USB LISM-PI26xx Interface

Interface Manual
USBLISMPI2632.DLL
USBLISMPI2664.DLL

(Rev 1.0)

COPTONIX

Falkentaler Steig 9
D – 13467 Berlin
Phone: +49 30 617 412 48
Fax: +49 30 617 412 47
www.coptonix.com

Contents

Function Reference: USB LISM-PI26xx API....5
Scope of the API....5
Important Notes....5
USB Device Management....6
GetErrorString....7
Initialize....8
EnumDevices....9
OpenDeviceByIndex....10
OpenDeviceBySerial....11
CloseDevice....13
DeviceCount....14
CurrentDeviceIndex....15
GetFWVersion....16
GetVendorName....17
GetProductName....18
GetSerialNumber....20
SetEpTimeout....21
GetEpTimeout....22
GetPacketLength....23
SetPacketLength....24
Data Transfer....26
SetCFG1....27
GetCFG1....28
WaitForPipe....30
GetPipe....31
GetFPS....33
I²C Bus Interface....34
SetSlaveAddress....34
GetSlaveAddress....36
SaveSlaveAddress....37
SetMuxChannel....38
GetMuxChannel....39
ReadI2C_Ext....41
WriteI2C_Ext....43
ReadI2C....44
WriteI2C....46
I2CRead1Byte....47
I2CRead2Bytes....48
I2CRead4Bytes....50
I2CWrite1Byte....51
I2CWrite2Bytes....52
I2CWrite4Bytes....54
I2CWriteCmd....55
Clock and Timing Generator (CTG) Register Access....57
HW_Reset....58

Resume....59
Suspend....60
UpdateParam....61
SetState....62
SetModeConfig....64
SetIFConfig....65
SetStHigh....67
SetStLow....68
SetStPulse....70
SetLinesPerFrame....71
SetQuadCount....73
SetSoftTriggerTime....74
SetTriggerDelay....75
SetTriggerWidth....77
SetPixelCount....78
SetEdgeDelay....79
SetADCVref....81
SetADCGain....82
SetADCOffset....84
SetADCBias....85
GetCTGState....87
GetState....88
GetModeConfig....89
GetIFConfig....91
GetStHigh....92
GetStLow....94
GetStPulse....95
GetLinesPerFrame....96
GetQuadCount....98
GetSoftTriggerTime....99
GetTriggerDelay....100
GetTriggerWidth....101
GetPixelCount....103
GetEdgeDelay....104
GetADCVref....105
GetADCGain....107
GetADCOffset....108
GetADCBias....110
SaveSettings....111
ReloadSettings....112
ResetSettings....114
GetInitStatus....115
GetComResult....116
Version and Hardware Identification....118
GetMCUVersion....119
GetCTGVersion....120
GetHW1Id....121
GetHW1Version....122

GetHW2Version.....123
Revision history.....125

Function Reference: USB LISM-PI26xx API

This document provides a complete and structured reference for all available functions in the USB LISM-PI26xx software interface. It covers every command that can be executed from the host side to communicate with, configure, and control the LISM-PI26xx line image sensor module via USB and I²C.

Each function is described with its Pascal and C/C++ declarations, purpose, input and output parameters, return values, usage notes, and typical application scenarios. The goal is to serve both as a programming manual and as a technical reference for integrating the LISM-PI26xx into measurement, inspection, and automation systems.

Scope of the API

The documented functions cover the following areas:

- **I²C register access** (1-, 2-, and 4-byte read/write operations)
- **Acquisition control** (start, stop, suspend, resume)
- **Timing configuration** (START pulse, soft trigger, trigger output)
- **Trigger mode selection** (external, software, quadrature)
- **Interface behavior** (clock polarity, signal levels, data width)
- **ADC and analog frontend settings** (gain, offset, voltage range)
- **EEPROM management** (save, reload, reset of configuration)
- **Hardware and firmware identification**
- **Status and diagnostic functions** (initialization state, last command result)

Important Notes

- All I²C commands automatically switch to **multiplexer channel 0**, providing direct access to the internal LISM-PI26xx registers.
- Unless otherwise stated, all timing values are expressed in **microseconds** or **pixel clock cycles**, depending on the context.
- Functions that alter EEPROM content may require a **power-cycle** to activate changes (e.g., I²C address reassignment).
- Each function is stateless; the host is responsible for preserving or reapplying settings as needed between sessions.

USB Device Management

This section describes the foundational USB interface functions required to initialize, detect, and manage communication with LISM-PI26xx modules. These functions represent the entry point for any host application intending to interact with the sensor hardware via USB. Before any image acquisition or configuration can take place, the system must establish a reliable communication path — a process handled entirely through the USB device management interface.

The initialization routine prepares internal memory buffers and background threads that form the data pipeline between the device and the host system. Once initialized, the interface supports device discovery through enumeration of all connected LISM-PI26xx modules. Devices can then be uniquely identified and selected using either their index in the enumeration list or their factory-assigned serial number. This makes it possible to consistently target a specific hardware module, even in multi-device environments or automated production setups.

After a device is opened, the interface provides functions to retrieve hardware descriptors, such as vendor name, product name, firmware version, and serial number. These identifiers can be used to display device status, filter supported hardware models, or log system configuration for traceability and debugging purposes.

USB configuration parameters — including endpoint timeouts and transfer packet lengths — are also handled within this function group. These parameters directly affect how the host software and the USB stack interact, and must be tuned to match the sensor resolution, transfer mode, and expected data throughput.

All functions in this section are designed to operate independently of the I²C subsystem or image sensor logic. They do not modify any acquisition or timing settings, but instead focus on establishing a stable and efficient communication channel. This separation ensures a clean architectural division between transport-layer handling and sensor-level configuration.

Whether used in desktop software, embedded systems, or automated test benches, the USB Device Management functions provide the critical low-level infrastructure that enables all higher-level operations on the LISM-PI26xx platform.

GetErrorString

Pascal Declaration:

```
function ls_geterrorstring(dwErr: DWORD): PAnsiChar; stdcall;
```

C/C++ Declaration:

```
const char* __stdcall ls_geterrorstring(UInt32 dwErr);
```

Description:
The `ls_geterrorstring` function converts a numerical error code into a human-readable, null-terminated ANSI string. It is useful for debugging and logging, especially when other DLL functions return non-zero values indicating an error.

Most functions in `usblismpi2632.dll` (32-bit) or `usblismpi2664.dll` (64-bit) return a DWORD:

- 0 indicates success.
- Non-zero indicates failure.

Passing such a value to `ls_geterrorstring` returns a textual description of the error.

Parameters:

- `dwErr`: The error code to convert.
  If `dwErr = 0`, the function returns the string `"SUCCESS"`.

Return Value:

- Returns a pointer to a null-terminated ASCII string (`const char*`) describing the error.

Usage Example:

```
res := ls_setpacketlength(4096);
if res <> 0 then
  ShowMessage(String(ls_geterrorstring(res)));
```

Remarks:
This function does not require an open device connection and can be used at any time after loading the DLL.

Initialize

Pascal Declaration:

```
function ls_initialize(dwPipeSize, dwPacketLength, dwThreadClass: DWORD;
                       iThreadPrio: Integer;
                       pcMsgID: PAnsiChar): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_initialize(UInt32 dwPipeSize,
                               UInt32 dwPacketLength,
                               UInt32 dwThreadClass,
                               Int32  iThreadPrio,
                               const char* pcMsgID);
```

Description:
The `ls_initialize` function prepares the internal infrastructure of the DLL for data acquisition. It allocates memory buffers (ring buffer) and configures internal parameters such as packet length and thread priority. This function must be called **once** before opening any USB device.

In addition, it registers a unique Windows message identifier using the provided `pcMsgID` string. This message ID can later be used to identify custom Windows messages posted by the DLL, such as **USB device connection or removal notifications**. It is **not** used to signal new image data availability.

> **Note:** The data acquisition thread is **not** started by this function. It is launched only when a device is opened using `ls_opendevicebyindex` or `ls_opendevicebyserial`.

Parameters:

- `dwPipeSize`: Size (in bytes) of the internal ring buffer used to store incoming image data (e.g., `4 * 1024 * 1024` for 4 MB).
- `dwPacketLength`: Number of bytes to be read per USB transaction. Must match or be a multiple of a full frame size.
- `dwThreadClass`: Thread priority class (e.g., `NORMAL_PRIORITY_CLASS`).
- `iThreadPrio`: Thread priority level (e.g., `THREAD_PRIORITY_NORMAL`).
- `pcMsgID`: A unique null-terminated string used to register a custom Windows message via `RegisterWindowMessage`.

Return Value:

- **Success**: Returns a message ID between `0xC000` and `0xFFFF`.
- **Failure**: Returns 0, indicating that initialization failed (e.g., due to allocation error or invalid parameters).

Remarks:

- The returned message ID should be stored and used in the application's Windows message loop.
- This function must be called once before invoking `ls_enumdevices` or any device open function.
- No device communication is performed at this stage; device-specific operations begin after opening a device.

Usage Example (Pascal):

```
var
  msgID: DWORD;
begin
  msgID := ls_initialize(4 * 1024 * 1024, 4096,
                         NORMAL_PRIORITY_CLASS,
                         THREAD_PRIORITY_NORMAL,
                         'LISMAppMessage');
  if msgID = 0 then
    ShowMessage('Initialization failed!');
end;
```

EnumDevices

Pascal Declaration:

```
function ls_enumdevices: Integer; stdcall;
```

C/C++ Declaration:

```
Int32 __stdcall ls_enumdevices(void);
```

Description:
The `ls_enumdevices` function scans all USB ports for connected USB LISM-PI26xx devices and updates the internal device list. It assigns a **zero-based index** to each detected device, which can later be used in calls to functions such as `ls_opendevicebyindex`, `ls_getserialnumber`, or `ls_getfwversion`.

This function should be called:

- Once during application startup, **after** `ls_initialize`,
- And again whenever USB device changes are expected (e.g., a device is plugged in or removed).

Return Value:

- Returns the number of connected devices found (0 or more).

Remarks:

- Each call to `ls_enumdevices` **overwrites the internal device list**.
- The device index values are **only valid until the next call** to this function.
- This function does **not** open or communicate with the devices – it only discovers and lists them.

Usage Example (Pascal):

```
var
  deviceCount: Integer;
begin
  deviceCount := ls_enumdevices;
  if deviceCount = 0 then
    ShowMessage('No devices found.');
end;
```

OpenDeviceByIndex

Pascal Declaration:

```
function ls_opendevicebyindex(index: Integer): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_opendevicebyindex(Int32 index);
```

Description:
The `ls_opendevicebyindex` function opens a connection to a specific USB LISM-PI26xx device, identified by its zero-based index from the internal list created by `ls_enumdevices`.

Once the device is opened:

- Internal communication is initialized,
- A background read thread is launched for continuous data acquisition.

Parameter:

- `index`: The index of the device to open, starting at 0.

Return Value:

- 0: Success – device opened.

- Non-zero: Error code (e.g., invalid index, device busy).

Remarks:

- Call `ls_enumdevices` first to populate or update the device list.
- Only one device can be opened at a time.
- Call `ls_closedevice` to close the connection and stop the read thread when done.
- After a successful call, you can query information about the connected device using:
  - `ls_getserialnumber(index)` – returns the device's serial number.
  - `ls_getfwversion(index)` – returns the firmware version.
  - `ls_getproductname(index)` – returns the device's product name.

Usage Example (Pascal):

```
var
  res: DWORD;
  serial: PAnsiChar;
begin
  if ls_enumdevices > 0 then
  begin
    res := ls_opendevicebyindex(0);
    if res = 0 then
    begin
      serial := ls_getserialnumber(0);
      ShowMessage('Device opened. Serial number: ' + String(serial));
    end
    else
      ShowMessage('Error opening device: ' + String(ls_geterrorstring(res)));
  end
  else
    ShowMessage('No devices found.');
end;
```

OpenDeviceBySerial

Pascal Declaration:

```
function ls_opendevicebyserial(pcserialnum: PAnsiChar): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_opendevicebyserial(const char* pcserialnum);
```

Description:

The `ls_opendevicebyserial` function opens a connection to a USB LISM-PI26xx device using its

serial number. This method is ideal when a specific physical device must be addressed consistently, regardless of how many devices are connected or in what order they were enumerated.

It establishes communication with the selected device and starts the internal read thread for data acquisition.

Parameter:

- `pcserialnum`: Pointer to a null-terminated ANSI string containing the target device's serial number (e.g., `'1600021'`).

Return Value:

- 0: Success – device opened.
- Non-zero: Error code if the serial number is not found, the device is already in use, or an internal error occurs.

Use Cases:

- Ensuring persistent device identification across reboots or reconnects.
- Automatically reconnecting to a known device in environments with multiple connected scanners.
- Production lines or test stations where fixed hardware assignment is essential.

Remarks:

- `ls_enumdevices` must be called before using this function to populate the internal device list.
- Only one device can be opened at a time via the DLL interface.
- Use `ls_closedevice` to close the device and release associated resources when finished.
- Serial numbers can be obtained beforehand using `ls_getserialnumber(index)`.

Usage Example (Pascal):

```
var
  res: DWORD;
begin
  if ls_enumdevices > 0 then
  begin
    res := ls_opendevicebyserial('1600021');
    if res = 0 then
      ShowMessage('Device opened successfully by serial.')
    else
      ShowMessage('Error: ' + String(ls_geterrorstring(res)));
  end
  else
    ShowMessage('No devices found.');
end;
```

CloseDevice

Pascal Declaration:

```
function ls_closedevice: DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_closedevice(void);
```

Description:
The `ls_closedevice` function terminates the connection to the currently opened USB LISM-PI26xx device. It stops the background read thread, deinitializes the USB interface, and releases any allocated resources such as memory buffers and USB endpoints.

This function is required to cleanly shut down communication with the device and prepare the system for device changes or application shutdown.

Return Value:

- 0: Success – device was closed, or no device was open (no error).
- Non-zero: Only returned in rare cases of internal shutdown errors.

Remarks:

- This function should be called:
    - Before exiting the application,
    - Before opening another device,
    - Before changing configuration settings such as USB packet size or timeouts.
- After calling `ls_closedevice`, all device-specific functions (e.g., `ls_getpipe`, `ls_setstate`, etc.) become invalid until a new device is opened.
- It is safe to call `ls_closedevice` multiple times. If no device is open, the function returns `0` and performs no action.

Usage Example (Pascal):

```
var
  res: DWORD;
begin
  res := ls_closedevice;
  if res = 0 then
    ShowMessage('Device closed (or no device was open).')
  else
    ShowMessage('Error during close: ' + String(ls_geterrorstring(res)));
end;
```

DeviceCount

Pascal Declaration:

```
function ls_devicecount: Byte; stdcall;
```

C/C++ Declaration:

```
UInt8 __stdcall ls_devicecount(void);
```

Description:
The `ls_devicecount` function returns the number of USB LISM-PI26xx devices currently listed in the internal device list. This list is created by the most recent call to `ls_enumdevices`.

Unlike `ls_enumdevices`, this function does **not** scan the USB bus or update the device list. It simply returns the count from the previously built list.

Return Value:

- The number of devices in the internal list (range: 0–255).
- If `ls_enumdevices` has not been called yet, the return value will be 0.

Remarks:

- This function can be used after `ls_enumdevices` to retrieve the current number of known devices.
- It does **not** detect real-time USB changes.
- To refresh the internal list, call `ls_enumdevices` again before querying.

Usage Example (Pascal):

```
var
  count: Byte;
begin
  ls_enumdevices;
  count := ls_devicecount;
  ShowMessage('Devices found: ' + IntToStr(count));
end;
```

CurrentDeviceIndex

Pascal Declaration:

```
function ls_currentdeviceindex: Integer; stdcall;
```

C/C++ Declaration:

```
Int32 __stdcall ls_currentdeviceindex(void);
```

Description:
The `ls_currentdeviceindex` function returns the **zero-based index** of the currently opened USB LISM-PI26xx device. This index corresponds to the order in the internal device list created by the most recent call to `ls_enumdevices`.

It allows the application to identify which device is currently active, which is useful for managing multiple devices or for logging/debugging purposes.

Return Value:

- 0, 1, 2, …: Index of the currently opened device.
- -1: No device is currently open.

Use Cases:

- To verify which device is currently active in the application.
- To retrieve additional properties (e.g., serial number, product name) of the active device using functions like `ls_getserialnumber(index)`.
- For diagnostics and user feedback in multi-device environments.

Remarks:

- The index is only valid relative to the internal list from the **last** call to `ls_enumdevices`.
- If `ls_enumdevices` is called **after** the device has been opened, the index may no longer correspond to the same physical device — avoid mixing both unless necessary.
- If no device has been opened, the function returns -1.
- This function is read-only and does not affect device state.

Usage Example (Pascal):

```
var
  idx: Integer;
  serial: PAnsiChar;
begin
  idx := ls_currentdeviceindex;
  if idx >= 0 then
```

```
  begin
    serial := ls_getserialnumber(idx);
    ShowMessage('Device '+IntToStr(idx)+' is open. Serial: '+ String(serial));
  end
  else
    ShowMessage('No device currently open.');
end;
```

GetFWVersion

Pascal Declaration:

```
function ls_getfwversion(index: Integer): WORD; stdcall;
```

C/C++ Declaration:

```
UInt16 __stdcall ls_getfwversion(Int32 index);
```

Description:
The `ls_getfwversion` function returns the firmware version of a connected USB LISM-PI26xx device, identified by its index in the internal device list. This version is retrieved from the device's USB descriptor and requires the device to be opened beforehand.

The version is encoded in a 16-bit word:

- High byte (MSB) = major version
- Low byte (LSB) = minor version

Examples:

- 0x0120 = version 1.32
- 0x0105 = version 1.5

Parameter:

- `index`: The zero-based index of the device (as returned by `ls_enumdevices`) and must refer to a currently opened device.

Return Value:

- Firmware version as a `WORD` (16-bit unsigned)
- Returns 0 if the device is not open, the index is invalid, or the version could not be read

Use Cases:

- Verifying firmware version for compatibility and diagnostics
- Displaying firmware info in UI or logs

- Detecting older devices for version-dependent behavior

Remarks:

- The function reads the version from the USB descriptor, not from internal MCU memory
- You must call `ls_opendevicebyindex` or `ls_opendevicebyserial` before calling this function
- Calling this function on a non-opened device will return 0

Usage Example (Pascal):

```
var
  fw: WORD;
  major, minor: Byte;
begin
  if ls_enumdevices > 0 then
  begin
    if ls_opendevicebyindex(0) = 0 then
    begin
      fw := ls_getfwversion(0);
      major := HiByte(fw);
      minor := LoByte(fw);
      ShowMessage('Firmware version: ' + IntToStr(major) + '.' +
IntToStr(minor));
    end
    else
      ShowMessage('Device could not be opened.');
  end;
end;
```

GetVendorName

Pascal Declaration:

```
function ls_getvendorname(index: Integer): PAnsiChar; stdcall;
```

C/C++ Declaration:

```
const char* __stdcall ls_getvendorname(Int32 index);
```

Description:
The `ls_getvendorname` function returns the vendor name string from the USB descriptor of a connected USB LISM-PI26xx device. The string is returned as a null-terminated ANSI string and typically reflects the manufacturer’s name (e.g., "Coptonix GmbH").

