc.h* file.

Syntax

```
DLN_RESULT DlnAdcGetSupportedResolutions(
    HDLN handle,
    uint8_t port,
    DLN_ADC_RESOLUTIONS *supportedResolutions
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the ADC port to be used.

supportedResolutions

Pointer to `DLN_ADC_RESOLUTIONS` structure, which will be filled by supported resolution values.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

DlnAdcGetValue() Function

The **`DlnAdcGetValue()`** function retrieves current voltage on the specified ADC channel of your DLN-series adapter.

This function is defined in the *dln_adc.h* file.

Syntax

```
DLN_RESULT DlnAdcGetValue(
    HDLN handle,
    uint8_t port,
    uint8_t channel,
    uint16_t* value
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the ADC port to retrieve the information from.

channel

A number of the ADC channel to retrieve the information from.

value

A pointer to an unsigned 8-bit integer. The integer will be filled with the voltage level on the specified ADC channel.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

DlnAdcGetAllValues() Function

The **`DlnAdcGetAllValues()`** function retrieves current voltage values from all enabled ADC channels of your DLN-series adapter.

This function is defined in the *dln_adc.h* file.

Syntax

```
DLN_RESULT DlnAdcGetAllValues(
   HDLN handle,
   uint8_t port,
   uint16_t* channelMask,
   uint16_t* values
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the ADC port to retrieve the information from.

channelMask

A pointer to an unsigned 16-bit integer. This integer will be filled with a bit mask, each of the bits corresponding to an ADC channel of the port. This parameter contains currently enabled ADC channels of your DLN-series device.

values

A pointer to an unsigned 16-bit integer. This integer will be filled with a bit mask, each of the bits corresponding to an ADC value of the ADC channel of the port.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

DlnAdcGetSupportedEventTypes() Function

The **`DlnAdcGetSupportedEventTypes()`** function returns all supported ADC event types for opened DLN-series adapter.

This function is defined in the *dln_adc.h* file.

Syntax

```
DLN_RESULT DlnAdcGetSupportedEventTypes(
    HDLN handle,
    uint8_t port,
    DLN_ADC_EVENT_TYPES *supportedEventTypes
);
```

Parameters

handle

A handle to the DLN-series adapter.

port

A number of the ADC port to be used.

supportedEventTypes

Pointer to **`DLN_ADC_EVENT_TYPES`** structure, which will be filled by supported event type values.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

12. Bootloader Interface

Bootloader interface is supported by every DLN-series adapter. It is used to update firmware in order to apply new features, updates and fixes.

Since DLN-series adapters are operated via the Application Programming Interface (API), a user can also update firmware for remote devices.

Attention: Firmware for DLN-series adapters is supplied encoded. Each DLN-series adapter is only compatible with specifically designed firmware. Do not try to install an incompatible firmware.

12.1 Bootloader Device Modes

There are three bootloader modes of a DLN-series adapter:

Mode	Description
Bootloader	This mode is activated in order to use bootloader functions. Only generic and bootloader functions are available in this mode. The Bootloader mode can be accesses either by sending the `DLN_BOOT_ENTER_BOOTLOADER` command or by setting the respective jumper on the DLN-series adapter board. The jumper must be set before the device is powered up. In case the device is not fitted with external power source, the jumper must be set before connecting the device to a USB port.
Application	When in this mode, the DLN-series adapter functions normally. However, some bootloader functions (**`DlnBootReadFlash()`** and **`DlnBootWriteFlash()`**) become unavailable.
Update in progress	This mode is activated automatically once the device receives the `DLN_BOOT_WRITE_FLASH` command. In case of a power failure, the DLN-series adapter will be rebooted and remain in this mode.

12.2 Bootloader Functions

This section describes the Bootloader functions. They are used to control and monitor the Bootloader module of a DLN-series adapter.

Actual control of the device is performed by use of commands and responses. Each function utilizes respective commands and responses. You can send such commands directly if necessary.

DlnBootEnterBootloader() Function

The **`DlnBootEnterBootloader()`** function enters the Bootloader mode.

This function is defined in the *dln_boot.h* file.

Syntax

```
DLN_RESULT DlnBootEnterBootloader(
   HDLN handle
);
```

Parameters

handle

A handle to the DLN-series adapter.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

DlnBootExitBootloader() Function

The **`DlnExitBootloader()`** function exits the Bootloader mode and continues in the Application mode.

This function is defined in the *dln_boot.h* file.

Syntax

```
DLN_RESULT DlnBootExitBootloader(
    HDLN handle
);
```

Parameters

handle

A handle to the DLN-series adapter.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

DlnBootGetMode() Function

The **`DlnBootGetMode()`** function retrieves current mode of your DLN-series adapter.

This function is defined in the *dln_boot.h* file.

Syntax

```
DLN_RESULT DlnBootGetMode(
    HDLN handle,
    DLN_BOOT_MODE* mode
 );
```

Parameters

handle

A handle to the DLN-series adapter.

mode

A pointer to the 32-bit `DLN_BOOT_MODE` variable. This variable will be used to store current device mode after the function execution.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

DlnBootGetMemDesc() Function

The **`DlnBootGetMemDesc()`** function retrieves information about the device internal flash memory.

This function is defined in the *dln_adc.h* file.

Syntax

