MAX22X88 DRIVER USER GUIDE

Table of Contents

MAX22X88 DRIVER USER GUIDE

1 INTRODUCTION

This document describes the MAX22x88 baremetal software driver for the MAX22088 and MAX22288 Home Bus System compatible transceivers.

1.1 OVERVIEW

The MAX22x88 driver provide support for sending and receiving data through MAX22088 and MAX22288 transceivers. The data is exchanged in terms of frames and it doesn't depend on any particular network protocol.

The driver is structured in four layers, of which two are platform dependent. From highest to lowest, the layers are:

- Core driver: Responsible for driver initialization and core functionality. Provides the driver API for the user application.
- IO layer: Responsible for interpreting and generating the input and output signals. This layer defines a HAL API for managing the required peripheral drivers. The driver provides a bitbang-based implementation.
- HAL implementation: Provides an implementation for the HAL API defined by the IO layer.
- Peripheral drivers: Responsible for directly interacting with the platform peripherals.

The structure is summarized in Figure 1. See Driver Structure for details.

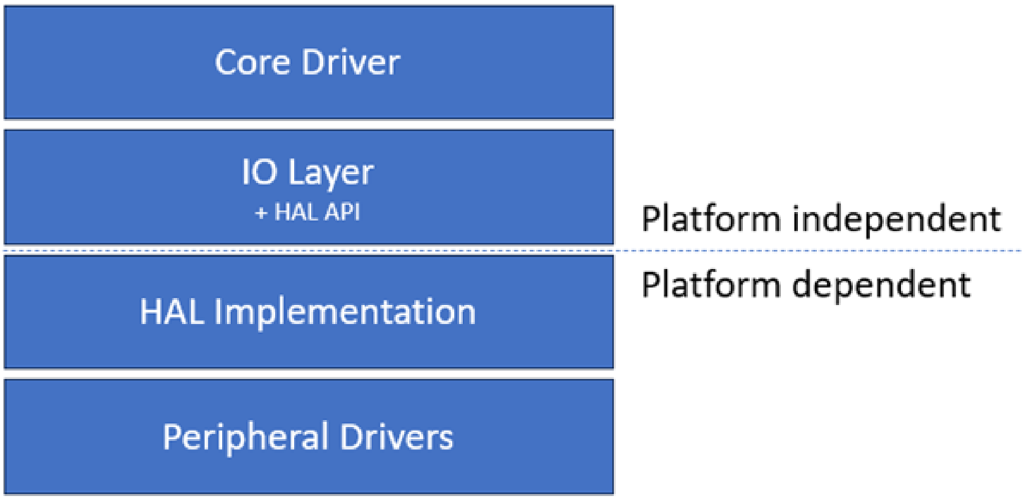


Figure 1. Driver structure.

A microcontroller is interfaced with the transceiver via the DIN, DOUT and RST pins. DIN and RST is driven by the microcontroller, and DOUT is driven by the transceiver. Pins H+ and H- provide a multi-directional interface with the Home Bus network. Figure 2 summarizes the transceiver's interface.

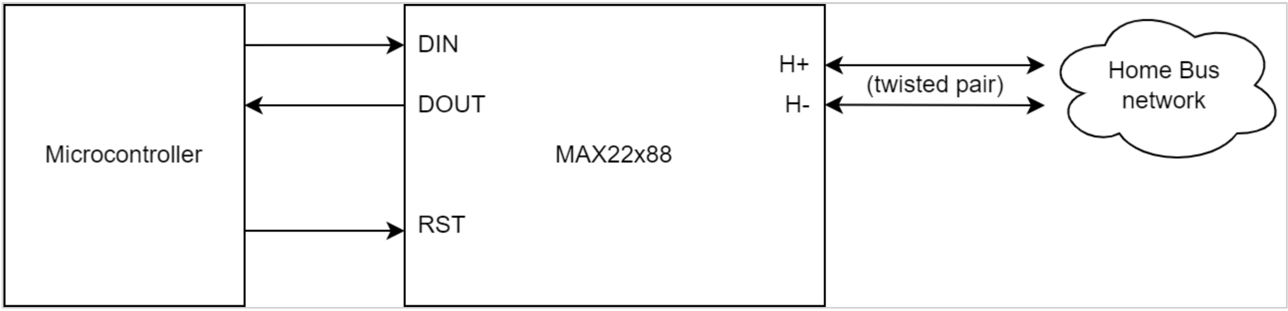


Figure 2. Transceiver interface.