Parameter:

- `index`: The zero-based index of the target device, corresponding to the most recent `ls_enumdevices` list. The device must be currently open.

Return Value:

- A pointer to a null-terminated ANSI string containing the vendor name.

Use Cases:

- Displaying manufacturer information in the user interface
- Filtering or grouping devices by vendor
- Logging and diagnostics

Remarks:

- The device must be opened before calling this function.
- The returned string resides in DLL-internal memory and must **not** be freed or modified.
- If the device is not open, the function returns an empty string or undefined behavior.

Usage Example (Pascal):

```
var
  vendor: PAnsiChar;
begin
  if ls_enumdevices > 0 then
  begin
    if ls_opendevicebyindex(0) = 0 then
    begin
      vendor := ls_getvendorname(0);
      ShowMessage('Vendor: ' + String(vendor));
    end
    else
      ShowMessage('Device open failed.');
  end;
end;
```

GetProductName

Pascal Declaration:

```
function ls_getproductname(index: Integer): PAnsiChar; stdcall;
```

C/C++ Declaration:

```
const char* __stdcall ls_getproductname(Int32 index);
```

Description:
The `ls_getproductname` function returns the product name string from the USB descriptor of a connected USB LISM-PI26xx device. The name identifies the device model or type and is returned as a null-terminated ANSI string, e.g., `"USB LISM-PI26xx"`.

Parameter:

- `index`: The zero-based index of the device, based on the most recent `ls_enumdevices` list. The device must be currently open.

Return Value:

- A pointer to a null-terminated ANSI string containing the product name.

Use Cases:

- Displaying device models in user interfaces
- Differentiating between hardware versions or series
- Logging device identifiers in automated environments

Remarks:

- The device must be opened before calling this function.
- The returned pointer refers to a static string managed by the DLL and must **not** be modified or freed.
- If the device is not open, the result may be empty or undefined.

Usage Example (Pascal):

```
var
  name: PAnsiChar;
begin
  if ls_enumdevices > 0 then
  begin
    if ls_opendevicebyindex(0) = 0 then
    begin
      name := ls_getproductname(0);
      ShowMessage('Product name: ' + String(name));
    end
    else
      ShowMessage('Device open failed.');
  end;
end;
```

GetSerialNumber

Pascal Declaration:

```
function ls_getserialnumber(index: Integer): PAnsiChar; stdcall;
```

C/C++ Declaration:

```
const char* __stdcall ls_getserialnumber(Int32 index);
```

Description:
The `ls_getserialnumber` function returns the unique serial number string from the USB descriptor of a connected USB LISM-PI26xx device. The serial number is assigned during manufacturing and is used to uniquely identify the physical device across different systems and sessions.

Parameter:

- `index`: The zero-based index of the device, based on the most recent `ls_enumdevices` list. The device must be currently open.

Return Value:

- A pointer to a null-terminated ANSI string containing the device's serial number (e.g., `'1600042'`).

Use Cases:

- Persistently identifying and reconnecting to a specific unit
- Logging and traceability in production or automation environments
- Linking saved settings or calibration data to specific hardware

Remarks:

- The device must be opened before calling this function
- The returned string is managed internally by the DLL and must **not** be freed or modified
- Use this value with `ls_opendevicebyserial` to reopen a specific device later

Usage Example (Pascal):

```
var
  serial: PAnsiChar;
begin
  if ls_enumdevices > 0 then
  begin
    if ls_opendevicebyindex(0) = 0 then
    begin
      serial := ls_getserialnumber(0);
      ShowMessage('Serial number: ' + String(serial));
    end
    else
      ShowMessage('Failed to open device.');
  end;
end;
```

SetEpTimeout

Pascal Declaration:

```
function ls_seteptimeout(dwtimeout: DWORD): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_seteptimeout(UInt32 dwtimeout);
```

Description:
The `ls_seteptimeout` function sets the timeout (in milliseconds) for USB IN endpoint transfers. This timeout defines how long the system will wait for incoming data from the USB LISM-PI26xx device before reporting a transfer failure.

This setting affects all USB bulk reads performed by the DLL's internal read thread.

Parameter:

- `dwtimeout`: Timeout duration in milliseconds. For example, 1000 means 1 second.

Return Value:

- 0: Success — timeout value set
- Non-zero: Error code if the timeout could not be applied (e.g., device is currently open)

Use Cases:

- Adjusting USB latency for high-speed free-running mode
- Preventing timeouts in low-frequency or externally triggered acquisitions
- Matching USB communication timing with sensor characteristics

Remarks:

- The device **must be closed** before calling this function
- Call `ls_closedevice` first if a device is currently open
- Use `ls_geteptimeout` to query the current timeout value
- This timeout applies to all subsequent read operations after opening the device

Usage Example (Pascal):

```
var
  res: DWORD;
begin
  res := ls_closedevice;
  res := ls_seteptimeout(1000); // Set to 1 second
  if res = 0 then
    ShowMessage('Timeout set successfully.')
  else
    ShowMessage('Failed to set timeout: ' + String(ls_geterrorstring(res)));
end;
```

GetEpTimeout

Pascal Declaration:

```
function ls_geteptimeout: DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_geteptimeout(void);
```

Description:
The `ls_geteptimeout` function returns the current timeout setting for USB IN endpoint transfers. This value determines how long the DLL's internal read thread will wait for incoming data before reporting a timeout error.

Return Value:

- The timeout value in milliseconds (e.g., 1000 = 1 second)
- Returns 0 if the timeout is explicitly set to zero or if an internal error occurs

Use Cases:

- Checking the current USB timeout configuration
- Logging communication parameters for diagnostics
- Adapting software behavior based on latency requirements

Remarks:

- This value is read from the DLL's internal configuration
- Use `ls_seteptimeout` to change the timeout — but only when no device is open
- This timeout affects all USB bulk read operations once the device is opened
- A timeout that is too short may cause frequent transfer errors in slow acquisition modes

Usage Example (Pascal):

```
var
  timeout: DWORD;
begin
  timeout := ls_geteptimeout;
  ShowMessage('Current USB timeout: ' + IntToStr(timeout) + ' ms');
end;
```

GetPacketLength

Pascal Declaration:

```
function ls_getpacketlength(var dwPacketLength: DWORD): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getpacketlength(UInt32* dwPacketLength);
```

Description:
The `ls_getpacketlength` function queries the currently opened USB LISM-PI26xx device for its recommended USB packet length — the number of bytes it will transfer per USB transaction.

This value is **read directly from the device**, based on its current internal configuration, particularly the **number of pixels per image** and the **number of buffered images per transfer**.

Parameter:

- `dwPacketLength`: A reference to a `DWORD` variable that will be filled with the packet length (in bytes).

Return Value:

- 0: Success — the value was successfully read from the device
- Non-zero: Error code (e.g., if no device is open or communication failed)

Use Cases:

- Ensuring the host application uses a correct, device-aligned packet size
- Adapting buffer allocation based on current sensor settings

- Validating configuration changes before applying with `ls_setpacketlength`

Remarks:

- This function requires an **open device**
- The returned value is determined by the module's current configuration, such as pixel count and internal buffering
- The LISM-PI26xx module always operates in **16-bit mode**, so packet length corresponds to `2 × pixel count × image count`
- The result may change after adjusting parameters like `ls_setcfg1` or pixel resolution

Usage Example (Pascal):

```
var
  pktLen: DWORD;
begin
  if ls_getpacketlength(pktLen) = 0 then
    ShowMessage('Recommended packet length: ' + IntToStr(pktLen))
  else
    ShowMessage('Failed to get packet length.');
end;
```

SetPacketLength

Pascal Declaration:

```
function ls_setpacketlength(dwPacketLength: DWORD): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_setpacketlength(UInt32 dwPacketLength);
```

Description:
The `ls_setpacketlength` function sets the expected USB packet length (in bytes) for the internal read mechanism of the DLL. This ensures that the DLL's software-side buffer size matches the actual packet size used by the LISM-PI26xx USB device, avoiding misaligned or incomplete data transfers.

Since the module always operates in 16-bit mode, each pixel uses 2 bytes. The correct packet length is therefore:

`2 × pixel count × image count`
or any **integer multiple** of that value.

The image count is configured via `ls_setcfg1`.

Parameter:

- `dwPacketLength`: Desired number of bytes per USB transfer. Must match or be a multiple of the actual transfer size of the device.

Return Value:

- 0: Success — the packet length was set in the DLL
- Non-zero: Error code (e.g., if the value is invalid or a device is currently open)

Use Cases:

- Synchronizing DLL-side expectations with the hardware's configured transfer size
- Avoiding buffer overflows or misaligned data parsing
- Preparing for specific frame sizes based on pixel and image count

Remarks:

- The device must be closed before calling this function
- Use `ls_closedevice` first if needed
- Use `ls_getpacketlength` to query the correct value from the hardware
- An incorrect packet length setting may lead to truncated, shifted, or misinterpreted image data

Usage Example (Pascal):

```
var
  res: DWORD;
begin
  res := ls_closedevice;
  res := ls_setpacketlength(4096); // For 2048 pixels and 1 image per transfer
  if res = 0 then
    ShowMessage('Packet length set successfully.')
  else
    ShowMessage('Error: ' + String(ls_geterrorstring(res)));
end;
```

Data Transfer

This section outlines the functions used to retrieve image data from the LISM-PI26xx module via the USB interface. Once the device has been properly initialized and opened, the internal firmware starts streaming image data to the host. This data is continuously collected by a dedicated background thread within the DLL and buffered in a host-side ring buffer for asynchronous access. The host application can then access this buffered data at any time, depending on the required acquisition strategy.

The data transfer mechanism is designed for high-speed and low-latency applications, where consistent throughput and reliable synchronization are essential. Each USB transaction delivers a fixed number of bytes that represent one or more complete image frames. The structure of these transfers — including the number of images grouped per transaction and whether a frame identifier is appended — can be configured in advance. This configuration enables precise control over transfer sizes and facilitates tracking of individual frames, which is especially useful for error detection in real-time systems.

The interface supports both polling and blocking access models. Applications can choose to wait until new data is available or check the buffer state at their own pace. This flexibility allows seamless integration into different system architectures, ranging from real-time control loops to multi-threaded image processing pipelines.

The raw image data is transmitted in a 16-bit pixel format and must be interpreted according to the currently active sensor and acquisition settings. It is the responsibility of the host software to handle alignment, parsing, and any higher-level image formatting or processing.

Overall, this group of functions provides the essential data path between the USB LISM-PI26xx hardware and the application layer. It enables robust, high-performance acquisition suitable for use in industrial automation, scientific imaging, inspection systems, and other demanding environments where throughput and timing reliability are critical.

SetCFG1

Pascal Declaration:

```
function ls_setcfg1(ucCFG1: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_setcfg1(UInt8 ucCFG1);
```

Description:
The `ls_setcfg1` function sets the value of the internal configuration register CFG1 on the USB LISM-PI26xx device. This register controls low-level parameters related to USB transfer formatting, including whether to append an image number and how many images are buffered per transfer.

Bit layout of ucCFG1:

Bit(s)	Function
0	Image number append enable
1	Reserved (ignored)
2–6	Number of buffered images − 1
7	Reserved (ignored)

Bit 0 – Image number append:

- 0: No image number appended
- 1: A 4-byte image number is appended at the end of each image frame
    - This replaces the last two pixel values
    - Useful for frame tracking and loss detection

Bits 2–6 – Buffered image count per USB transfer:

- Encodes the number of images to buffer before triggering a USB transfer
- Formula: `image count = (value of bits 2–6) + 1`
- **The effective packet length becomes:**
  `packet_length = image_size × image_count`
  where `image_size = pixel count × 2`

Important constraint:

- The maximum number of images per transfer depends on the pixel count
- Example limits:
    - `4096 pixels`: max. 2 images

  - `2048 pixels`: max. 4 images
  - `1024 pixels`: max. 8 images
  - `512 pixels`: max. 16 images
  - `256 pixels`: max. 32 images
- **If the image count is too high for the current pixel resolution, the device will clip or ignore excess data.**

Parameter:

- `ucCFG1`: The configuration byte to be written. Must be constructed according to the bit layout above.

Return Value:

- 0: Success
- Non-zero: Error during communication or invalid state

Remarks:

- Changes to CFG1 only take effect **after a power cycle**
- This function configures low-level USB behavior — use with care
- Recommended: call `ls_getpacketlength` after setting this value to confirm the resulting transfer size

Usage Example (Pascal):

```
// Enable image number and buffer 4 images per transfer (buffer count = 3)
ls_closedevice;
ls_setcfg1($1D); // %00011101 = Bit 0 = 1, Bits 2–6 = 3
```

GetCFG1

Pascal Declaration:

```
function ls_getcfg1(var ucCFG1: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getcfg1(UInt8* ucCFG1);
```

Description:
The `ls_getcfg1` function reads the current value of the CFG1 configuration register from the USB LISM-PI26xx device. This register defines internal USB transfer behavior, such as image numbering and the number of buffered images per transfer.

The register layout is identical to `ls_setcfg1` and provides insight into the module's low-level configuration.

Bit layout of ucCFG1:

Bit(s)	Function
0	Image number append enable
1	Reserved (ignored)
2–6	Number of buffered images − 1
7	Reserved (ignored)

Usage Details:

- **Bit 0 = 1**: A 4-byte image number is appended to each frame (replacing last two pixels)
- **Bits 2–6**: Number of images per USB transfer = (value + 1)
- The actual transfer size in bytes is:
  `packet_length = pixel count × 2 × image count`

Parameter:

- `ucCFG1`: A reference to a byte variable that will receive the current register value

Return Value:

- 0: Success
- Non-zero: Error if device is not open or communication failed

Remarks:

- This function is the counterpart to `ls_setcfg1`
- The device must be open before calling
- The returned value reflects the last value written to the hardware (takes effect only after power cycle)

Usage Example (Pascal):

```
var
  cfg: Byte;
begin
  if ls_getcfg1(cfg) = 0 then
    ShowMessage('CFG1 value: ' + IntToStr(cfg));
end;
```

WaitForPipe

Pascal Declaration:

```
function ls_waitforpipe(dwTimeOut: DWORD): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_waitforpipe(UInt32 dwTimeOut);
```

Description:
The `ls_waitforpipe` function blocks the calling thread until new data becomes available in the DLL's internal ring buffer (pipe), or until the specified timeout period elapses.

It is typically used in polling-based or threaded acquisition loops to wait for image data from the device before calling `ls_getpipe`.

Parameter:

- `dwTimeOut`: Maximum wait time in milliseconds
  - `0` = non-blocking check
  - `INFINITE` (DWORD(-1)) = wait indefinitely until data arrives

Return Value:

- 0: Success — data is available in the pipe
- Non-zero: Timeout occurred or an error (e.g., no device open)

Use Cases:

- Synchronizing image data retrieval in separate threads
- Waiting efficiently for data in externally triggered acquisition setups
- Avoiding unnecessary CPU usage in acquisition loops

Remarks:

- This function does **not** retrieve any data; it only waits
- After a successful wait, use `ls_getpipe` to read the image data
- `ls_waitforpipe` is optional — you can call `ls_getpipe` directly at any time
- Internally, the function uses event-based waiting
- Typical usage pattern:

```
if ls_waitforpipe(1000) = 0 then
  ls_getpipe(...);
```

Usage Example (Pascal):

```
var
  res: DWORD;
begin
  res := ls_waitforpipe(1000); // Wait up to 1 second
  if res = 0 then
    ShowMessage('Data available.')
  else
    ShowMessage('Timeout or error.');
end;
```

GetPipe

Pascal Declaration:

```
function ls_getpipe(lpBuffer: Pointer; dwToRead: DWORD; var dwRead: DWORD):
DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getpipe(void* lpBuffer, UInt32 dwToRead, UInt32* dwRead);
```

Description:
The `ls_getpipe` function reads raw image data from the DLL's internal ring buffer (pipe), which receives continuous data streams from the USB LISM-PI26xx device. It is the primary method for accessing image data during acquisition.

Parameters:

- `lpBuffer`: Pointer to a buffer where the data will be copied. If `nil`, no data is copied.
- `dwToRead`: Number of bytes to read. If 0, no data is copied.
- `dwRead`: Variable that receives the number of bytes actually read, or (in non-copy mode) the number of bytes currently available.

Return Value:

- 0: Success
- Non-zero: Error (e.g., no device open, invalid parameters, USB read error)

Use Cases:

- Retrieving full or partial frames of image data
- Reading data into memory or saving to disk
- Polling buffer fill state to monitor throughput

Data Format:

- Data is transmitted as raw **16-bit pixels**
- For example:
  - `2048 pixels/line → 4096 bytes/line`
  - `packet_length = 2 × pixel count × image count`

Blocking Behavior:

- If `dwToRead > 0` and `lpBuffer` is not `nil`:
  → The function **blocks indefinitely** until at least `dwToRead` bytes are available in the pipe
- If `dwToRead = 0` or `lpBuffer = nil`:
  → The function returns **immediately**, and `dwRead` reports the number of available bytes

Remarks:

- You may call `ls_waitforpipe` beforehand, but it is optional
- It is the responsibility of the application to manage image alignment and interpret the raw data
- This function is typically used in continuous acquisition loops or threaded contexts

Usage Example (Pascal):

```
var
  buffer: array[0..4095] of Byte;
  readBytes: DWORD;
begin
  if ls_getpipe(@buffer, 4096, readBytes) = 0 then
    ShowMessage('Read ' + IntToStr(readBytes) + ' bytes of image data.')
  else
    ShowMessage('Failed to read image data.');
end;
```

GetFPS

Pascal Declaration:

```
function ls_getfps: DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getfps(void);
```

Description:
The `ls_getfps` function returns the current USB data throughput from the USB LISM-PI26xx device to the host, measured in bytes per second. It reflects how quickly image data is being received and can be used to estimate the actual acquisition rate.

Return Value:

- Number of bytes transferred per second
- Returns 0 if no device is open, no data is being transferred, or an error occurs

Use Cases:

- Real-time monitoring of acquisition speed
- Estimating actual line/frame rate
- Logging or validating USB performance over time

Typical Calculation:
To calculate the effective line rate (lines per second), divide by the image line size:
`line_rate := ls_getfps / (pixel_count × 2)`
Example for 2048 pixels per line:
`line_rate := ls_getfps / 4096`

Remarks:

- The result is internally averaged and updated by the DLL
- The function requires that a device is currently open
- Meaningful values are only returned during active image data transfer

Usage Example (Pascal):

```
var
  fps, linesPerSec: DWORD;
begin
  fps := ls_getfps;
  linesPerSec := fps div (2048 * 2);
  ShowMessage('Line rate: ' + IntToStr(linesPerSec) + ' lines/sec');
end;
```

I²C Bus Interface

This section provides a complete interface for communicating with internal and external I²C devices through the USB LISM-PI26xx interface. The I²C bus is a central element of the system's configuration and control architecture, enabling the host to directly access sensor registers, timing generators, EEPROMs, and other peripherals — both on the module itself and on the user-defined external I²C bus.