```
DLN_RESULT DlnBootGetMemDesc(
    HDLN handle,
    uint32_t memory,
    DLN_BOOT_MEM_DESC *descriptor
);
```

Parameters

handle

A handle to the DLN-series adapter.

memory

Memory type value. Available only **`DLN_BOOT_MEM_FLASH`** value.

descriptor

A pointer to the **`DLN_BOOT_MEM_DESC`** structure, which will be filled with data after the function execution.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

Function was successfully executed.

DlnBootGetMemCount() Function

The **`DlnBootGetMemCount()`** function retrieves the number of flash panes of your DLN-series adapter.

This function is defined in the *dln_boot.h* file.

Syntax

```
DLN_RESULT DlnBootGetMemCount(
    HDLN handle,
    uint32_t *count
);
```

Parameters

handle

A handle to the DLN device.

count

A pointer to 8-bit unsigned integer, which will be filled with the number of flash panes after the function execution.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The number of flash panes was successfully retrieved.

DlnBootRead() Function

The **`DlnBootRead()`** function reads data on the DLN-series adapter current firmware.

This function is defined in the *dln_boot.h* file.

Attention: This function is only available, when the device operates in Bootloader mode.

Syntax

```
DLN_RESULT DlnBootReadFlash(
    HDLN handle,
    uint32_t memory,
    uint32_t address,
    uint16_t size,
    uint8_t* buffer
 );
```

Parameters

handle

A handle to the DLN-series adapter.

memory

Memory type value. Available only **`DLN_BOOT_MEM_FLASH`** value.

address

Address to start reading from.

size

Size of the data to read.

buffer

A pointer to an unsigned 8 bit integer. This integer will be filled with the firmware data after the function execution.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The number of flash panes was successfully retrieved.

DlnBootWrite() Function

The **`DlnBootWrite()`** function writes new firmware into the device internal memory. This function is only available, when the device functions in the Bootloader mode.

This function is defined in the *dln_boot.h* file.

Syntax

```
DLN_RESULT DlnBootWrite(
    HDLN handle,
    uint32_t memory,
    uint32_t address,
    uint16_t size,
    uint8_t* buffer
 );
```

Parameters

handle

A handle to the DLN-series adapter.

memory

Memory type value. Available only **`DLN_BOOT_MEM_FLASH`** value.

address

The address to start writing from.

size

The size of the firmware.

buffer

Buffer with the firmware data.

Return value

Possible return codes include, but are not limited to, the following:

DLN_RES_SUCCESS

The number of flash panes was successfully retrieved.

13. C/C++ Examples

We offer dozens of free and open source software applications to be used with DLN-series adapters. Most of the software applications are designed to be used with all DLN-series adapters. Others can be only used with specific adapters, for example, SPI slave and I2C slave interfaces are available in DLN-4S adapter only.

New software is added frequently so check back often. Do you need additional functionality? Feel free to contact us at support@diolan.com. We review all requests for software customizing. If the requested software can be helpful to a wide range of customers, we will make this software free of charge. Otherwise we will offer the software customizing at an affordable price.

Return Code Reference

DLN_RES_COMMAND_NOT_SUPPORTED (0x91)

This function is not supported be the specified DLN adapter. There are 2 possible reasons for this code:
1) The functionality related to the called function is not supported by this DLN-series adapter. If you need this functionality, you can order another DLN-series adapter.
2) The firmware version in your adapter is old. To enable this functionality, you need to update the device firmware.

DLN_RES_CONNECTION_LOST (0xA0)

The connection to the DLN adapter or DLN server was interrupted.

DLN_RES_INVALID_EVENT_PERIOD (0xAC)

Defining the event configuration, you specified the invalid event period. It may occur, for example, if you set the zero event period for the ALWAYS event type. For details, read Digital Input Events.

DLN_RES_INVALID_EVENT_TYPE (0xA9)

You specified the invalid event type. For details, read Digital Input Events.

DLN_RES_INVALID_HANDLE (0x8F)

The specified handle is not valid.

DLN_RES_INVALID_PIN_NUMBER

The specified pin number is not valid. Use the `DlnGpioGetPinCount()` function to get the available number of pins for your DLN-series adapter.

DLN_RES_NON_ZERO_RESERVED_BIT (0xAD)

One or more of the reserved bits in the DLN_GPIO_PIN_CONFIG structure are set to 1. These bits are reserved for future and must be set to zero. If the DLN-series adapter founds that any of these bits is set to 1, it returns the error code.

DLN_RES_PIN_IN_USE (0xA5)

The pin is assigned to another module of the adapter and cannot be assigned to the GPIO module. Use the `DlnGetPinCfg()` function to get the name of the module which the pin is assigned to.

DLN_RES_PIN_NOT_CONNECTED_TO_MODULE (0xAA)

The pin is not assigned to the GPIO module. Use the `DlnGpioPinEnable()` function to assign the pin to the GPIO module. Use the `DlnGpioPinIsEnabled()` function to check whether or not the pin is assigned to the GPIO module. Use the `DlnGetPinCfg()` function to get the name of the module which the pin is assigned to.

DLN_RES_SUCCESS (0x00)

The function was executed successfully.