The requirements for the microcontroller pins depend on the IO Layer implementation. See section IO Layer Implementations for details.

For more detail on the pins and transceiver operation, consult the part's datasheet.

1.2 SUPPORTED PLATFORMS

The driver supports platforms running without an operating system. For the driver to run on a platform, a HAL API port to the desired platform must be provided.

To demonstrate the driver integration, the bitbang IO layer has been ported to the MAX32670 MCU. The software package contains an example project to illustrate its usage. See Example Project for details on the example project, and IO Layer and IO Layer Implementations for details on implementation.

2 SYSTEM REQUIREMENTS

2.1 DRIVER REQUIREMENTS

The driver requires a platform with the following peripherals, as required by the bitbang-based IO layer implementation:

- 3x GPIO
    - Two GPIO configured as outputs.
    - One GPIO configured as input with falling-edge detection interrupt support.
- 1x Timer
    - Configured to generate compare interrupts at 4x the desired data transmission speed.

There are no specific software requirements, except for a C toolchain for the target platform.

There are the following optional dependencies:

- Doxygen: Used to generate the API reference manual. See API documentation for details.
- Make: used to avail of the integration mechanism provided for building the driver. See Project Build Integration for details.

2.2 OTHER REQUIREMENTS

There are specific software and hardware requirements for running the example project. See Example Project for details.

3 GETTING STARTED

The MAX22x88 driver is available on a GitHub repository. The repository contains the driver source code, a Doxyfile for generating API Reference Manual, supporting files for Make integration and an example project.

3.1 FOLDER STRUCTURE

- .vscode/ - Visual Studio Code IDE files for building example projects
- doc/ - API documentation
- examples/ - Example projects
- inc/ - Driver headers
- src/ - Driver sources

3.2 API DOCUMENTATION

The API Reference Manual can be generated by running Doxygen.

1. Having a copy of the repository, open the root repository folder in a terminal.
2. cd into the “doc” folder.
3. Run the Doxygen tool with “doxygen”.
4. The output is generated in “doc/output/html/”. To access the documentation, open the “index.html” file in a web browser.

4 DRIVER STRUCTURE

4.1 CORE DRIVER

This layer defines the API for initializing and operating the driver. It is responsible for initializing the core driver functionality.

The core driver functionality includes:

- Configuring the transceiver for Rx and Tx operations. The transceiver is normally on Rx mode and is set to Tx during the execution of data transmissions requested by the user.
- Initializing an Rx circular buffer, which stores incoming frames as detected by the IO layer.
- Connecting to the IO layer by means of function pointers.
- Performing IO layer initialization.

4.2 IO LAYER

The IO layer is responsible for input and output operations.

The input operations refer to detecting and decoding incoming frames. Once decoded, the frame is passed to a callback defined by the core layer, which then to store the received frame and makes it readable by the user application. The output operations refer to encoding and transmitting frames requested by the user.

The means by incoming frames are detected and decoded, and output frames are encoded and transmitted, depend on the specific IO layer implementation. The HAL API is defined based on the requirements of these implementation-specific operations.

It is the IO layer implementation that defines the requirements for integrating the driver to a particular platform. It consists of implementing the HAL API and performing any platform configuration steps.

For detailed information on specific implementations, see IO Layer Implementations.

4.2.1 HAL API

The HAL API serves as an interface for the required platform peripheral driver functionality required by the IO layer implementation. Performance requirements, if any, are defined by the IO layer implementation.

4.3 HAL IMPLEMENTATION

This is a platform-dependent layer, which provides an implementation to the HAL API defined by the IO layer. Through this implementation, the IO layer has access to the platform peripherals.

4.4 PERIPHERAL DRIVERS

This is a platform dependent layer that controls the platform’s peripherals.

5 IO LAYER IMPLEMENTATIONS

This section details the IO layer implementation bundled with the driver.

5.1 BITBANG IO LAYER

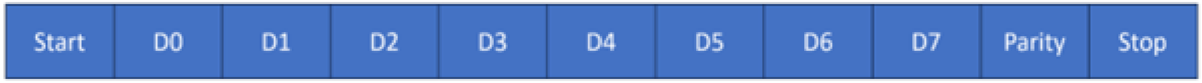
A bitbang protocol is implemented, which makes use of GPIOs, timers and their associated interrupts to transmit and receive frames.