The USB LISM-PI26xx hardware includes an integrated I²C multiplexer that distinguishes between two logical channels: the internal bus (channel 0), which connects to the LISM-PI26xx module, and the external bus (channel 1), which is available for user-specific I²C devices such as additional sensors, memory devices, or microcontrollers. The bus switching is handled automatically in most cases, but can also be controlled manually for advanced scenarios.

The API includes low-level functions for reading and writing raw bytes, words, and 32-bit values from/to I²C devices, as well as high-level operations for accessing specific registers using structured commands. It also provides mechanisms to change and store I²C slave addresses, making it easy to assign unique addresses to multiple modules or to resolve conflicts on shared buses.

A key feature of this interface is its flexibility. Developers can use it to implement their own register protocols, support device-specific configurations, or extend the LISM-PI26xx system with custom hardware. All I²C communication is performed through the USB interface, and the host acts as the I²C master at all times.

This function group is particularly useful in system integration, calibration setups, or production environments where dynamic hardware configuration and diagnostics are required. It bridges the gap between USB-based host software and the sensor's internal control logic, providing fine-grained access to configuration data and real-time system state.

SetSlaveAddress

Pascal Declaration:

```
function ls_setslaveaddress(addr, newaddr: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_setslaveaddress(UInt8 addr, UInt8 newaddr);
```

Description:
The `ls_setslaveaddress` function writes a new I²C slave address to the `TEMP_SLAVE_ADDRESS` register (0xF0) of the LISM-PI26xx module via I²C. This allows temporary reconfiguration of the

module's I²C address — particularly useful in systems with multiple LISM-PI26xx modules or bus conflicts.

The new address is not applied immediately. It is only stored in a volatile register and will not take effect until:

1. The address is explicitly saved to EEPROM, and
2. The module is restarted (power cycle)

Parameters:

- `addr`: Current I²C slave address of the LISM-PI26xx module
- `newaddr`: New I²C slave address to assign (valid range: 0x02 - 0xFE)

Return Value:

- 0 on success
- Non-zero error code if the I²C write fails

Behavior:

- Writes 1 byte to register 0xF0 (`TEMP_SLAVE_ADDRESS`)
- The module continues to respond to `addr` until restart and EEPROM save
- After reboot, if saved, `newaddr` becomes the active I²C address

Technical note:

- EEPROM save must be triggered via `ls_savesettings` or similar
- The new address takes effect only after a full power cycle (USB disconnect)
- Until then, the module must be addressed using the current `addr`

Notes:

- Does not affect USB communication or device enumeration
- Useful for initial provisioning or dynamic address assignment in multi-device systems
- Not persistent without EEPROM commit

Use cases:

- Assign unique addresses to multiple modules on the same I²C bus
- Resolve address conflicts during system configuration
- Prepare a new address remotely without immediate interruption

GetSlaveAddress

Pascal Declaration:

```
function ls_getslaveaddress(addr: Byte; var newaddr: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getslaveaddress(UInt8 addr, UInt8* newaddr);
```

Description:
The `ls_getslaveaddress` function reads the 8-bit value from the `TEMP_SLAVE_ADDRESS` register (0xF0) of the LISM-PI26xx module via I²C. This register holds the temporary I²C address most recently written using `ls_setslaveaddress`.

> ⚠ This function does **not** return the module's currently active I²C address — only the address that will become active **after a full power-cycle**, if retained.

Parameters:

- `addr`: Current I²C slave address of the LISM-PI26xx module
- `newaddr`: Output variable to receive the temporary address stored in register 0xF0

Return Value:

- 0 on success
- Non-zero error code if the I²C read fails

Behavior:

- Reads 1 byte from register 0xF0 (`TEMP_SLAVE_ADDRESS`)
- The returned value indicates the pending new I²C address
- This address is only used after a **power cycle** and if properly saved

Technical note:

- A call to `ls_saveslaveaddress` must be made to store the temporary address in EEPROM
- Until the module is restarted, it continues to respond to the currently active address

Notes:

- Use this function to confirm or log address changes before applying them
- For safety, always confirm the stored value before performing a reboot
- Recommended for use in configuration software or installation tools

Use cases:

- Display pending I²C address for user confirmation
- Automate verification before EEPROM commit
- Debug address assignment issues in multi-device systems

SaveSlaveAddress

Pascal Declaration:

```
function ls_saveslaveaddress(addr: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_saveslaveaddress(UInt8 addr);
```

Description:
The `ls_saveslaveaddress` function sends a store command to the `STORE_SLAVE_ADDRESS` register (0xF1) of the LISM-PI26xx module via I²C. This command causes the module to write the currently configured temporary I²C address (from register 0xF0) into non-volatile EEPROM memory.

After a power-cycle, the module will adopt the newly stored address as its permanent I²C slave address.

Parameters:

- `addr`: The current I²C slave address of the LISM-PI26xx module (used to access the module for this command)

Return Value:

- 0 on success
- Non-zero error code if the I²C write fails or the command is not acknowledged

Behavior:

- Executes a special command write to register 0xF1 (`STORE_SLAVE_ADDRESS`)
- Copies the value from `TEMP_SLAVE_ADDRESS` (0xF0) into EEPROM
- The address will take effect after a power-cycle

Technical note:

- No parameters are passed to the register — it is a command-only write
- This function must be called **after** `ls_setslaveaddress` to make the change persistent

- If skipped, the address is lost after power-off

Notes:

- Does **not** apply the new address immediately
- After saving, the module continues to respond to the old address until power is removed
- Useful in production or system integration environments where I²C address configuration must survive reboots

Use cases:

- Finalize a new I²C address assignment
- Prevent address loss after system power-down
- Configure modules in multi-device setups with unique addresses

SetMuxChannel

Pascal Declaration:

```
function ls_setmuxchannel(ucmux: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_setmuxchannel(UInt8 ucmux);
```

Description:
The `ls_setmuxchannel` function selects the active I²C bus channel on the USB LISM-PI26xx. The internal I²C multiplexer allows switching between two separate buses:

- **Channel 0**: Connected internally to the LISM-PI26xx module (default)
- **Channel 1**: Exposed to external I²C devices (e.g., user sensors, EEPROMs)

This mechanism allows secure and isolated access to user-defined I²C hardware while ensuring safe operation of the internal module.

Channel Mapping:

ucmux	I²C Channel	Description
0	Internal	For configuring and operating LISM-PI26xx
1	External (user bus)	For communicating with external I²C devices

Parameter:

- `ucmux`: I²C channel to activate (0 or 1)

Return Value:

- 0: Success — I²C multiplexer was switched
- Non-zero: Error (e.g., invalid value or communication failure)

Use Cases:

- Accessing external I²C components through the USB interface
- Separating internal system communication from external devices
- Custom applications that require direct control of I²C routing

Remarks:

- Only one I²C channel is active at any given time
- The selected channel applies immediately to all subsequent I²C transactions
- The USB connection to the host remains unaffected by this change
- **In normal operation, the hardware automatically switches to the required I²C channel depending on the target.**
  Manual selection via `ls_setmuxchannel` is only necessary for special use cases involving external I²C access

Usage Example (Pascal):

```
var
  res: DWORD;
begin
  res := ls_setmuxchannel(1); // Switch to external I²C bus
  if res = 0 then
    ShowMessage('External I²C channel activated.')
  else
    ShowMessage('Failed to switch I²C channel.');
end;
```

GetMuxChannel

Pascal Declaration:

```
function ls_getmuxchannel(var cmux: Shortint): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getmuxchannel(Int8* cmux);
```

Description:
The `ls_getmuxchannel` function queries the current state of the I²C multiplexer integrated in the USB LISM-PI26xx. It reports which I²C bus is currently active for communication — either the

internal bus connected to the LISM-PI26xx module or the external bus available for user-defined I²C devices.

This allows the host system to verify the communication path before initiating I²C transactions.

Return values via `cmux`:

cmux	I²C Channel	Description
0	Internal	Connected to LISM-PI26xx module (sensor/control)
1	External	Connected to user I²C bus (external devices)
-1	None / Error	No valid channel active or switch/read error

Parameter:

- `cmux`: Output variable that receives the currently selected I²C channel (type `Shortint` to allow for -1)

Return Value:

- 0: Success — active channel reported via `cmux`
- Non-zero: Error — reading multiplexer state failed

Use Cases:

- Verifying which I²C bus is currently active
- Logging current bus state for debugging or status display
- Ensuring that I²C transactions are routed to the intended target

Remarks:

- This function reads the multiplexer state directly from the USB LISM-PI26xx hardware
- A `cmux` value of -1 indicates that no valid channel is active or a hardware fault occurred
- Only one I²C channel can be active at any time
- Use `ls_setmuxchannel` to change the I²C bus if needed (manual override)

Usage Example (Pascal):

```
var
  channel: Shortint;
begin
  if ls_getmuxchannel(channel) = 0 then
  begin
    if channel = 0 then
      ShowMessage('Active I²C bus: internal (LISM-PI26xx)')
    else if channel = 1 then
      ShowMessage('Active I²C bus: external (user devices)')
    else
      ShowMessage('No valid I²C bus active.');
  end
  else
    ShowMessage('Error reading I²C channel.');
end
```

ReadI2C_Ext

Pascal Declaration:

```
function ls_readi2c_ext(i2caddr: Byte; pBuf: Pointer; var wLength: Word): DWORD;
stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_readi2c_ext(UInt8 i2caddr, void* pBuf, UInt16* wLength);
```

Description:
The `ls_readi2c_ext` function reads data from an external I²C device connected to the USB LISM-PI26xx interface's I²C multiplexer channel 1. Upon invocation, the controller automatically switches to channel 1 (external bus) and performs the read operation as an I²C master.

This mechanism allows the host PC to access user-defined I²C hardware (e.g., sensors, EEPROMs) independently of the LISM-PI26xx module.

Parameters:

- `i2caddr`: I²C slave address of the external device to communicate with
- `pBuf`: Pointer to a buffer where the received bytes will be stored
- `wLength`:
    - **Input**: Maximum number of bytes to read
    - **Output**: Actual number of bytes read and written into `pBuf`

Return Value:

- 0: Success — data was read successfully
- Non-zero: I²C error (e.g., no ACK, timeout, invalid device, buffer issue)

Behavior:

- The USB controller automatically switches to **channel 1** (external bus) before initiating the read
- After the read, **channel 1 remains active** unless changed manually via `ls_setmuxchannel`
- The read is performed using the controller's built-in I²C master functionality

Use Cases:

- Reading sensor data, EEPROM contents, or configuration values from external devices
- Host-based diagnostics or reconfiguration of embedded I²C components
- Generic I²C master communication initiated from the PC

Remarks:

- This function operates independently of the LISM-PI26xx module itself (which is connected to channel 0)
- For writing to external I²C devices, use a complementary function such as `ls_writei2c_ext`
- **Concurrent access** to both I²C channels (internal and external) is not supported — use sequential access
- Ensure that external devices are properly powered and pull-up resistors are in place

Usage Example (Pascal):

```
var
  res: DWORD;
  buf: array[0..15] of Byte;
  len: Word;
begin
  len := 16;
  // Read 16 bytes from slave address $50
  res := ls_readi2c_ext($50, @buf, len);
  if res = 0 then
    ShowMessage('Read ' + IntToStr(len) + ' bytes from external I²C device.')
  else
    ShowMessage('I²C read failed: ' + String(ls_geterrorstring(res)));
end;
```

WriteI2C_Ext

Pascal Declaration:

```
function ls_writei2c_ext(i2caddr: Byte; pBuf: Pointer; wLength: Word): DWORD;
stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_writei2c_ext(UInt8 i2caddr, const void* pBuf,
                                 UInt16 wLength);
```

Description:
The `ls_writei2c_ext` function writes data to an external I²C device connected to I²C multiplexer channel 1 of the USB LISM-PI26xx interface. When called, the controller **automatically switches to channel 1**, which is routed to the external I²C bus for user-defined hardware.

This enables isolated and host-controlled communication with I²C peripherals such as sensors, EEPROMs, or microcontrollers, without interfering with the internal bus used by the LISM-PI26xx module.

Parameters:

- `i2caddr`: I²C slave address of the target external device
- `pBuf`: Pointer to the buffer containing data to write
- `wLength`: Number of bytes to write from the buffer

Return Value:

- 0: Success — data was transmitted to the I²C slave
- Non-zero: I²C error (e.g., no ACK, write timeout, invalid device address)

Behavior:

- Automatically switches the I²C multiplexer to **channel 1** before performing the operation
- Writes `wLength` bytes to the I²C device at the specified address
- The active channel remains at 1 after the operation, unless changed manually via `ls_setmuxchannel`

Use Cases:

- Sending configuration commands or register writes to external I²C sensors
- Writing data to serial EEPROMs or actuators connected to the external I²C bus
- Creating a host-based I²C master interface for general-purpose communication

Remarks:

- This function is complementary to `ls_readi2c_ext`, which performs reading
- Both internal and external I²C buses may use the same I²C addresses — they are isolated by the multiplexer
- Only one I²C channel can be active at any given time — ensure sequential access if mixing internal and external I²C operations
- The USB connection to the host remains unaffected by the I²C channel switch

Usage Example (Pascal):

```
var
  data: array[0..1] of Byte;
  res: DWORD;
begin
  data[0] := $01; // Register address
  data[1] := $55; // Value to write
  res := ls_writei2c_ext($50, @data, 2); // Send 2 bytes to device at address
$50
  if res = 0 then
    ShowMessage('I²C write successful.')
  else
    ShowMessage('I²C write failed: ' + String(ls_geterrorstring(res)));
end;
```

ReadI2C

Pascal Declaration:

```
function ls_readi2c(i2caddr: Byte; pBuf: Pointer; var wLength: Word): DWORD;
stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_readi2c(UInt8 i2caddr, void* pBuf, UInt16* wLength);
```

Description:
The `ls_readi2c` function reads data from the LISM-PI26xx module via its internal I²C bus (multiplexer channel 0). When called, the USB controller automatically switches to channel 0 to ensure a direct and isolated connection to the internal LISM-PI26xx I²C interface.

This function is typically used to access internal registers of components like the timing generator, configuration processor, or sensor interface logic.

Parameters:

- `i2caddr`: I²C slave address of the internal LISM-PI26xx device

- `pBuf`: Pointer to a buffer where the received data will be stored
- `wLength`:
    - **Input**: Maximum number of bytes to read
    - **Output**: Actual number of bytes read

Return Value:

- 0: Success — data was read correctly
- Non-zero: I²C error (e.g., no ACK, device not responding, bus error)

Behavior:

- Automatically switches the I²C multiplexer to **channel 0**
- Performs a standard I²C read using the specified slave address
- Leaves channel 0 selected after completion

Use Cases:

- Reading internal configuration registers
- Accessing timing or sensor status values
- Implementing low-level I²C transactions for diagnostics or control

Remarks:

- Use `ls_readi2c_ext` for reading from **external** I²C devices (channel 1)
- The buffer pointed to by `pBuf` must be large enough to hold the expected number of bytes
- This function performs a **raw I²C read** — it does not interpret or validate the received content
- Only one channel can be active at a time — this function sets the correct one automatically

Usage Example (Pascal):

```
var
  res: DWORD;
  data: array[0..3] of Byte;
  len: Word;
begin
  len := 4;
  res := ls_readi2c($FE, @data, len); // Read 4 bytes from LISM-PI26xx module
  if res = 0 then
    ShowMessage('Read ' + IntToStr(len) + ' bytes from internal I²C.')
  else
    ShowMessage('Read failed: ' + String(ls_geterrorstring(res)));
end
```

WriteI2C

Pascal Declaration:

```
function ls_writei2c(i2caddr: Byte; pBuf: Pointer; wLength: Word): DWORD;
stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_writei2c(UInt8 i2caddr, const void* pBuf, UInt16 wLength);
```

Description:
The `ls_writei2c` function sends data to the LISM-PI26xx module via the internal I²C bus (multiplexer channel 0). The USB controller automatically switches to channel 0 before initiating the write operation, ensuring direct and safe access to the internal registers of the module.

This function is intended for low-level communication and allows custom I²C commands to be issued by the host application.

Parameters:

- `i2caddr`: I²C slave address of the internal LISM-PI26xx module (default is 0xFE)
- `pBuf`: Pointer to the data buffer to be transmitted
- `wLength`: Number of bytes to write

Return Value:

- 0: Success — all bytes transmitted
- Non-zero: I²C error (e.g., no ACK, invalid address, bus error)

Behavior:

- Automatically switches the I²C multiplexer to **channel 0**
- Writes `wLength` bytes to the device at `i2caddr`
- Leaves channel 0 active after execution

Use Cases:

- Writing directly to control or configuration registers
- Sending commands to the timing generator or sensor interface
- Implementing development or test routines outside the standard API scope

Remarks:

- Use `ls_writei2c_ext` to access external devices on channel 1

- This function performs a **raw I²C write** — the caller is responsible for formatting address and data according to the target's register map
- No response data is returned — use `ls_readi2c` for readback if needed

Usage Example (Pascal):

```
var
  res: DWORD;
  cmd: array[0..1] of Byte;
begin
  cmd[0] := $03; // Register address: MODE_CONFIG
  cmd[1] := $00; // Data: Free Running mode
  res := ls_writei2c($FE, @cmd, 2); // Write 2 bytes to LISM-PI26xx module
  if res = 0 then
    ShowMessage('Write successful.')
  else
    ShowMessage('Write failed: ' + String(ls_geterrorstring(res)));
end;
```

I2CRead1Byte

Pascal Declaration:

```
function ls_i2cread1byte(i2caddr, i2creg: Byte; var ucvalue: Byte): DWORD;
stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_i2cread1byte(UInt8 i2caddr, UInt8 i2creg, UInt8* ucvalue);
```

Description:
The `ls_i2cread1byte` function reads a single byte from a specific I²C register of the LISM-PI26xx module. It performs a standard two-step I²C transaction:

1. Write the register address (`i2creg`)
2. Read one byte from that address

The function automatically switches the I²C multiplexer to **channel 0**, ensuring access to the internal module bus without requiring manual channel control.

Parameters:

- `i2caddr`: I²C slave address of the LISM-PI26xx module (default is 0xFE)
- `i2creg`: Register address within the I²C device to read from
- `ucvalue`: Output variable to receive the value read from the register

Return Value:

- 0: Success — one byte was read from the register
- Non-zero: I²C error (e.g., no ACK, device not found, communication failure)

Use Cases:

- Quickly reading status, mode, or identification registers
- Simplifying typical read transactions into a one-liner
- Useful in polling or initialization logic

Remarks:

- The function performs a combined I²C write-then-read operation (repeated start)
- Only the internal I²C bus (channel 0) is accessed — for external devices, use `ls_readi2c_ext`
- The register address must be valid within the selected slave device
- For writing a single byte, use `ls_i2cwrite1byte`

Usage Example (Pascal):

```
var
  res: DWORD;
  value: Byte;
begin
  // Read INIT_STATUS (0xEE) from default address $FE
  res := ls_i2cread1byte($FE, $EE, value);
  if res = 0 then
    ShowMessage('Read value: ' + IntToStr(value))
  else
    ShowMessage('I²C read failed: ' + String(ls_geterrorstring(res)));
end;
```

I2CRead2Bytes

Pascal Declaration:

```
function ls_i2cread2bytes(i2caddr, i2creg: Byte; var wvalue: Word): DWORD;
stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_i2cread2bytes(UInt8 i2caddr, UInt8 i2creg, UInt16* wvalue);
```

Description:

The `ls_i2cread2bytes` function reads a 16-bit word from a specific I²C register of the LISM-

PI26xx module via the internal I²C bus (mux channel 0). The USB controller automatically switches to channel 0 and performs the read operation using the standard two-step I²C sequence:

1. Write: 1-byte register address (`i2creg`)
2. Read: 2 bytes from that register — first MSB, then LSB

The two bytes are combined into a 16-bit word (`wvalue`) using **big-endian** order.

Parameters:

- `i2caddr`: I²C slave address of the LISM-PI26xx module (default: 0xFE)
- `i2creg`: Register address to read from
- `wvalue`: Output variable to receive the 16-bit result

Return Value:

- 0: Success — the word was read successfully
- Non-zero: Error during I²C communication (e.g., NACK, no device response)

Use Cases:

- Reading 16-bit configuration or status registers
- Retrieving firmware version, pixel count, or timing values
- Efficient access to two-byte data structures

Remarks:

- Automatically switches to **mux channel 0** before reading
- Data is returned in **big-endian format**:
  `wvalue := (MSB shl 8) + LSB`
- For 1-byte reads, use `ls_i2cread1byte`
- For writing 2 bytes, use `ls_i2cwrite2bytes`

Usage Example (Pascal):

```
var
  res: DWORD;
  wordval: Word;
begin
  // Read SENSOR_BOARD_ID (0xFA) from address $FE
  res := ls_i2cread2bytes($FE, $FA, wordval);
  if res = 0 then
    ShowMessage('Read word: ' + IntToStr(wordval))
  else
    ShowMessage('I²C read failed: ' + String(ls_geterrorstring(res)));
end;
```

I2CRead4Bytes

Pascal Declaration:

```
function ls_i2cread4bytes(i2caddr, i2creg: Byte; var dwvalue: DWORD): DWORD;
stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_i2cread4bytes(UInt8 i2caddr, UInt8 i2creg, UInt32* dwvalue);
```

Description:
The `ls_i2cread4bytes` function reads a 32-bit (4-byte) value from a specified register of the LISM-PI26xx module via its internal I²C bus. The USB controller automatically switches to **mux channel 0** to access the module and performs a standard combined I²C transaction:

1. Write the register address (`i2creg`)
2. Read 4 consecutive bytes from that register (starting at `i2creg`)

The result is returned as a `DWORD` in **big-endian** order.

Parameters:

- `i2caddr`: I²C slave address of the LISM-PI26xx module (default: 0xFE)
- `i2creg`: Starting register address to read from
- `dwvalue`: Output variable to receive the 32-bit result

Return Value:

- 0: Success — value was read correctly
- Non-zero: Error during I²C communication

Use Cases:

- Reading 32-bit configuration values such as `START_PULSE_HIGH`, `SOFT_TRIGGER_PERIOD`, or timing delays
- Efficient access to 4-byte status or control registers
- Logging or exporting hardware configuration data

Remarks:

- Automatically switches to **mux channel 0** before reading
- Data is returned in **big-endian** order:
  `dwvalue := (B0 shl 24) + (B1 shl 16) + (B2 shl 8) + B3`
- Use `ls_i2cread1byte` or `ls_i2cread2bytes` for smaller registers

- This function simplifies multi-byte I²C reads by handling buffer assembly internally

Usage Example (Pascal):

```
var
  res: DWORD;
  value: DWORD;
begin
  // Read SOFT_TRIGGER_PERIOD (0x08) from $FE
  res := ls_i2cread4bytes($FE, $08, value);
  if res = 0 then
    ShowMessage('Read DWORD: ' + IntToStr(value))
  else
    ShowMessage('Read failed: ' + String(ls_geterrorstring(res)));
end;
```

I2CWrite1Byte

Pascal Declaration:

```
function ls_i2cwrite1byte(i2caddr, i2creg: Byte; ucvalue: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_i2cwrite1byte(UInt8 i2caddr, UInt8 i2creg, UInt8 ucvalue);
```

Description:
The `ls_i2cwrite1byte` function writes a single byte to a specific I²C register of the LISM-PI26xx module using the internal I²C bus (mux channel 0). The USB controller automatically switches to channel 0 and performs a standard I²C write sequence:

1. Send the register address (`i2creg`)
2. Send the data byte (`ucvalue`)

This function is a convenient way to write control or configuration values directly from the host PC.

Parameters:

- `i2caddr`: I²C slave address of the LISM-PI26xx module (default: 0xFE)
- `i2creg`: Register address to write to
- `ucvalue`: The value to write into the register

Return Value:

- 0: Success — byte was written successfully
- Non-zero: I²C communication error

Use Cases:

- Setting modes or flags in configuration registers
- Triggering actions (e.g. `START_ACQUISITION`)
- Writing to control bits without multi-byte transactions

Remarks:

- Automatically switches to **mux channel 0**
- Equivalent to a 2-byte I²C write: `[register address][data byte]`
- Use `ls_i2cwrite2bytes` or `ls_i2cwrite4bytes` for writing 16- or 32-bit registers
- This function does not return the result of the write — use `ls_i2cread1byte` to verify, if needed

Usage Example (Pascal):

```
var
  res: DWORD;
begin
  // Write 0x01 to START_ACQUISITION (0x02)
  res := ls_i2cwrite1byte($FE, $02, $01);
  if res = 0 then
    ShowMessage('I²C write successful.')
  else
    ShowMessage('I²C write failed: ' + String(ls_geterrorstring(res)));
end;
```

I2CWrite2Bytes

Pascal Declaration:

```
function ls_i2cwrite2bytes(i2caddr, i2creg: Byte; wvalue: Word): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_i2cwrite2bytes(UInt8 i2caddr, UInt8 i2creg, UInt16 wvalue);
```

Description:
The `ls_i2cwrite2bytes` function writes a 16-bit word to a register of the LISM-PI26xx module over the internal I²C bus (mux channel 0). The USB controller automatically switches to channel 0 to ensure isolated and direct communication with the module.

The function performs a standard I²C write sequence:

1. Send the register address (`i2creg`)
2. Send the 16-bit word as two bytes: **MSB first**, then **LSB**

Parameters:

- `i2caddr`: I²C slave address of the LISM-PI26xx module (default: 0xFE)
- `i2creg`: Register address to write to
- `wvalue`: 16-bit value to be written to the register

Return Value:

- 0: Success — value written successfully
- Non-zero: I²C communication error (e.g., no ACK, bus error)

Use Cases:

- Writing to 16-bit registers such as `PIXEL_COUNT`, `LINES_PER_FRAME`
- Sending configuration values from the host
- Simplifying host code by avoiding manual byte-splitting

Remarks:

- The data is sent in **big-endian** order: MSB first, then LSB
- Automatically switches to **mux channel 0** before transmission
- For 1-byte or 4-byte writes, use `ls_i2cwrite1byte` or `ls_i2cwrite4bytes` respectively
- The function does not perform internal validation of register addresses or values

Usage Example (Pascal):

```
var
  res: DWORD;
begin
  res := ls_i2cwrite2bytes($FE, $0B, 2048); // Write 2048 to PIXEL_COUNT (0x0B)
  if res = 0 then
    ShowMessage('16-bit I²C write successful.')
  else
    ShowMessage('I²C write failed: ' + String(ls_geterrorstring(res)));
end;
```

I2CWrite4Bytes

Pascal Declaration:

```
function ls_i2cwrite4bytes(i2caddr, i2creg: Byte; dwvalue: DWORD): DWORD;
stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_i2cwrite4bytes(UInt8 i2caddr, UInt8 i2creg, UInt32 dwvalue);
```

Description:
The `ls_i2cwrite4bytes` function writes a 32-bit (DWORD) value to a specified register of the LISM-PI26xx module via the internal I²C interface. The USB controller automatically switches to multiplexer channel 0 to guarantee isolated communication with the sensor module.

The function performs the following I²C write sequence:

1. Sends the target register address (`i2creg`)
2. Sends the 32-bit value (`dwvalue`) in big-endian byte order:
    - Byte 3 (MSB) = `dwvalue >> 24`
    - Byte 2 = `dwvalue >> 16`
    - Byte 1 = `dwvalue >> 8`
    - Byte 0 (LSB) = `dwvalue & 0xFF`

Parameters:

- `i2caddr`: I²C slave address of the LISM-PI26xx module (default: 0xFE)
- `i2creg`: 8-bit register address to write to
- `dwvalue`: 32-bit value to be written to the register (DWORD)

Return Value:

- 0: Success — write completed without error
- Non-zero: I²C communication error (e.g., no ACK, bus error)

Use Cases:

- Writing to 32-bit registers like:
    - `START_PULSE_HIGH` (0x04)
    - `SOFT_TRIGGER_PERIOD` (0x08)
    - `TRIGGER_OUTPUT_WIDTH` (0x0A)

- Updating timing parameters in µs
- Fast configuration from host applications without manual byte handling

Remarks:

- Data is transmitted in big-endian format (MSB first)
- Automatically switches to multiplexer channel 0 before writing
- For writing 1 or 2 bytes, use `ls_i2cwrite1byte` or `ls_i2cwrite2bytes` instead
- No internal validation of register ranges or values is performed

Usage Example (Pascal):

```
var
  res: DWORD;
begin
  // Write 1000 µs to SOFT_TRIGGER_PERIOD (0x08)
  res := ls_i2cwrite4bytes($FE, $08, 1000);
  if res = 0 then
    ShowMessage('4-byte I²C write successful.')
  else
    ShowMessage('I²C write failed: ' + String(ls_geterrorstring(res)));
end;
```

I2CWriteCmd

Pascal Declaration:

```
function ls_i2cwritecmd(i2caddr, i2creg: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_i2cwritecmd(UInt8 i2caddr, UInt8 i2creg);
```

Description:
The `ls_i2cwritecmd` function writes a single-byte command to the LISM-PI26xx module via I²C by sending only the register address. No data bytes follow the address. This is specifically used for command-type registers that trigger internal actions when addressed. The USB controller automatically switches to mux channel 0 to enable isolated communication with the module.

The I²C write sequence consists of:

1. Sending the register address (`i2creg`)
2. No data transmission

Parameters:

- `i2caddr`: I²C slave address of the LISM-PI26xx module (default: 0xFE)
- `i2creg`: Command register to write to (e.g. `STORE_SETTINGS`, `RESET_SETTINGS`)

Return Value:

- `0`: Success — command sent and acknowledged
- Non-zero: I²C communication error (e.g., no ACK, bus error)

Use Cases:

- `STORE_SETTINGS`
- `STORE_SLAVE_ADDRESS`
- `RELOAD_SETTINGS`
- `RESET_SETTINGS`
- `HW_RESET`

Remarks:

- This function sends only the register address — no data bytes
- Used exclusively for write-only command registers that trigger internal actions
- Automatically switches to mux channel 0 before writing
- No effect on normal read/write configuration registers

Usage Example (Pascal):

```
var
  res: DWORD;
begin
  res := ls_i2cwritecmd($FE, $F2); // Trigger STORE_SETTINGS
  if res = 0 then
    ShowMessage('Command executed successfully.')
  else
    ShowMessage('Command failed: ' + String(ls_geterrorstring(res)));
end;
```

Clock and Timing Generator (CTG) Register Access

This section describes the complete set of functions used to configure and control the internal clock and timing generator (CTG) of the LISM-PI26xx module. The CTG is the core of the acquisition logic — it defines when and how image lines are captured, how trigger signals are interpreted, and how timing pulses are generated. Its behavior governs the temporal precision of every image frame and line, and therefore plays a central role in all measurement, inspection, and synchronization applications.

Access to the CTG is provided through a dedicated set of I²C-mapped registers, which allow the host application to configure parameters such as integration time, line period, trigger mode, signal polarities, and synchronization delays. These registers are grouped by function: acquisition control, trigger configuration, START pulse shaping, interface timing, and analog frontend settings (e.g. gain, offset, voltage references).

The interface allows for both one-time configuration during system initialization and dynamic reconfiguration at runtime. This includes switching between free-running and externally triggered modes, adjusting frame or line timing to match motion profiles, or adapting exposure parameters in response to changing lighting conditions. Changes take effect immediately or after a short synchronization sequence, depending on the register type.

Beyond configuration, the CTG function set includes tools to monitor and control the acquisition state, suspend and resume timing operations, apply updated parameters without full reset, and reset the timing generator to a known default state. Version and status registers provide insight into the current operating conditions and firmware compatibility of the timing engine.

All settings are applied through a consistent, low-level access model based on 1-, 2-, or 4-byte I²C transfers, with attention to timing-safe updates. The host is fully in control of the acquisition cycle and can manage every aspect of signal generation and data timing from software.

This level of control makes the LISM-PI26xx suitable for demanding applications such as encoder-synchronized scanning, precise trigger-aligned imaging, and complex multi-line frame structures. Whether the system operates continuously or under external control, the CTG register interface ensures deterministic timing behavior and complete transparency for the application developer.

HW_Reset

Pascal Declaration:

```
function ls_hw_reset(addr: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_hw_reset(UInt8 addr);
```

Description:
The `ls_hw_reset` function sends a hardware reset command to the clock and timing generator (P2) of the LISM-PI26xx module via I²C. This reset reinitializes the internal acquisition control logic, clears all volatile state, and reloads configuration data from EEPROM.

It does **not** affect the USB controller or communication processor (P1), and communication with the host remains active throughout the reset.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module (typically 0xFE unless reconfigured)

Return Value:

- 0 if the reset command was successfully sent
- Non-zero error code if the I²C transmission failed (e.g., invalid address or no response)

Behavior:

- Sends a command to the special write-only register 0xFF (`HW_RESET`)
- No data byte is required; the address write triggers the reset
- Causes the timing generator to stop, reset its logic, and reapply EEPROM-stored configuration

What it resets:

- Pixel clock and START pulse timing generator
- Acquisition state and mode configuration
- Trigger source and trigger logic (software, external, encoder)
- All non-persistent (volatile) control registers

Notes:

- EEPROM configuration is reloaded automatically after reset
- Communication remains active; USB connection is unaffected

- The slave address is **not** changed, even if a new address was previously stored
- Trigger mode, pulse lengths, and timing settings should be revalidated or reapplied

Use cases:

- Recover from invalid or undefined timing states
- Force module reinitialization without disconnecting power
- Apply saved settings without restarting the entire USB stack

Resume

Pascal Declaration:

```
function ls_resume(addr: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_resume(UInt8 addr);
```

Description:
The `ls_resume` function controls the internal clock and timing generator of the LISM-PI26xx module via I²C. Specifically, it writes the command to the `RESUME_GENERATOR` register (0xx00), which allows the system to either resume normal operation or suspend all timing activity.
To resume timing operation, the function sends the value 0xAA to the 0x00 register. Sending any other value (implicitly or via lower-level commands) would suspend the generator, but this function is designed to resume only.

Parameters:

- `addr`: The I²C slave address of the target LISM-PI26xx module (default 0xFE)

Return Value:

- 0 if the resume command was successfully written
- Non-zero error code if I²C communication failed

Effects of resume:

- Restarts pixel clock and line/frame generation
- Restores signal timing (`FRAME_VALID`, `LINE_VALID`, data clock)
- Preserves previously configured parameters (e.g. integration time, trigger settings)

Technical note:

- Sends a single-byte I²C write

- Register address: 0x00 (`RESUME_GENERATOR`)
- Data byte: 0xAA (resume command)
- The timing generator resumes only if 0xAA is received exactly
- Can return a NACK if sent during internal processing — retry after short delay if needed

Notes:

- The clock generator must be resumed before acquisition can begin
- The I²C connection must be active and correctly addressed
- Use `ls_resume` after `ls_suspend` to restore operation

Suspend

Pascal Declaration:

```
function ls_suspend(addr: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_suspend(UInt8 addr);
```

Description:
The `ls_suspend` function halts the internal clock and timing generator of the LISM-PI26xx module by sending a suspend command via I²C. This puts the module into an idle state in which no timing signals (such as `FRAME_VALID`, `LINE_VALID`, or pixel clock) are generated. However, the I²C interface remains active and fully accessible.
This function writes any value other than 0xAA (commonly 0x00) to the `RESUME_GENERATOR` register (0x00). This disables the generator without affecting the device configuration or EEPROM content.

Parameters:

- `addr`: The I²C slave address of the target module

Return Value:

- 0 on success
- Non-zero error code if the I²C write fails

Effects of suspension:

- All acquisition timing stops
- Pixel clock, data output, and trigger logic are disabled

- Internal state is paused but preserved
- Settings may still be changed over I²C

Technical note:

- Sends a single-byte I²C write
- Register address: 0x00 (`RESUME_GENERATOR`)
- Data byte: any value **except** 0xAA (typically 0x00)
- The suspend state is active until `ls_resume` is called
- Suspension takes effect immediately; no confirmation is sent by the module

Notes:

- The module can be resumed using `ls_resume(addr)` (sends 0xAA)
- Suspension is non-destructive and does not require reconfiguration afterward
- Acquisition should always be stopped (via `ls_setstate`) before suspending the generator, to avoid undefined timing behavior

UpdateParam

Pascal Declaration:

```
function ls_updateparam(addr: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_updateparam(UInt8 addr);
```

Description:
The `ls_updateparam` function sends the `UPDATE_PARAMS` command (0x01) to the clock and timing generator of a LISM-PI26xx module via I²C. This command instructs the module to read all relevant configuration registers and reapply their contents to the internal logic.
The purpose of this function is not to reinitialize or reset, but to synchronize the working state of the timing generator with the current values in registers such as:

- `START_PULSE_HIGH (0x04)`
- `START_PULSE_LOW (0x05)`
- `SOFT_TRIGGER_PERIOD (0x08)`
- `LINES_PER_FRAME (0x06)`
- and others affecting the timing generator

This ensures that all internal calculations are up to date and consistent with the latest control values.