5.1.1 Protocol, Encoding, and Frame Format

This layer implements an asynchronous protocol with parity bit check, 50% duty cycle, configurable baud, and 8 data bits per frame, transmitted lowest significant bit first.

The signal defaults to “high” voltage level when idle, and the start of a frame is defined by a start bit that brings the line “low”. It is followed by 8 data bits, a parity bit, and a stop bit. The frame format is shown in Figure 3.

The parity bit serves as an integrity check and the stop bit guarantees the line is back “high” after the frame is transmitted. The value of the parity bit is set as to make the complete frame have an even number of "low" bits.



Start	D0	D1	D2	D3	D4	D5	D6	D7	Parity	Stop

Figure 3. Frame format.

Because of the 50% duty cycle, each bit period is divided in two sub-periods of equal duration: on the first half, referred to as the “on-duty period”, the signal corresponds to the actual bit value (0 for “low”, 1 for “high”). On the second half, referred to as the “off-duty period”, the line is held “high”.

As an example, consider the how the value 123 would be transmitted. In binary, it’s represented as 01111011b. The parity bit is 0, since that adds up to an even number of 0s: 1 from the start bit, 2 from the data bits, and 1 from the parity bit itself. The complete signal for the frame, including the start, parity, stop bits and 50% duty cycle is represented on Figure 4.

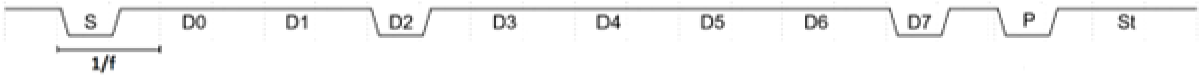


Figure 4. Signal representing the transmission of the value “123”. S is the start bit, Dn are the data bits, P is the parity bit, and St is the stop bit. f is the symbol rate, and 1/f is the bit time.

For information regarding the signal on the power-over-data line, consult the transceiver’s datasheet.

5.2 RX AND TX PROCEDURE

The Rx procedure depends on falling edge ISR and timer interrupts-on-compare on regular intervals. It begins when a falling edge is detected on DOUT and the user application ISR invokes the bitbang IO layer’s callback. The signal on DIN is sampled at 2x the configured baud rate, twice per bit period. Every bit is sampled once on the on-duty and once on the off-duty period. The first sampling of the procedure is performed is in the on-duty period of the start bit, and the last one is the off-duty period of the stop bit.

If there are integrity problems with the frame, it is discarded. Otherwise, it's pushed to the Rx software buffer. Integrity problems include:

- Signal sampled "high" during start bit on-duty period.
- Signal is sample "low" during stop bit on-duty period.
- Parity bit mismatch.
- Signal is sampled "low" during any off-duty period.

The Tx procedure depends on timer interrupts on regular intervals. It starts when the user invokes the transmission function. The driver enables the transceiver's transmitter via RST, then writes the signal to DIN at the appropriate times. Once all frames requested have been transmitted, the transmitter is disabled.

The detection of incoming frames is disabled during transmission.

5.2.1 HAL

The HAL API is specified by the functions declared in "bitbang.hal" and "common.hal". The functions make use of GPIO, timer and interrupt configuration.

The hardware requirements for the platform are:

- 3x GPIO
    - 1x connected to RST. It's configured as an output and enables/disables the transmitter appropriately.
    - 1x connected to DIN. It's configured as an output. The signal at this pin represents the data written to the bus.
    - 1x connected to DOUT. It's configured as an input. The signal at this pin represents the data on the bus. It must support falling edge interrupts.
- 1x Timer
    - It generates interrupts-on-compare at 4 times the configured signal baud. Note that this rate is higher than the Tx and Rx sampling times.

Note: The transceiver's DIN and DOUT pins are named given the transceiver's perspective, and that is why DIN is an output and DOUT is an input from the platform's perspective.

The HAL implementation determines which GPIO and timer peripheral are used.

Check the file contents or the API Reference Manual for more information.

5.2.2 Initialization

The IO layer is initialized during the core driver initialization.

A wrapper function **adi_max22x88_InitBitbang** is provided which initializes the driver with the bitbang IO layer.