Parameters:

- `addr`: The I²C slave address of the LISM-PI26xx module (default 0xFE)

Return Value:

- 0 if the update command was accepted
- Non-zero error code if the I²C command failed

Technical note:

- This function sends an I²C write with only a register address and no data
- Register: 0x01 `(UPDATE_PARAMS)`
- No data payload is required
- If the timing generator is currently busy (e.g. during acquisition), the module may respond with a NACK — retrying after a short delay is recommended

Notes:

- Should be called after modifying timing-related registers to apply changes
- Does not alter acquisition state, memory, or stored settings
- Can be called multiple times without side effects

SetState

Pascal Declaration:

```
function ls_setstate(addr, ucState: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_setstate(UInt8 addr, UInt8 ucState);
```

Description:
The `ls_setstate` function controls whether the LISM-PI26xx module is actively acquiring image data. It writes a value to the `START_ACQUISITION` register (0x02) over I²C to either start or stop the acquisition process.
This register directly controls the internal timing and control generator. When acquisition is active, the module begins generating start pulses and outputs synchronized image data according to the selected trigger mode.

Parameters:

- `addr`: I²C slave address of the target LISM-PI26xx module (default 0xFE)
- `ucState`: A byte that sets the acquisition state:
  - 0x01 → Start acquisition
  - 0x00 → Stop acquisition

Return Value:

- 0 if the command was successfully sent
- A non-zero error code if the I²C write failed

Behavior:

- Start (0x01):
  - Activates the internal generator and begins the acquisition cycle
  - Data clock, FRAME_VALID and LINE_VALID signals become active
  - The trigger configuration (software, external, encoder) determines timing
- Stop (0x00):
  - Acquisition stops after the current line is fully read out
  - All output signals then become inactive
  - The timing generator is paused but configuration remains intact

Technical note:

- Sends a one-byte I²C write
- Register address: 0x02 (`START_ACQUISITION`)
- Data byte: 0x00 (stop) or 0x01 (start)

Notes:

- Must be called after `ls_resume`, otherwise the timing generator is suspended and cannot run
- Can be called as needed while the timing generator is active

SetModeConfig

Pascal Declaration:

```
function ls_setmodeconfig(addr, ucConfig: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_setmodeconfig(UInt8 addr, UInt8 ucConfig);
```

Description:
The `ls_setmodeconfig` function writes a configuration byte to the `MODE_CONFIG` register (0x03) of the LISM-PI26xx module via I²C. This register defines the trigger mode used for acquisition as well as the quadrature encoder configuration.
It determines how image capture is initiated—via internal timer, external trigger, or encoder movement—and how encoder signals are interpreted.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module (default 0xFE)
- `ucConfig`: A byte value defining trigger mode and encoder settings

Bit layout of `ucConfig`:

Bits	Name	Value / Meaning
[2:0]	Trigger Mode	000 = Free Running
		001 = External Rising Edge (Single-Line)
		010 = External High Level
		011 = Internal Software Trigger
		100 = Quadrature Encoder Trigger
		101 = External Rising Edge (Multi-Line)
		110, 111 = Reserved (defaults to Free Running)
[3]	*Reserved*	*Not used, reserved for future use*
[4]	Encoder Direction	0 = Count UP (CW)
		1 = Count DOWN (CCW)
[5]	Encoder Counter Enable	0 = Counter disabled (trigger on every pulse)
		1 = Trigger after QUAD_COUNT pulses
[6–7]	*Reserved*	*Not used, reserved for future use*

Return Value:

- 0 if the configuration was successfully written
- Non-zero error code if I²C communication failed

Technical note:

- Sends a one-byte I²C write to register 0x03 (`MODE_CONFIG`)
- Reserved bits [3], [6], and [7] must always be 0
- For invalid trigger mode values (0x06, 0x07), the module defaults to Free Running
- The trigger mode can be changed at any time, even during active acquisition, and takes effect immediately

Example (Pascal):

```
var
  ucConfig: Byte;
  res: DWORD;
begin
  // Quadrature Encoder Trigger, UP direction, counter enabled
  ucConfig := $24; //%00100100 = Bit 5 = 1, Bit 4 = 0, Bits 2:0 = 100
  res := ls_setmodeconfig($FE, ucConfig);
  if res = 0 then
    ShowMessage('Trigger mode configured successfully.')
  else
    ShowMessage('Configuration failed: ' + String(ls_geterrorstring(res)));
end;
```

Notes:

- Changing the trigger mode is possible at any time and takes effect immediately
- Make sure the encoder and related parameters (e.g. `QUAD_COUNT`) are set accordingly
- This setting influences how the module reacts to trigger signals, but does not affect the data output format

SetIFConfig

Pascal Declaration:

```
function ls_setifconfig(addr, ucConfig: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt8 __stdcall ls_setifconfig(UInt8 addr, UInt8 ucConfig);
```

Description:
The `ls_setifconfig` function writes a configuration byte to the `DATA_IF_CONFIG` register (0x0D) of the LISM-PI26xx module via I²C. This register defines the timing and logic polarity of all output signals on the parallel data interface.
It controls the pixel and data clock frequencies, the polarity of synchronization signals, and the

active level of the trigger output. This allows adaptation to different external hardware requirements.

Parameters:

- `addr`: I²C slave address of the target LISM-PI26xx module
- `ucConfig`: Bit-coded interface configuration byte (see below)

Bit layout of `ucConfig`:

Bits	Function	Description
[2:0]	Clock Frequency	Selects pixel and data clock speeds:
		000 = 200 kHz / 400 kHz (Pixel / Data)
		001 = 250 kHz / 500 kHz
		010 = 400 kHz / 800 kHz
		011 = 500 kHz / 1 MHz
		100 = 1 MHz / 2 MHz
		101 = 2 MHz / 4 MHz
		110 = 5 MHz / 10 MHz
		111 = 10 MHz / 20 MHz (maximum supported speed)
[3]	Data Clock Polarity	0 = Data output on falling edge (default)
		1 = Data output on rising edge
[4]	Line Valid Polarity	0 = High-active
		1 = Low-active
[5]	Frame Valid Polarity	0 = High-active
		1 = Low-active
[6]	Trigger Output Polarity	0 = High-active
		1 = Low-active
[7]	*Reserved*	*Not used, reserved for future use*

Return Value:

- 0 if the configuration was written successfully
- Non-zero error code if the I²C write failed

Technical note:

- Affects both timing and logic levels of the output interface
- Clock frequency settings directly influence units in registers such as `START_PULSE_HIGH`.
- The function sends a single I²C write to register 0x0D (`DATA_IF_CONFIG`)
- Changes apply immediately — no module reset is required

Notes:

- Choose a clock frequency that matches the capabilities of the receiving system (e.g. FPGA, microcontroller)
- Adjust signal polarity to align with external logic expectations (active-high vs. active-low)
- After changing interface timing, review dependent timing registers to ensure consistency

SetStHigh

Pascal Declaration:

```
function ls_setsthigh(addr: Byte; dwHigh: DWORD): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_setsthigh(UInt8 addr, UInt32 dwHigh);
```

Description:
The `ls_setsthigh` function sets the high-time duration of the START pulse in the LISM-PI26xx module by writing a 32-bit value to the `START_PULSE_HIGH` register (0x04) via I²C.
This value defines the integration window — the period during which the sensor collects light before readout begins. The actual integration time depends on the pixel clock frequency (from `DATA_IF_CONFIG`) and a sensor-specific offset.

Parameters:

- `addr`: I²C address of the module
- `dwHigh`: Pulse high duration in clock cycles (32-bit unsigned)

Return Value:

- 0 on success
- Non-zero error code if I²C communication failed

Timing resolution:

The time per clock cycle depends on the `DATA_IF_CONFIG` setting (bits [2:0]):

Setting	Pixel Clock	Cycle Duration
000	200 kHz	5.0 µs
001	250 kHz	4.0 µs
010	400 kHz	2.5 µs
011	500 kHz	2.0 µs
100	1 MHz	1.0 µs

Setting	Pixel Clock	Cycle Duration
101	2 MHz	0.5 µs
110	5 MHz	0.2 µs
111	10 MHz	0.1 µs

Effective integration time:

```
T_integration = (dwHigh + offset) × cycle_duration
```

For example, with sensor **S11639-01**, the offset is **48 clock cycles**, so the total integration time is:

```
T_integration = (dwHigh + 48) × cycle_duration
```

Technical note:

- Sends a 32-bit I²C write to register 0x04 (`START_PULSE_HIGH`)
- Value is transmitted in big-endian order (MSB first)
- This function sets only the high phase of the START pulse
- To define the full cycle, use `ls_setstlow` for the low phase

Notes:

- Used to control the sensor's exposure time
- Should be tuned depending on ambient light and required frame rate
- Effective immediately — no restart or update command required

SetStLow

Pascal Declaration:

```
function ls_setstlow(addr: Byte; dwLow: DWORD): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_setstlow(UInt8 addr, UInt32 dwLow);
```

Description:
The `ls_setstlow` function writes a 32-bit value to the `START_PULSE_LOW` register (0x05) of the LISM-PI26xx module via I²C. This value defines the duration of the low phase of the START pulse, i.e. the time between the end of integration and the beginning of the next acquisition cycle.

While `START_PULSE_HIGH` defines the integration time (exposure), `START_PULSE_LOW` determines the duration of the idle phase. The total line cycle time is:

```
T_cycle = (START_PULSE_HIGH + START_PULSE_LOW) × T_clk
Line Rate = 1 / T_cycle
```

where `T_clk` is the pixel clock period defined in `DATA_IF_CONFIG`.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `dwLow`: Duration of the low phase of the START pulse, in clock cycles (32-bit unsigned)

Return Value:

- 0 if successful
- Non-zero error code if the I²C write fails

Timing resolution:

The actual time per cycle depends on the pixel clock frequency set via `DATA_IF_CONFIG` (bits [2:0]):

Setting	Pixel Clock	Cycle Duration
000	200 kHz	5.0 µs
001	250 kHz	4.0 µs
010	400 kHz	2.5 µs
011	500 kHz	2.0 µs
100	1 MHz	1.0 µs
101	2 MHz	0.5 µs
110	5 MHz	0.2 µs
111	10 MHz	0.1 µs

Example:
At a pixel clock of **10 MHz**, a value of `dwLow = 1000` results in:

```
T_low = 1000 × 0.1 µs = 100 µs
```

Technical note:

- Sends a 32-bit I²C write to register 0x05 (`START_PULSE_LOW`)
- Value is transmitted in big-endian order (MSB first)
- The total cycle duration must fulfill the sensor’s timing requirements
- Example for sensor **S11639-01** (2048 pixels):

```
ST(Period)_min =
    2048   // number of pixels
  +   89   // delay before valid data output (88 + 1)
  +    9   // ADC output latency
  +    5   // FRAME_VALID trailing margin
  = 2151 clock cycles
```

- To ensure valid timing:

  `START_PULSE_LOW ≥ 2151 - START_PULSE_HIGH`

Notes:

- This function does not affect exposure, but it defines the readout and idle phase
- Pixel data output typically starts during the **low phase** and may extend into the **high phase**, depending on the configured timing and sensor behavior
- Always ensure that the total cycle duration allows complete data output

SetStPulse

Pascal Declaration:

```
function ls_setstpulse(addr: Byte; dwHigh, dwLow: DWORD): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_setstpulse( UInt8 addr, UInt32 dwHigh, UInt32 dwLow);
```

Description:
The `ls_setstpulse` function sets the complete START pulse timing on the LISM-PI26xx module by writing both the high and low durations of the pulse in a single I²C operation.
This defines the full acquisition cycle consisting of:

- **High phase (`dwHigh`)**: the integration window during which the sensor collects light
- **Low phase (`dwLow`)**: the idle period before the next integration begins

This function combines the effects of `ls_setsthigh` and `ls_setstlow` into one atomic operation, ensuring consistent and synchronized timing.

Parameters:

- `addr`: I²C slave address of the target LISM-PI26xx module
- `dwHigh`: Duration of the high phase (integration), in clock cycles
- `dwLow`: Duration of the low phase (idle), in clock cycles

Return Value:

- 0 if the I²C write was successful
- Non-zero error code if the communication failed

Timing principle:

- `T_cycle = (dwHigh + dwLow) × T_clk`
  `Line Rate = 1 / T_cycle`
  where `T_clk` is the pixel clock period determined by `DATA_IF_CONFIG`

Technical note:

- Performs a **single 8-byte I²C write** starting at register 0x04
- Data format:

```
Register 0x04 → dwHigh (4 bytes, MSB first)
                dwLow  (4 bytes, MSB first)
```

- The combined transfer ensures that both pulse phases are updated simultaneously and consistently
- Changes apply immediately

Notes:

- Ensures synchronized update of integration and cycle timing
- Recommended in real-time systems or when switching between timing profiles
- The total cycle duration (`dwHigh + dwLow`) must fulfill sensor-specific timing constraints (e.g. ≥ 2151 clock cycles for S11639-01 to ensure full pixel readout)

SetLinesPerFrame

Pascal Declaration:

```
function ls_setlinesperframe(addr: Byte; wLines: WORD): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_setlinesperframe( UInt8 addr, UInt16 wLines);
```

Description:
The `ls_setlinesperframe` function configures how many individual line acquisitions the LISM-PI26xx module groups into one complete image frame. It writes a 16-bit value to the `LINES_PER_FRAME` register (0x06) via I²C.
This setting determines the duration and structure of the `FRAME_VALID` signal, which indicates the boundaries of a full frame in multi-line acquisition modes.

Parameters:

- `addr`: I²C address of the LISM-PI26xx module

- `wLines`: Number of lines per frame (valid range: 1 to 65535)

Return Value:

- 0 on success
- Non-zero error code on I²C communication failure

Behavior:

- During acquisition, the `LINE_VALID` signal is asserted once per line
- The `FRAME_VALID` signal goes high **10 data clock cycles before** the first `LINE_VALID`, and goes low **10 data cycles after** the last
- The number of `LINE_VALID` pulses per frame equals `wLines`

Technical note:

- Writes a 16-bit value to register 0x06 (`LINES_PER_FRAME`)
- Data is sent in big-endian order (MSB first)
- This setting has no effect in single-line trigger modes
  (unless used intentionally with `wLines = 1`)
- Internally, frame timing logic is synchronized to this value to control `FRAME_VALID`

Notes:

- Required for:
  - Multi-line acquisition with external trigger (`MODE_CONFIG = 0x05`)
  - Frame construction using internal software trigger
  - Proper alignment of `FRAME_VALID` for downstream frame-based systems
- Use with care in encoder-triggered or continuous modes to avoid desynchronization
- Changing this value during acquisition may affect the current frame timing

Examples:

- `wLines = 1` → Each frame contains one line (single-line mode)
- `wLines = 1024` → Frame consists of 1024 lines

SetQuadCount

Pascal Declaration:

```
function ls_setquadcount(addr: Byte; dwCount: DWORD): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_setquadcount(UInt8 addr, UInt32 dwCount);
```

Description:
The `ls_setquadcount` function writes a 32-bit value to the `QUAD_COUNT` register (0x07) of the LISM-PI26xx module via I²C. This value specifies how many valid quadrature encoder steps must occur before a new image line is acquired.
This setting is only relevant when the acquisition mode is configured for **Quadrature Encoder Trigger** (`MODE_CONFIG = 0x04`) and the **encoder counter is enabled** (`MODE_CONFIG` Bit 5 = 1).

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `dwCount`: Number of encoder steps required to trigger one image line

Return Value:

- 0 if the command was successful
- Non-zero error code on I²C communication failure

Behavior:

- The encoder interface monitors rising and falling edges on input channels A/B
- If the counter is **disabled**, each encoder step immediately triggers a line
- If the counter is **enabled**, a line is triggered only after `dwCount` steps
- After a trigger, the counter resets and begins counting for the next line

Technical note:

- Writes a 32-bit value to register 0x07 (`QUAD_COUNT`)
- Value is transmitted in big-endian order (MSB first)
- Only active in `MODE_CONFIG = 0x04` with Bit 5 set
- The encoder direction (CW or CCW) is controlled via `MODE_CONFIG` Bit 4
- Used in motion-synchronized acquisition scenarios (e.g., linear travel or rotation)

Notes:

- Works in conjunction with `LINES_PER_FRAME` to group encoder-based lines into frames
- Use appropriate `DATA_IF_CONFIG` settings for timing alignment
- Typical in conveyor belt imaging or rotary scanning systems
- If `dwCount = 1`, behavior is equivalent to direct triggering on each encoder event

Example:
If `dwCount = 100`, a new line will be captured **after every 100 encoder steps**, based on direction and counting mode from `MODE_CONFIG`.

SetSoftTriggerTime

Pascal Declaration:

```
function ls_setsofttriggertime(addr: Byte; dwTime: DWORD): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_setsofttriggertime(UInt8 addr, UInt32 dwTime);
```

Description:
The `ls_setsofttriggertime` function writes a 32-bit value to the `SOFT_TRIGGER_PERIOD` register (0x08) of the LISM-PI26xx module via I²C. This value defines the time interval — in microseconds — at which the internal software trigger generates new acquisition events.
This function is only effective when the acquisition mode is set to **Software Trigger** (`MODE_CONFIG = 0x03`).

Parameters:

- `addr`: I²C address of the LISM-PI26xx module
- `dwTime`: Trigger period in microseconds (must be ≥ total START pulse duration)

Return Value:

- 0 if the value was successfully written
- Non-zero error code if the I²C operation failed

Timing behavior:

- The internal timing generator issues a new START pulse every `dwTime` microseconds
- The line rate is defined by:

  ```
  line_rate = 1 / dwTime [in seconds]
  ```

For example: `dwTime = 500 → line_rate = 2000 lines/sec`

- Time unit is fixed: **1 µs per step**
- Timing resolution is independent of pixel clock configuration

Technical note:

- Writes a 32-bit value to register 0x08 (`SOFT_TRIGGER_PERIOD`)
- Value is sent in big-endian byte order (MSB first)
- Effective only when `MODE_CONFIG = 0x03`
- The generator internally compares elapsed time to this value to issue new triggers

Notes:

- `dwTime` must be **strictly greater** than `START_PULSE_HIGH + START_PULSE_LOW`, otherwise trigger intervals will overlap
- Can be used together with `LINES_PER_FRAME` to define complete frame timing
- Useful for precise timing control when no external trigger source is available
- Changes apply immediately

Example:
To set a software trigger interval of **500 µs** (→ 2000 lines per second):

```
res := ls_setsofttriggertime($FE, 500);
```

SetTriggerDelay

Pascal Declaration:

```
function ls_settriggerdelay(addr: Byte; dwDelay: DWORD): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_settriggerdelay(UInt8 addr, UInt32 dwDelay);
```

Description:
The `ls_settriggerdelay` function writes a 32-bit value to the `TRIGGER_OUTPUT_DELAY` register (0x09) of the LISM-PI26xx module via I²C. This value defines the delay time — in microseconds — between the start of integration (beginning of the START pulse) and the activation of the trigger output signal.
This feature is typically used to synchronize external devices (e.g. light sources or data capture systems) with a defined point in the sensor’s exposure cycle.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `dwDelay`: Time delay in microseconds between integration start and trigger output activation

Return Value:

- 0 on success
- Non-zero error code if the I²C write fails

Timing behavior:

- Delay is measured from the leading edge of the START pulse
- Timing resolution is fixed at **1 µs**
- If `dwDelay = 0`, the trigger output is asserted **immediately** when integration begins

Technical note:

- Writes a 32-bit value to register 0x09 (`TRIGGER_OUTPUT_DELAY`)
- Value is transmitted in big-endian format (MSB first)
- The rising edge of the trigger output is delayed by `dwDelay` after integration start
- This setting works independently of acquisition mode

Notes:

- The **duration** of the trigger output is set separately via `ls_settriggerwidth`
- Make sure that:

  `dwDelay + trigger_width ≤ START_PULSE_HIGH + START_PULSE_LOW`

  to avoid trigger overlap with the next cycle
- The polarity of the trigger output signal is controlled via `DATA_IF_CONFIG` (bit 6)

SetTriggerWidth

Pascal Declaration:

```
function ls_settriggerwidth(addr: Byte; dwWidth: DWORD): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_settriggerwidth(UInt8 addr, UInt32 dwWidth);
```

Description:
The `ls_settriggerwidth` function writes a 32-bit value to the `TRIGGER_OUTPUT_WIDTH` register (0x0A) of the LISM-PI26xx module via I²C. This value defines the active duration — in microseconds — of the trigger output signal, starting after the delay configured by `ls_settriggerdelay`.
It allows precise pulse-width control to synchronize external hardware components such as light sources, cameras, or logic devices.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `dwWidth`: Pulse width in microseconds (active duration of trigger output)

Return Value:

- 0 if the write was successful
- Non-zero error code if I²C communication failed

Timing behavior:

- The trigger output becomes active after the time defined by `TRIGGER_OUTPUT_DELAY`
- It stays active for `dwWidth` microseconds
- If the **delay + width** exceeds the total START pulse duration, the pulse is automatically cut off at the start of the next cycle

Technical note:

- Writes a 32-bit value to register 0x0A (`TRIGGER_OUTPUT_WIDTH`)
- Value is transmitted in big-endian byte order (MSB first)
- Trigger pulse generation is relative to each START pulse, repeated line-by-line

Notes:

- Ensure:

  ```
  TRIGGER_OUTPUT_DELAY + TRIGGER_OUTPUT_WIDTH ≤ START_PULSE_HIGH + START_PULSE_LOW
  ```

  to prevent incomplete trigger pulses
- The **trigger output polarity** is configured via bit 6 in `DATA_IF_CONFIG`
- This setting works in all acquisition modes and takes effect immediately

SetPixelCount

Pascal Declaration:

```
function ls_setpixelcount(addr: Byte; wCount: WORD): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_setpixelcount(UInt8 addr, UInt16 wCount);
```

Description:
The `ls_setpixelcount` function writes a 16-bit value to the `PIXEL_COUNT` register (0x0B) of the LISM-PI26xx module via I²C. This value defines how many pixels will be read out and output per acquisition line. It enables dynamic configuration of line length based on sensor resolution or region-of-interest requirements.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `wCount`: Number of pixels per line (e.g. 2048 for full-width on S11639-01)

Return Value:

- 0 on success
- Non-zero error code if the I²C command fails

Behavior:

- The module outputs exactly `wCount` pixels per valid line
- The `LINE_VALID` signal remains high for the duration of those pixels
- If `wCount` is less than the physical sensor resolution, output is cropped accordingly
- The configured START pulse duration (`START_PULSE_HIGH + LOW`) must be long enough to accommodate all pixels

Technical note:

- Writes a 16-bit value to register 0x0B (`PIXEL_COUNT`)
- Value is transmitted in big-endian order (MSB first)
- **The LISM-PI26xx module applies the new value immediately**
- **However, the USB controller must be restarted to apply the new value in its internal data streaming logic** — otherwise, host-side transfer sizes may not match

Notes:

- Ensure the readout timing (START pulse) is long enough for `wCount` pixels
- Use together with `EDGES_BEFORE_DATA` to apply horizontal offset
- Pixel format is always 16-bit: **MSB first, then LSB**, transmitted over the parallel interface
- This setting is crucial when switching between sensors with different resolutions or when reducing data bandwidth

SetEdgeDelay

Pascal Declaration:

```
function ls_setedgedelay(addr: Byte; ucEdges: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_setedgedelay(UInt8 addr, UInt8 ucEdges);
```

Description:
The `ls_setedgedelay` function writes an 8-bit value to the `EDGES_BEFORE_DATA` register (0x0C) of the LISM-PI26xx module via I²C. This value defines how many pixel clock edges to wait after the falling edge of the START pulse before the output of pixel data begins.
It allows the timing of pixel readout to be aligned with the valid data window of the connected linear image sensor.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `ucEdges`: Number of pixel clock edges to delay data output (1–255)

Return Value:

- 0 on success
- Non-zero error code if the I²C command fails

Behavior:

- After the falling edge of the START pulse, the system waits for `ucEdges` falling pixel clock edges
- After this delay, the `LINE_VALID` signal is asserted and pixel output begins
- This mechanism ensures that only valid sensor data is transmitted

Technical note:

- Writes an 8-bit value to register 0x0C (`EDGES_BEFORE_DATA`)
- Value is applied immediately and affects every line
- **For the Hamamatsu S11639-01 sensor**:
    - Valid data begins on the **89th rising clock edge** after START
    - This corresponds to `88` pixel clock cycles of internal sensor delay
    - The LISM-PI26xx adds **9 additional ADC delay cycles** internally
    - Therefore, `LINE_VALID` is asserted **after 98 pixel clock cycles total**
- To match this timing, set `ucEdges = 88` → remaining delay is added automatically

Notes:

- Must be coordinated with `PIXEL_COUNT` and the total `START_PULSE` duration
- The time per edge is determined by the pixel clock (see `DATA_IF_CONFIG`)
- `LINE_VALID` remains high during the output of all `PIXEL_COUNT` pixels
- Cropping of the beginning of a sensor line is possible via this setting

Example:

```
// For Hamamatsu S11639-01 sensor
res := ls_setedgedelay($FE, 88); // Starts pixel output after 98 clocks total
```

SetADCVref

Pascal Declaration:

```
function ls_setadcvref(addr, ucvref: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_setadcvref(UInt8 addr, UInt8 ucvref);
```

Description:
The `ls_setadcvref` function writes an 8-bit value to the `ADC_FULL_SCALE` register (0x20) of the LISM-PI26xx module via I²C. This value selects the analog input voltage range that corresponds to the full 16-bit digital output scale of the ADC.
Changing this range affects the mapping of the sensor signal to the 16-bit output (0…65535), and influences sensitivity, dynamic range, and resolution.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `ucvref`: Selection of ADC full-scale voltage (must be 0x00 or 0x01)

Return Value:

- 0 if the command was accepted
- Non-zero error code on I²C failure

Valid values:

ucvref	ADC Input Range	Description
0x00	1.6 V	Higher resolution, reduced dynamic range
0x01	2.0 V	Wider input range, lower sensitivity

Technical note:

- Writes a single byte to register 0x20 (`ADC_FULL_SCALE`)
- Value is applied immediately and does not affect offset or gain
- Select a range that best matches the analog signal level from the sensor
- ADC resolution is always 16-bit; this setting scales the input window

Notes:

- Defines the voltage range that corresponds to digital values 0…65535

- Can be combined with `ls_setadcgain`, `ls_setadcoffset`, or `ls_setadcextref` for calibration
- Changing this setting does **not** affect polarity, digital format, or data output method

Use cases:

- Match sensor analog output to the ADC input range
- Improve signal-to-noise ratio (SNR) under low-light or high-contrast conditions
- Tune ADC behavior based on external circuitry or gain stages

SetADCGain

Pascal Declaration:

```
function ls_setadcgain(addr, ucGain: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_setadcgain(UInt8 addr, UInt8 ucGain);
```

Description:
The `ls_setadcgain` function writes an 8-bit value to the `ADC_GAIN` register (0x21) of the LISM-PI26xx module via I²C. This value sets the analog gain applied to the sensor signal before digitization. It allows amplification of weak signals to make better use of the ADC's full-scale range.

The gain is adjustable in **64 steps**, from **1.0× to 5.85×**, with logarithmic spacing. The gain value is encoded as a 6-bit unsigned integer (`G = 0…63`) and mapped to actual voltage gain via a fixed formula.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `ucGain`: 6-bit gain code (range: 0–63); bits 7–6 must be 0

Return Value:

- 0 on success
- Non-zero error code if the I²C write fails

Gain encoding:

Bits	Function
D7–D6	Reserved, must be 0
D5–D0	Gain code G (value 0 to 63)

Gain formula:

```
Gain = 76 / (76 − G)
```

Gain reference values:

G	Gain (V/V)	Gain (dB)
0	1.00	0.00 dB
31	~2.45	~7.77 dB
63	5.85	~15.34 dB

Technical note:

- Writes a single byte to register 0x21 (`ADC_GAIN`)
- Gain is applied **before digitization**
- Increasing gain raises signal amplitude, but also amplifies analog noise
- Default value is typically 0 (1.0×)

Notes:

- Affects analog signal path — not digital image scaling
- For best results, combine with `ADC_OFFSET` and `ADC_FULL_SCALE`
- May be changed dynamically during calibration or auto-exposure control
- Does not clip or compress the ADC output range — clipping only occurs if the signal exceeds full-scale

Use cases:

- Boost weak signals from low-light or short exposure situations
- Adapt analog input range to match ADC input window
- Enable dynamic gain control in embedded imaging systems

SetADCOffset

Pascal Declaration:

```
function ls_setadcoffset(addr: Byte; wOffset: Word): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_setadcoffset(UInt8 addr, UInt16 wOffset);
```

Description:
The `ls_setadcoffset` function writes a 16-bit value to the `ADC_OFFSET` register (0x22) of the LISM-PI26xx module via I²C. This offset is added to the analog sensor signal before digitization and allows shifting the signal baseline to optimize usage of the ADC input range.

This feature is useful to correct for dark signal levels, sensor-specific bias, or system-wide offsets that would otherwise waste part of the ADC's dynamic range.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `wOffset`: Signed offset value, using sign-magnitude encoding (see below)

Return Value:

- 0 on success
- Non-zero error code if the I²C command fails

Encoding format:

- **Sign-magnitude, 9-bit value within 16 bits**
    - Bit 8 = sign bit: 0 = positive, 1 = negative
    - Bits 7–0 = magnitude (0…255)
- The offset range is ±250 mV in 512 steps (approx. 0.98 mV/step)

Example encoding table:

Value (`wOffset`)	Offset (mV)
0x000	+0.00
0x001	+0.98
0x0FF	+250.00
0x100	−0.00
0x1FF	−250.00

Technical note:

- Writes a 16-bit value to register 0x22 (`ADC_OFFSET`)
- Value is sent in big-endian order (MSB first)
- The offset is applied in the **analog signal path**, not digitally
- Has immediate effect on ADC input voltage level

Notes:

- Particularly useful when the sensor has a dark level that is not zero
- Should be adjusted together with `ADC_GAIN` and `ADC_FULL_SCALE`
- May require individual calibration per sensor or application

Use cases:

- Correct sensor-internal black level offset
- Maximize effective ADC resolution
- Compensate for thermal drift or analog input bias

SetADCBias

Pascal Declaration:

```
function ls_setadcbias(addr: Byte; wBias: Word): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_setadcbias(UInt8 addr, UInt16 wBias);
```

Description:
The `ls_setadcbias` function writes a 16-bit value to the `ADC_INPUT_BIAS` register (0x30) of the LISM-PI26xx module via I²C.

This register sets the output of an internal DAC, which acts as an analog input bias voltage. This voltage is subtracted from the sensor signal prior to amplification and digitization. It serves as a coarse correction for sensor signals with elevated DC levels or baseline offsets.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `wBias`: DAC value (range: 0…1023), corresponding to an output voltage from 0 V to 2.5 V

Return Value:

- 0 if the value was written successfully
- Non-zero error code on I²C failure

Voltage calculation:

wBias	Output Voltage	Step Size
0	0.000 V	
512	~1.250 V	
1023	~2.500 V	≈ 2.44 mV/step

Formula: `V_bias = (wBias / 1023) × 2.5 V`

Technical note:

- Writes a 16-bit value to register 0x30 (`ADC_INPUT_BIAS`)
- Value is sent in big-endian order (MSB first)
- The DAC-generated voltage is subtracted from the analog input signal before gain and digitization
- Effective immediately — no restart required

Notes:

- Used for **coarse alignment** of the analog baseline
- Should be set **before** adjusting `ADC_GAIN` or `ADC_OFFSET`
- Helps prevent clipping and ensures optimal use of the ADC range by aligning the signal baseline close to zero

Use cases:

- Compensate for sensor black level or output offset
- Shift input signal range to match the ADC’s usable input span
- Combine with `ls_setadcoffset` for fine-grained black level correction

GetCTGState

Pascal Declaration:

```
function ls_getctgstate(addr: Byte; var ucState: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getctgstate(UInt8 addr, UInt8* ucState);
```

Description:
The `ls_getctgstate` function reads the current state of the LISM-PI26xx module's internal clock and timing generator from the `RESUME_GENERATOR` register (0x00) via I²C.
It allows the application to determine whether the timing generator is currently **active (resumed)** or **suspended**.

Parameters:

- `addr`: I²C slave address of the target LISM-PI26xx module
- `ucState`: Output variable that receives the raw state byte from the register

Return Value:

- 0 on success
- Non-zero error code if the I²C read fails

Behavior:

`ucState` value	Meaning
0xAA	Generator is **active** (resumed)
any other value	Generator is **suspended** (inactive)

Technical note:

- Reads a single byte from register 0x00 (`RESUME_GENERATOR`)
- The register is writable (to control state) and readable (to check current state)
- Reading does **not** change the generator state

Notes:

- Useful for verifying whether the generator was started (`ls_resume`) or stopped (`ls_suspend`)
- When suspended, no acquisition activity (e.g. `START`, `FRAME_VALID`, or trigger signals) occurs
- This function provides non-intrusive monitoring of the generator's status

Use cases:

- Confirm correct state before issuing acquisition commands
- Debug acquisition issues due to inactive generator
- Display current system state in a status monitor or UI

GetState

Pascal Declaration:

```
function ls_getstate(addr: Byte; var ucState: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getstate(UInt8 addr, UInt8* ucState);
```

Description:
The `ls_getstate` function queries the current acquisition state of the LISM-PI26xx module via I²C by reading the value of the `START_ACQUISITION` register (0x02).
It allows the host system to determine whether the module is actively acquiring image lines or is currently idle.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `ucState`: Output variable that receives the current acquisition state (0x00 or 0x01)

Return Value:

- 0 if the read was successful
- Non-zero error code if I²C communication failed

Behavior:

`ucState` value	Meaning
0x00	Acquisition stopped
0x01	Acquisition running

Technical note:

- Reads a single byte from register 0x02 (`START_ACQUISITION`)
- This reflects the currently active state of the timing generator and trigger system
- Reading does **not** affect acquisition behavior

Notes:

- This function complements `ls_setstate`, which starts or stops acquisition
- Acquisition must be active for any image data, timing signals, or trigger outputs to be generated
- This function only reports ON/OFF status — it does **not** indicate errors or faults
- Can be polled regularly in systems that lack interrupt or event notification

Use cases:

- Verify whether acquisition has started correctly
- Detect whether acquisition was stopped (e.g. manually or due to external control)
- Integrate status polling into GUIs or embedded control loops
- Use in test/debug tools to monitor sensor readiness

GetModeConfig

Pascal Declaration:

```
function ls_getmodeconfig(addr: Byte; var ucConfig: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getmodeconfig(UInt8 addr, UInt8* ucConfig);
```

Description:
The `ls_getmodeconfig` function reads the 8-bit value from the `MODE_CONFIG` register (0x03) of the LISM-PI26xx module via I²C. This register defines the currently active trigger mode and quadrature encoder configuration used for acquisition.

By reading this register, the host can determine which trigger source is selected, the encoder direction mode, and whether encoder-based counting is enabled.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `ucConfig`: Output variable that receives the 8-bit mode configuration

Return Value:

- 0 on success
- Non-zero error code if the I²C read fails

Bit layout of `ucConfig`:

Bits	Function	Value(s)	Description
[2:0]	Trigger Mode Selection	000	Free Running
		001	External Rising Edge (1 line per pulse)
		010	External High Level (continuous while high)
		011	Software Trigger (uses SOFT_TRIGGER_PERIOD)
		100	Quadrature Encoder
		101	External Rising Edge (multi-line)
[3]	Reserved	Must be 0	Reserved bit – always write/read as zero
[4]	Encoder Direction Mode	0 = UP (CW)	Clockwise (increment)
		1 = DOWN (CCW)	Counter-clockwise (decrement)
[5]	Encoder Counter Enable	0 = Disabled	Trigger on every encoder step
		1 = Enabled	Trigger only after QUAD_COUNT steps
[6–7]	Reserved	Must be 0	Reserved bits – always write/read as zero

Technical note:

- Reads one byte from register 0x03 (`MODE_CONFIG`)
- Bits 3, 6, and 7 are reserved and should be masked when evaluating
- Values correspond directly to those written via `ls_setmodeconfig`

Notes:

- This function complements `ls_setmodeconfig`
- Only valid bit combinations should be interpreted
- Useful for runtime checks or debugging mode selection

Use cases:

- Confirm active trigger configuration
- Monitor encoder trigger settings during motion acquisition
- Auto-detect mode in adaptive acquisition software

GetIFConfig

Pascal Declaration:

```
function ls_getifconfig(addr: Byte; var ucConfig: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getifconfig(UInt8 addr, UInt8* ucConfig);
```

Description:
The `ls_getifconfig` function reads the 8-bit value from the `DATA_IF_CONFIG` register (0x0D) of the LISM-PI26xx module via I²C. This register controls how image data and synchronization signals are transmitted on the parallel interface — including pixel / data clock frequency, clock polarity, and active signal levels.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `ucConfig`: Output variable that receives the 8-bit interface configuration

Return Value:

- 0 on success
- Non-zero error code if the I²C read fails

Bit layout of `ucConfig`:

Bits	Function	Value(s)	Description
[2:0]	Clock Frequency Select	000–111	Selects pixel/data clock rate (see table below)
[3]	Data Clock Polarity	0 = falling edge, 1 = rising	Selects edge for data output timing
[4]	LINE_VALID Polarity	0 = high-active, 1 = low	Active level for line validity signal
[5]	FRAME_VALID Polarity	0 = high-active, 1 = low	Active level for frame validity signal
[6]	TRIG_OUT Polarity	0 = high-active, 1 = low	Active level for trigger output signal
[7]	Reserved	Always 0	Reserved — must be ignored on read

Clock frequency mapping (bits [2:0]):

Setting	Pixel Clock	Data Clock
000	200 kHz	400 kHz
001	250 kHz	500 kHz
010	400 kHz	800 kHz
011	500 kHz	1.0 MHz

Setting	Pixel Clock	Data Clock
100	1.0 MHz	2.0 MHz
101	2.0 MHz	4.0 MHz
110	5.0 MHz	10.0 MHz
111	10.0 MHz	20.0 MHz

Technical note:

- Reads one byte from register 0x0D (`DATA_IF_CONFIG`)
- Each bit controls an aspect of the data interface output timing
- Corresponds directly to values set via `ls_setifconfig`

Notes:

- Use this function to confirm or validate interface settings at runtime
- Signal polarity settings must match connected hardware logic levels
- The pixel clock and signal timing affect integration time calculations

Use cases:

- Confirm signal edge selection for external hardware (FPGA, microcontroller)
- Debug incorrect signal interpretation or data alignment issues
- Detect startup configuration in self-adaptive acquisition systems

GetStHigh

Pascal Declaration:

```
function ls_getsthigh(addr: Byte; var dwHigh: DWORD): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getsthigh(UInt8 addr, UInt32* dwHigh);
```

Description:
The `ls_getsthigh` function reads a 32-bit value from the `START_PULSE_HIGH` register (0x04) of the LISM-PI26xx module via I²C. This value defines the duration of the high phase of the START pulse — i.e., the integration period in clock cycles during which the sensor collects light.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module

- `dwHigh`: Output variable that receives the high-phase duration (in clock cycles)

Return Value:

- 0 if the value was read successfully
- Non-zero error code if the I²C read fails

Behavior:

- Reads 4 bytes from register 0x04 (`START_PULSE_HIGH`)
- The value represents the number of pixel clock cycles the START signal remains high
- This defines the integration time base for one line acquisition

Technical note:

- Value is read in big-endian byte order (MSB first)
- Use together with `ls_getstlow` or `ls_getstpulse` to reconstruct the full acquisition cycle
- Integration time depends on clock frequency set in `DATA_IF_CONFIG`:

  ```
  T_integration = (dwHigh + offset) × T_clk
  ```

Notes:

- The pixel clock cycle duration (`T_clk`) is determined by `DATA_IF_CONFIG` (bits [2:0])
- The integration time may need to be increased for low-light or high-sensitivity applications
- When using software triggering, ensure:

  ```
  SOFT_TRIGGER_PERIOD > START_PULSE_HIGH + START_PULSE_LOW
  ```

Use cases:

- Display or log sensor exposure settings
- Verify sensor configuration during runtime diagnostics
- Support timing calculations for adaptive acquisition or trigger systems

GetStLow

Pascal Declaration:

```
function ls_getstlow(addr: Byte; var dwLow: DWORD): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getstlow(UInt8 addr, UInt32* dwLow);
```

Description:
The `ls_getstlow` function reads a 32-bit value from the `START_PULSE_LOW` register (0x05) of the LISM-PI26xx module via I²C. This value specifies the duration of the low phase of the START pulse, which begins after the integration phase and includes sensor readout and idle time before the next acquisition cycle.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `dwLow`: Output variable that receives the low-phase duration (in clock cycles)

Return Value:

- 0 on success
- Non-zero error code if the I²C read fails

Behavior:

- Reads 4 bytes from register 0x05 (`START_PULSE_LOW`)
- The value represents the number of pixel clock cycles the START signal remains low
- It is used to define the total line cycle time along with the high phase:

  `T_cycle = (START_PULSE_HIGH + START_PULSE_LOW) × T_clk`

Technical note:

- Value is read in big-endian byte order (MSB first)
- Must be long enough to accommodate sensor readout and inter-line delay
- Works in all trigger modes and directly affects line rate

Notes:

- The low phase must be sufficient to allow transmission of all `PIXEL_COUNT` pixels
- Actual time is determined by the clock frequency set via `DATA_IF_CONFIG`
- This setting does not affect integration but controls readout and idle margin

Use cases:

- Retrieve complete timing configuration for acquisition cycles
- Calculate and validate frame rate and soft trigger periods
- Diagnose performance issues related to line timing or sensor sync

GetStPulse

Pascal Declaration:

```
function ls_getstpulse(addr: Byte; var dwHigh, dwLow: DWORD): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt __stdcall ls_getstpulse(UInt8 addr, UInt32* dwHigh, UInt32* dwLow);
```

Description:
The `ls_getstpulse` function reads two 32-bit values from the LISM-PI26xx module via I²C that define the full START pulse timing.
It retrieves the durations of both the **high phase** (integration) and the **low phase** (idle) that make up the acquisition cycle.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `dwHigh`: Output variable that receives the high-phase duration (clock cycles)
- `dwLow`: Output variable that receives the low-phase duration (clock cycles)

Return Value:

- 0 on success
- Non-zero error code if either I²C read fails

Behavior:

- Reads two consecutive 32-bit values:
    - Register 0x04 → `START_PULSE_HIGH`
    - Register 0x05 → `START_PULSE_LOW`
- `dwHigh`: defines the integration time base (pulse high duration)
- `dwLow`: defines readout and idle time between lines
- Together they form the full cycle: `T_cycle = (dwHigh + dwLow) × T_clk`

Technical note:

- Values are returned in big-endian order (MSB first)
- Timing resolution depends on pixel clock frequency (set via `DATA_IF_CONFIG`)
- Values read here should match what was written via `ls_setstpulse`, `ls_setsthigh`, or `ls_setstlow`
- Used to reconstruct timing for diagnostics or adaptive acquisition

Notes:

- The total line rate is: `Line Rate = 1 / T_cycle`
- In software-triggered mode, compare this with the trigger period from `ls_getsofttriggertime`
- Ensure the total cycle time meets sensor timing requirements (e.g. ≥ 2151 clocks for S11639-01)

Use cases:

- Monitor or verify sensor timing configuration
- Visualize integration vs. readout ratio in real-time systems
- Log timing for adaptive control, tuning, or calibration

GetLinesPerFrame

Pascal Declaration:

```
function ls_getlinesperframe(addr: Byte; var wLines: Word): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getlinesperframe(UInt8 addr, UInt16* wLines);
```

Description:
The `ls_getlinesperframe` function reads a 16-bit value from the `LINES_PER_FRAME` register (0x06) of the LISM-PI26xx module via I²C. This value specifies how many lines are grouped together to form a complete image frame during acquisition.

The setting affects the timing of the `FRAME_VALID` signal, which is asserted before the first line and deasserted after the last line of each frame.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `wLines`: Output variable to receive the number of lines per frame

Return Value:

- 0 if the read was successful
- Non-zero error code if the I²C access failed

Behavior:

- Reads 2 bytes from register 0x06 (`LINES_PER_FRAME`)
- Value is transmitted in big-endian format (MSB first)
- The `FRAME_VALID` signal:
    - Goes high **10 data clock cycles before** the first `LINE_VALID` pulse
    - Goes low **10 data clock cycles after** the last line in the frame

Notes:

- Used in multi-line acquisition modes such as software or encoder-triggered frame construction
- Must match host or external device expectations for frame size
- In modes where `LINES_PER_FRAME = 1`, each line forms an individual frame

Use cases:

- Determine the active frame structure setting
- Visualize or monitor number of lines per frame during runtime
- Synchronize external systems (e.g. strobes, data loggers) to frame timing

GetQuadCount

Pascal Declaration:

```
function ls_getquadcount(addr: Byte; var dwCount: DWORD): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getquadcount(UInt8 addr, UInt32* dwCount);
```

Description:
The `ls_getquadcount` function reads a 32-bit value from the `QUAD_COUNT` register (0x07) of the LISM-PI26xx module via I²C. This register defines how many quadrature encoder events must occur before the module triggers the acquisition of one image line — but only if **Quadrature Encoder Trigger mode** is active **and** the **counter is enabled**.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `dwCount`: Output variable to receive the configured encoder event count

Return Value:

- 0 if successful
- Non-zero error code if the I²C read fails

Behavior:

- Reads 4 bytes from register 0x07 (`QUAD_COUNT`)
- The value defines the number of valid encoder steps (A/B transitions) required before one line acquisition is triggered
- The counter resets after each successful trigger

Notes:

- This value is only evaluated if:
    - Trigger mode = Quadrature Encoder (`MODE_CONFIG[2:0] = 0x04`)
    - Encoder counter is enabled (`MODE_CONFIG[5] = 1`)
- If the counter is disabled, **every encoder step** triggers a line and this register is ignored
- The direction mode (`MODE_CONFIG[4]`) affects the count direction (UP/DOWN)

Use cases:

- Read back encoder-to-line mapping in motion-based acquisition

- Support runtime configuration displays or diagnostics
- Confirm encoder step resolution during high-speed imaging

GetSoftTriggerTime

Pascal Declaration:

```
function ls_getsofttriggertime(addr: Byte; var dwTime: DWORD): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getsofttriggertime(UInt8 addr, UInt32* dwTime);
```

Description:
The `ls_getsofttriggertime` function reads a 32-bit value from the `SOFT_TRIGGER_PERIOD` register (0x08) of the LISM-PI26xx module via I²C. This value defines the period, in microseconds, at which the internal timing generator issues START pulses during software-triggered acquisition.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `dwTime`: Output variable to receive the software trigger period in microseconds

Return Value:

- 0 on success
- Non-zero error code if the I²C read fails

Behavior:

- Reads 4 bytes from register 0x08 (`SOFT_TRIGGER_PERIOD`)
- Value defines the time interval between START pulses when `MODE_CONFIG[2:0] = 0x03` (software trigger)
- Line rate is inversely proportional to the period:

  ```
  line_rate = 1,000,000 / dwTime   [lines per second]
  ```

Technical note:

- Time unit is fixed: **1 µs per step**
- Value is interpreted as an unsigned 32-bit integer (big-endian order)
- This setting is ignored in non-software-trigger modes

Notes:

- Only effective when trigger mode is set to Software Trigger (`MODE_CONFIG = 0x03`)
- Ensure: `dwTime > START_PULSE_HIGH + START_PULSE_LOW`

  to allow sufficient time per acquisition cycle
- Used to control internal acquisition rate without relying on external signals

Use cases:

- Retrieve configured soft trigger period
- Log or display current trigger rate
- Verify or adjust timing settings in adaptive imaging systems

GetTriggerDelay

Pascal Declaration:

```
function ls_gettriggerdelay(addr: Byte; var dwDelay: DWORD): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_gettriggerdelay(UInt8 addr, UInt32* dwDelay);
```

Description:
The `ls_gettriggerdelay` function reads a 32-bit value from the `TRIGGER_OUTPUT_DELAY` register (0x09) of the LISM-PI26xx module via I²C. This value specifies the time delay, in microseconds, between the start of integration (rising edge of the START pulse) and the activation of the trigger output signal.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `dwDelay`: Output variable to receive the trigger delay time (in µs)

Return Value:

- 0 if successful
- Non-zero error code if the I²C read fails

Behavior:

- Reads 4 bytes from register 0x09 (`TRIGGER_OUTPUT_DELAY`)
- The value defines the delay from the start of each acquisition line until the trigger output is asserted

- Trigger pulse timing is defined as:

  ```
  TRIGGER_OUTPUT = ACTIVE from (START + dwDelay) to (START + dwDelay + width)
  ```

Technical note:

- Value is returned as a 32-bit unsigned integer in big-endian order
- Time resolution is **1 µs per step**
- Must be used together with `TRIGGER_OUTPUT_WIDTH` to define the full pulse duration
- Trigger output polarity is defined in `DATA_IF_CONFIG` (bit 6)

Notes:

- Ensure:

  ```
  TRIGGER_OUTPUT_DELAY + TRIGGER_OUTPUT_WIDTH ≤ START_PULSE_HIGH + START_PULSE_LOW
  ```

  to prevent the trigger pulse from exceeding the current acquisition cycle
- Can be used in all acquisition modes
- Immediate effect after register update

Use cases:

- Measure or verify trigger-to-exposure alignment
- Analyze signal timing in automated test setups
- Control activation timing of external illumination or capture systems

GetTriggerWidth

Pascal Declaration:

```
function ls_gettriggerwidth(addr: Byte; var dwWidth: DWORD): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_gettriggerwidth(UInt8 addr, UInt32* dwWidth);
```

Description:
The `ls_gettriggerwidth` function reads a 32-bit value from the `TRIGGER_OUTPUT_WIDTH` register (0x0A) of the LISM-PI26xx module via I²C. This value defines the duration of the trigger output signal, in microseconds, following the configured trigger delay.

It is typically used to synchronize external equipment (e.g. strobes, cameras) with specific phases of the sensor acquisition cycle.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `dwWidth`: Output variable to receive the trigger pulse width (in µs)

Return Value:

- 0 on success
- Non-zero error code if the I²C read fails

Behavior:

- Reads 4 bytes from register 0x0A (`TRIGGER_OUTPUT_WIDTH`)
- The pulse begins after the delay specified in `TRIGGER_OUTPUT_DELAY`
- Trigger output remains active for `dwWidth` microseconds, unless the acquisition cycle ends earlier

Technical note:

- Value is a 32-bit unsigned integer, returned in big-endian format
- Resolution is fixed at **1 µs per step**
- Trigger output polarity is configured in `DATA_IF_CONFIG` (bit 6)
- If the total pulse time exceeds the current cycle, it is automatically truncated

Notes:

- Used in combination with `ls_gettriggerdelay` to define the full timing window
- Ensure that:

  ```
  TRIGGER_OUTPUT_DELAY + TRIGGER_OUTPUT_WIDTH ≤ START_PULSE_HIGH + START_PULSE_LOW
  ```

  to prevent premature cutoff
- Applies to all acquisition modes

Use cases:

- Retrieve trigger pulse timing for diagnostics or alignment
- Monitor synchronization timing for external systems
- Adjust or verify timing in adaptive acquisition workflows

GetPixelCount

Pascal Declaration:

```
function ls_getpixelcount(addr: Byte; var wCount: Word): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getpixelcount(UInt8 addr, UInt16* wCount);
```

Description:
The `ls_getpixelcount` function reads a 16-bit value from the `PIXEL_COUNT` register (0x0B) of the LISM-PI26xx module via I²C. This value defines how many pixels are read out per line during acquisition.
It determines the duration of the `LINE_VALID` signal and the number of 16-bit pixel values transmitted via the parallel data interface.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `wCount`: Output variable to receive the number of pixels per line

Return Value:

- 0 if successful
- Non-zero error code if the I²C read fails

Behavior:

- Reads 2 bytes from register 0x0B (`PIXEL_COUNT`)
- The number of pixels determines:
    - Duration of the `LINE_VALID` signal
    - Amount of data output on the parallel bus per line
- `LINE_VALID` stays high during output of all configured pixels

Technical note:

- Value is returned in big-endian byte order (MSB first)
- Output format is fixed: 16-bit pixels, MSB first
- The value must be compatible with the actual sensor (e.g. 2048 for S11639-01)
- The configured START pulse duration must allow sufficient time for readout

Notes:

- Timing constraint:

  `START_PULSE_LOW ≥ minimum readout time for wCount pixels`

- Use `EDGES_BEFORE_DATA` to define readout offset or implement horizontal cropping
- Can be used in all trigger modes

Use cases:

- Verify current resolution or line length
- Confirm buffer size and timing compatibility with host interface
- Support diagnostics, live displays, or software-driven reconfiguration

GetEdgeDelay

Pascal Declaration:

```
function ls_getedgedelay(addr: Byte; var ucEdges: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getedgedelay(UInt8 addr, UInt8* ucEdges);
```

Description:
The `ls_getedgedelay` function reads an 8-bit value from the `EDGES_BEFORE_DATA` register (0x0C) of the LISM-PI26xx module via I²C. This value defines how many falling edges of the pixel clock are counted after the falling edge of the START pulse before pixel data output begins.

It allows the output timing to be aligned with the valid signal window of the connected image sensor and enables cropping at the beginning of each line.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `ucEdges`: Output variable to receive the number of clock edges before data output

Return Value:

- 0 on success
- Non-zero error code if the I²C read fails

Behavior:

- Reads 1 byte from register 0x0C (`EDGES_BEFORE_DATA`)
- The value defines a delay (in clock edges) before `LINE_VALID` is asserted
- Data output begins with the first pixel clock after the delay expires

Technical note:

- Value is an unsigned 8-bit integer (0–255)
- Sensor-specific default values:
  - For S11639-01: `88` edges before valid data begins
  - Plus internal ADC delay of 9 cycles → `LINE_VALID` begins after ~98 clocks
- Used in combination with `PIXEL_COUNT` and `DATA_IF_CONFIG`

Notes:

- Allows horizontal offset (cropping) of sensor lines
- Must be matched to the sensor's analog-to-digital latency and black level padding
- Clock timing depends on frequency setting in `DATA_IF_CONFIG` (bits [2:0])

Use cases:

- Match readout timing to sensor-specific latency
- Prevent output of invalid or overshoot pixels
- Dynamically adjust image region-of-interest during runtime

GetADCVref

Pascal Declaration:

```
function ls_getadcvref(addr: Byte; var ucVRef: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getadcvref(UInt8 addr, UInt8* ucVRef);
```

Description:
The `ls_getadcvref` function reads an 8-bit value from the `ADC_FULL_SCALE` register (0x20) of the LISM-PI26xx module via I²C. This value defines the full-scale input voltage range of the analog-to-digital converter (ADC), which determines how analog input signals are mapped to the 16-bit digital output range (0…65535).

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `ucVRef`: Output variable to receive the ADC voltage range selection

Return Value:

- 0 on success
- Non-zero error code if the I²C read fails

Value mapping:

ucVRef	Input Range	Description
0x00	1.6 V	High resolution, reduced dynamic range
0x01	2.0 V	Wider range, suitable for stronger signals

Technical note:

- Register 0x20 (`ADC_FULL_SCALE`) contains a single byte
- Value is applied immediately by the internal ADC control logic
- Affects only analog input scaling, not digital format or polarity

Notes:

- Use together with `ADC_GAIN` and `ADC_OFFSET` for optimal analog signal conditioning
- Choosing the appropriate full-scale range helps maximize signal-to-noise ratio (SNR) without clipping

Use cases:

- Verify current ADC range during calibration routines
- Match sensor output amplitude to ADC range for optimal use of resolution
- Support diagnostics, auto-tuning, or field configuration of analog front-end

GetADCGain

Pascal Declaration:

```
function ls_getadcgain(addr: Byte; var ucGain: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getadcgain(UInt8 addr, UInt8* ucGain);
```

Description:
The `ls_getadcgain` function reads the 8-bit value from the `ADC_GAIN` register (0x21) of the LISM-PI26xx module via I²C. This value determines the analog gain factor applied to the input signal before it is converted by the ADC.

The gain allows amplification of small or low-light signals to improve utilization of the ADC's dynamic range and enhance signal-to-noise performance.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `ucGain`: Output variable to receive the ADC gain setting (valid range: 0–63)

Return Value:

- 0 if the value was read successfully
- Non-zero error code if I²C communication failed

Gain mapping and formula:

ucGain	Gain (approx.)	Description
0	1.00×	Unity gain (no boost)
31	~2.45×	Moderate amplification
63	5.85×	Maximum programmable gain

Gain is computed as:

```
Gain = 76 / (76 − ucGain)
```

Technical note:

- Only bits [5:0] of `ucGain` are valid
- Bits [7:6] are reserved and must be zero
- Gain is applied before digitization and affects the analog signal level entering the ADC

Notes:

- Use in conjunction with `ADC_OFFSET` and `ADC_FULL_SCALE` for optimal analog signal conditioning
- Especially useful for dim signals or when sensor operates near its noise floor
- May require calibration for precise gain-to-voltage mapping

Use cases:

- Read current analog gain setting for display or diagnostics
- Restore saved gain values in application profiles
- Monitor and adjust gain in auto-exposure or feedback-controlled systems

GetADCOffset

Pascal Declaration:

```
function ls_getadcoffset(addr: Byte; var wOffset: Word): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getadcoffset(UInt8 addr, UInt16* wOffset);
```

Description:
The `ls_getadcoffset` function reads a 16-bit value from the `ADC_OFFSET` register (0x22) of the LISM-PI26xx module via I²C. This value defines a signed analog offset that is applied to the video signal before digitization. It allows the system to shift the signal baseline, improving black level alignment and ensuring optimal use of the ADC input range.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `wOffset`: Output variable to receive the signed offset (in sign-magnitude format)

Return Value:

- 0 on success
- Non-zero error code if the I²C read fails

Encoding format:

wOffset	Sign	Magnitude	Approx. Voltage Shift
0x0000	+	0	0.00 mV
0x0001	+	1	+0.98 mV
0x00FF	+	255	+250 mV
0x0100	−	0	−0.00 mV
0x01FF	−	255	−250 mV

Format details:

- Bit 8 (MSB) = Sign bit (0 = positive, 1 = negative)
- Bits 7–0 = Magnitude (0…255)
- Total range: ±250 mV in ≈0.98 mV steps

Technical note:

- Value is applied in the analog domain prior to gain and ADC conversion
- Offset helps compensate for sensor-specific baseline or temperature-induced drift
- Value is returned in big-endian byte order

Notes:

- Use in combination with `ADC_GAIN` and `ADC_FULL_SCALE` for complete analog path tuning
- May need to be re-evaluated for each sensor type or operating condition
- Recommended to read and log this value during calibration routines

Use cases:

- Read current black level correction setting
- Display and analyze signal baseline configuration
- Dynamically adjust offset in systems with adaptive dark-level compensation

GetADCBias

Pascal Declaration:

```
function ls_getadcbias(addr: Byte; var wBias: Word): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getadcbias(UInt8 addr, UInt16* wBias);
```

Description:
The `ls_getadcbias` function reads a 16-bit value from the `ADC_INPUT_BIAS` register (0x30) of the LISM-PI26xx module via I²C.

This value corresponds to the output of an internal DAC used to apply an analog bias voltage to the ADC input signal. The bias voltage shifts the input signal downward to compensate for DC offsets in the sensor output.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `wBias`: Output variable to receive the DAC setting (0–1023)

Return Value:

- 0 on success
- Non-zero error code if the I²C read fails

Voltage mapping:

wBias	Output Voltage	Step Size
0	0.000 V	
512	~1.250 V	
1023	~2.500 V	≈ 2.44 mV/step

Formula: `V_bias = (wBias / 1023) × 2.5 V`

Technical note:

- Full range: 0 V to 2.5 V
- Resolution: 1024 steps, controlled by internal DAC
- Value is returned in big-endian byte order
- The configured bias voltage is subtracted from the sensor signal before gain and digitization

Notes:

- Use in conjunction with `ADC_GAIN`, `ADC_OFFSET`, and `ADC_FULL_SCALE` for precise analog signal conditioning
- Affects the absolute input baseline of the ADC
- Useful for coarse correction of sensor-specific DC levels or dark offsets

Use cases:

- Retrieve and monitor the configured input bias voltage used for signal offset correction
- Verify analog front-end calibration state or default initialization
- Support adaptive real-time adjustments or multi-sensor synchronization

SaveSettings

Pascal Declaration:

```
function ls_savesettings(addr: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt __stdcall ls_savesettings(UInt8 addr);
```

Description:
The `ls_savesettings` function writes a command to the `STORE_SETTINGS` register (0xF2) of the LISM-PI26xx module via I²C. This instructs the module to store the current configuration of all relevant registers into non-volatile EEPROM memory.

The stored settings will be automatically restored during the next power-up, allowing for persistent operation without reinitialization.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module

Return Value:

- 0 on success
- Non-zero error code if the store command fails or is not acknowledged

Behavior:

- Triggers a special EEPROM write process via command register 0xF2
- Stores the current state of all configurable registers (timing, mode, ADC, interface, etc.)

- Does **not** store the temporary I²C address unless saved separately via `ls_saveslaveaddress`

Technical note:

- No data is passed with the command; only the register address is written
- EEPROM write requires a few milliseconds; ensure no power interruption during this time
- Affects all registers that define the module's acquisition and signal behavior

Notes:

- Stored settings include:
  - `START_PULSE_HIGH`, `START_PULSE_LOW`
  - `TRIGGER_OUTPUT_DELAY`, `TRIGGER_OUTPUT_WIDTH`
  - `MODE_CONFIG`, `PIXEL_COUNT`, `QUAD_COUNT`, etc.
  - `DATA_IF_CONFIG`, `ADC_GAIN`, `ADC_OFFSET`, `ADC_EXT_REF`
- `TEMP_SLAVE_ADDRESS` is excluded unless committed using `ls_saveslaveaddress`
- After power-on, the EEPROM-stored configuration is automatically restored before operation begins

Use cases:

- Make sensor configuration persistent across power cycles
- Ship or deploy fully preconfigured modules
- Enable setup profiles in calibration, test, or production tools

ReloadSettings

Pascal Declaration:

```
function ls_reloadsettings(addr: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_reloadsettings(UInt8 addr);
```

Description:
The `ls_reloadsettings` function sends a command to the `RELOAD_SETTINGS` register (0xF3) of the LISM-PI26xx module via I²C. It instructs the module to reload all configuration registers from EEPROM memory, restoring the previously stored settings and discarding any temporary or runtime changes.

This function is equivalent to performing a full configuration reset — without needing to power-cycle the device.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module

Return Value:

- 0 on success
- Non-zero error code if the reload command failed

Behavior:

- Reads the latest stored settings from EEPROM and writes them into all active configuration registers
- Overwrites:
    - START pulse and trigger timing
    - MODE_CONFIG
    - ADC and interface settings
    - Any other writable control register
- Does **not** affect the current slave address

Technical note:

- Command-only operation; no data payload
- Acquisition must be stopped prior to execution (e.g., via `ls_setstate(addr, 0x00)`)
- Changes take effect immediately after reload

Notes:

- Reloaded settings reflect the last configuration saved with `ls_savesettings`
- Useful for undoing dynamic reconfiguration or software-side experiments
- Helps ensure a clean operational state without requiring physical reboot

Use cases:

- Restore factory or user-saved configuration state
- Implement "revert to saved" functionality in control software
- Ensure repeatable initialization without reloading from host side

ResetSettings

Pascal Declaration:

```
function ls_resetsettings(addr: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_resetsettings(UInt8 addr);
```

Description:
The `ls_resetsettings` function writes a command to the `RESET_SETTINGS` register (0xF4) of the LISM-PI26xx module via I²C. This command instructs the module to:

1. Immediately reset all configuration registers to firmware-defined factory defaults
2. Overwrite the EEPROM with these default values — erasing any previously saved user configuration

This ensures that both volatile and non-volatile configuration states reflect the factory baseline without requiring any additional restart or reload.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module

Return Value:

- 0 on success
- Non-zero error code if the reset command fails

Behavior:

- Triggers an internal reset of all configurable registers
- Writes default values directly to EEPROM
- The new configuration becomes **instantly active** in the module's operation
- All prior saved settings are permanently discarded

Technical note:

- No restart or `ls_reloadsettings` is needed — defaults take effect immediately
- EEPROM write may require several milliseconds
- The I²C slave address remains unchanged

Notes:

- Use only if all custom configuration is to be erased

- Ideal for factory reset, field servicing, or standardized module provisioning

Use cases:

- Return a module to factory default configuration
- Erase custom user settings before redeployment
- Eliminate unknown or unstable configurations in support scenarios

GetInitStatus

Pascal Declaration:

```
function ls_getinitstatus(addr: Byte; var ucStatus: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getinitstatus(UInt8 addr, UInt8* ucStatus);
```

Description:
The `ls_getinitstatus` function reads an 8-bit value from the `INIT_STATUS` register (0xFE) of the LISM-PI26xx module via I²C. Each bit of this register reflects the presence and initialization state of specific hardware components and interfaces within the sensor and processor boards.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `ucStatus`: Output variable to receive the 8-bit status byte

Return Value:

- 0 on success
- Non-zero error code if the I²C read fails

Bit layout of `ucStatus`:

Bit	Meaning	Status
0	EEPROM HW1 present	1 = OK, 0 = not detected
1	DAC present	1 = OK, 0 = not detected
2	EEPROM HW2 present	1 = OK, 0 = not detected
3	Clock & Timing Generator (CTG) present	1 = OK, 0 = not detected
4	I²C bus initialized	1 = OK, 0 = failed
5	DAC initialized	1 = OK, 0 = failed
6	ADC initialized	1 = OK, 0 = failed
7	CTG (Clock & Timing Generator) initialized	1 = OK, 0 = failed

Behavior:

- Reads register 0xEE and provides detailed hardware readiness status
- Each bit can be interpreted individually to locate the source of initialization problems
- All bits must be set (0xFF) for full operational readiness

Notes:

- A value of 0xFF means all devices were detected and initialized successfully
- Use this function after startup or `ls_hw_reset` to validate system state
- Bits 0–3 reflect hardware detection, bits 4–7 reflect initialization success

Use cases:

- Detect missing or malfunctioning components during startup
- Provide user feedback or diagnostics in setup and service tools
- Prevent acquisition start if essential subsystems are not operational

GetComResult

Pascal Declaration:

```
function ls_getcomresult(addr: Byte; var ucResult: Byte): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getcomresult(UInt8 addr, UInt8* ucResult);
```

Description:
The `ls_getcomresult` function reads an 8-bit value from the `COM_RESULT` register (0xFE) of the LISM-PI26xx module via I²C. This register provides detailed information about the result of the **most recently executed command**, including whether communication with various internal subsystems succeeded or failed.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `ucResult`: Output variable to receive the command result code

Return Value:

- 0 on successful I²C read

- Non-zero error code if the I²C communication itself failed

Bit layout of `ucResult`:

Bit	Subsystem	Meaning
0	CTG (Clock & Timing Gen.)	1 = Last access successful, 0 = error
1	ADC	1 = Last access successful, 0 = error
2	DAC	1 = Last access successful, 0 = error
3	EEPROM (HW2, Processor)	1 = Last access successful, 0 = error
4	EEPROM (HW1, Sensorboard)	1 = Last access successful, 0 = error
5–7	Reserved	Always 0, ignore

Behavior:

- Indicates for each internal hardware unit whether the last access attempt was acknowledged successfully
- The register is updated automatically after each configuration or control command targeting a device behind the I²C multiplexer

Notes:

- A value of 0x1F (binary 00011111) means all involved components confirmed the last command
- This register is **not cleared automatically** after being read
- Useful for isolating which internal I²C slave (CTG, DAC, etc.) failed to respond or execute a command

Use cases:

- Validate the integrity of recent write or read operations
- Enable detailed diagnostics when configuration commands fail
- Log subsystem-level errors during system operation or testing

Version and Hardware Identification

This section provides access to essential versioning and identification data stored within the LISM-PI26xx module. These functions allow the host application to retrieve firmware versions, hardware revisions, and board identifiers from various subsystems of the device. Such information is crucial for ensuring compatibility between software and hardware, enabling version-dependent behavior, and maintaining traceability in production or field deployments.

The module includes two distinct firmware domains: one for the communication and configuration processor (MCU P1) and another for the clock and timing generator (P2). Each operates independently and may evolve separately. Dedicated functions are provided to read their respective version numbers, enabling fine-grained version control across software updates or hardware generations.

In addition to firmware versions, hardware identification values are available for both the sensor board (HW1) and the processor board (HW2). These identifiers reflect manufacturing variants, layout revisions, or sensor-specific configurations. Applications can use this information to adapt calibration routines, enforce hardware compatibility checks, or record system configurations for service and quality control purposes.

These version and ID queries are read-only and do not alter the system state. They are intended to be used during startup or diagnostics to ensure that the expected firmware and hardware environment is present. This safeguards system stability and simplifies deployment in environments with mixed hardware revisions or modular system architectures.

GetMCUVersion

Pascal Declaration:

```
function ls_getmcuversion(addr: Byte; var wVersion: Word): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_getmcuversion(UInt8 addr, UInt16* wVersion);
```

Description:
The `ls_getmcuversion` function reads a 16-bit value from the `P1_FM_VERSION` register (0xFD) of the LISM-PI26xx module via I²C. This register contains the firmware version of the secondary processor (MCU P1), which is responsible for I²C communication, EEPROM access, and coordinating configuration with the timing generator.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `wVersion`: Output variable to receive the firmware version of the internal MCU (P1)

Return Value:

- 0 on success
- Non-zero error code if the I²C read fails

Version encoding:

Format	Example
High byte	Major
Low byte	Minor
0x0102	Version 1.2

Behavior:

- Reads 2 bytes from register 0xFD
- The version returned reflects the firmware currently running on the MCU that manages all internal configuration (not USB communication)

Important distinction:

- This function does **not** return the version of the USB controller firmware
- To query the USB interface firmware version, use `ls_getfwversion(index: Integer): Word;`

Use cases:

- Detect installed MCU firmware version for compatibility checks
- Enable conditional features depending on firmware capability
- Report version numbers during diagnostics or in system logs

Notes:

- The returned version is static during runtime
- Always read via I²C interface, not USB
- Used to verify module-side firmware when multiple hardware revisions exist

GetCTGVersion

Pascal Declaration:

```
function ls_getctgversion(addr: Byte; var wVersion: Word): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt __stdcall ls_getctgversion(UInt8 addr, UInt16* wVersion);
```

Description:
The `ls_gettgversion` function reads a 16-bit value from the `P2_FM_VERSION` register (0xFE) of the LISM-PI26xx module via I²C. This register reports the firmware version of the **clock and timing generator** subsystem (processor P2), which is responsible for START pulse timing, acquisition control, line/frame synchronization, and trigger logic.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `wVersion`: Output variable to receive the firmware version of the timing generator (P2)

Return Value:

- 0 on success
- Non-zero error code if the I²C read fails

Version encoding:

Format	Example
High byte	Major
Low byte	Minor
0x0105	Version 1.5

Behavior:

- Reads 2 bytes from register 0xFE
- The value reflects the version of the real-time subsystem that governs acquisition and output signal behavior

Notes:

- Value is **read-only** and defined at firmware build time
- To read the firmware version of the configuration and communication processor (P1), use `ls_getmcuversion`

Use cases:

- Perform feature compatibility checks based on firmware version
- Log and track installed P2 firmware across devices
- Verify version-dependent timing behavior or feature support (e.g. encoder logic, multi-line trigger)

GetHW1Id

Pascal Declaration:

```
function ls_gethw1id(addr: Byte; var wId: Word): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_gethw1id(UInt8 addr, UInt16* wId);
```

Description:
The `ls_gethw1id` function reads a 16-bit value from the `SENSOR_BOARD_ID` register (0xFA) of the LISM-PI26xx module via I²C. This register provides the **hardware ID** of the connected sensor board (referred to as HW1). It encodes information such as the sensor model, board layout variant, or custom configuration, allowing host software to identify and adapt to the attached hardware.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module

- `wId`: Output variable to receive the sensor board ID (HW1)

Return Value:

- 0 on success
- Non-zero error code if the I²C read fails

Behavior:

- Reads 2 bytes from register 0xFA
- The value is defined during production and stored in EEPROM
- Intended for identification of the front-end sensor board only

Notes:

- Value is **read-only** and not configurable by the user
- Interpretation of the ID is module- or manufacturer-specific
- Helps ensure correct calibration, configuration, or firmware mapping

Use cases:

- Select appropriate timing or ADC parameters for specific sensor types
- Verify hardware identity during automated test or setup
- Record sensor board version and compatibility in service logs

GetHW1Version

Pascal Declaration:

```
function ls_gethw1version(addr: Byte; var wVersion: Word): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt16 __stdcall ls_gethw1version(UInt8 addr, UInt16* wVersion);
```

Description:
The `ls_gethw1version` function reads a 16-bit value from the `SB_HW_VERSION` register (0xFB) of the LISM-PI26xx module via I²C. This register provides the **hardware version** of the connected sensor board (HW1), which may reflect design revisions, layout changes, or supported features beyond what is indicated by the board ID alone.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `wVersion`: Output variable to receive the hardware version of the sensor board

Return Value:

- 0 on success
- Non-zero error code if the I²C read fails

Behavior:

- Reads 2 bytes from register 0xFB
- Used to distinguish between different revisions of sensor boards that may share the same ID (`SENSOR_BOARD_ID`)
- Allows firmware or host-side logic to adapt to board-specific behavior

Notes:

- Register is **read-only** and programmed at the factory
- Independent of the board ID (`ls_gethw1id`)
- Use in conjunction with `ls_gethw2version` to report complete hardware configuration

Use cases:

- Enable hardware-specific configuration or calibration paths
- Ensure compatibility between firmware and sensor front-end electronics
- Track hardware versions during testing, production, or field deployment

GetHW2Version

Pascal Declaration:

```
function ls_gethw2version(addr: Byte; var wVersion: Word): DWORD; stdcall;
```

C/C++ Declaration:

```
UInt32 __stdcall ls_gethw2version(UInt8 addr, UInt16* wVersion);
```

Description:
The `ls_gethw2version` function reads a 16-bit value from the `PB_HW_VERSION` register (0xFC) of the LISM-PI26xx module via I²C. This register returns the **hardware version** of the processor

board (HW2), which includes the timing generator (P2), supporting logic, and digital interface circuitry.

This version information helps identify design revisions or hardware variations that may influence timing, signal behavior, or configuration options.

Parameters:

- `addr`: I²C slave address of the LISM-PI26xx module
- `wVersion`: Output variable to receive the processor board hardware version

Return Value:

- 0 on success
- Non-zero error code if the I²C read fails

Behavior:

- Reads 2 bytes from register `0xFC`
- Reports the factory-programmed hardware revision of the digital electronics subsystem (HW2)

Notes:

- This is a **read-only** identifier set during production
- Do not confuse with `P2_FM_VERSION`, which indicates firmware version of the clock generator
- Combine with `ls_gethw1version` to identify the full module hardware version (sensor + processor)

Use cases:

- Match firmware behavior to specific processor board revisions
- Distinguish between hardware generations in production or service environments
- Enable conditional support for optional features depending on board capabilities

Revision history

Rev.	Date	Description
1.0	06 August 2025	Initial release