5.2.3 Integration

The user must configure a falling edge interrupt on the GPIO connected to DOUT. When the falling edge is detected, the user must call the **adi_max22x88_FallingEdgeIntCallback** function.

6 DRIVER CONFIGURATION AND INITIALIZATION

The driver is initialized by calling the **adi_max22x88_Init** function. This function is responsible for initializing the core driver and the IO layer. It dynamically allocates the Rx buffer based on the length argument and forwards the initialization parameters to the IO layer.

The initialization function returns **MAX22X88_ERR_OK** on success.

7 DRIVER USAGE

The driver can be used by the user application once it is initialized, and any IO layer integration steps have been performed.

Data availability can be checked by polling the driver with **adi_max22x88_IsAvailable**. This function checks if there is data available in the Rx buffer. If data is available, it can be read with **adi_max22x88_Read**. Once data is read, it is removed from the Rx buffer.

The following code snippet shows how the driver can be initialized with the bitbang IO layer operating at 9600 baud, and an Rx buffer of length 128. An infinite loop is used to poll and read any incoming data, which is then echoed back to the bus.

```c
int main(void) {
    adi_max22x88_t driver;  // Create instance of the driver.
    adi_max22x88_bitbang_InitParams_t params = { .hbs_baud = 9600 };  // Create IO layer initialization parameters.
    size_t rx_buffer_len = 128;  // Buffer
    adi_max22x88_InitBitbang(&driver, &params, rx_buffer_len);  // Initialize the driver. In this case, the `max22x88_init` wrapper provided by the IO layer is used.
    while (1) {
        if (adi_max22x88_IsAvailable(&driver)) {  // Check the Rx buffer for available data.
            uint8_t rx_data;
            adi_max22x88_Read(&driver, &rx_data);  // Read one frame from the Rx buffer.
            adi_max22x88_Transmit(&driver, &rx_data, 1);  // Echo the received data.
        }
    }
}
```

See the example project for a complete application code.

8 PROJECT BUILD INTEGRATION

To use the drivers with your project, three sets of source files must be compiled:

- Core driver
- IO layer
- IO layer HAL implementation

The core driver and bitbang IO layer sources are in "src/". The include folder "inc/" must be specified.

The IO layer HAL implementation is generally provided by the user. An implementation for the MAX32670 is available in "src/platform/hal/max32670".

8.1 MAKE INTEGRATION

If the project build system is based on Make, the user can make use of a make integration file "Filelists.mk". This file sets variables which aggregate the required source files and include paths, so they don't have to be manually specified.

Each variable specified servers a different purpose

- Prefix
  - HOMEBUS_MAX22X88_*: core driver files
  - HOMEBUS_MAX22X88_BITBANG_*: bitbang IO layer.
- Suffix:
  - *_SRCS: list of source files to be compiled
  - *_INC: list of required include directories.

The expected usage is:

1. Within the project's Makefile, set the variable "MAX22X88_ROOT_DIR" to the driver's repository root folder, relative to the application's Makefile.
2. Include "Filelists.mk".
3. The variables set in "Filelists.mk" are now accessible from the project's Makefile.
4. Configure the project's Makefile to compile the sources "*_SRCS" and use the include paths "*_INC".

9 EXAMPLE PROJECT

The example project shows two devices exchanging information over a Home Bus network. The devices are MAX32670 evaluation kits, and the Home Bus network is formed by a MAX22088 evaluation kit. The project makes use of the MAX22x88 driver with a bitbang IO-layer HAL port for the MAX32670 microcontroller, and a protocol stack.

9.1 SETUP

The requirements for setting up the example project are:

Hardware:

- 2x MAX32670 Evaluation Kit
- 1x MAX22088 Evaluation Kit
- 1x 18 V Power supply
- 1x 6V Power supply

Software:

- Analog Devices' MSDK (February 2024 Release)
- Visual Studio Code (tested with v1.86.1)
    - Visual Studio Code extension: VSCode-Maxim (v1.7.0) https://github.com/analogdevicesinc/VSCode-Maxim/tree/v1.7.0
        - See https://github.com/analogdevicesinc/VSCode-Maxim/tree/v1.7.0#installation for installation instructions
        - vscode-cpptools (tested with v1.18.5)
        - cortex-debug (tested with v1.12.1)

The on-board selection jumpers on the MAX22088 and MAX32670 evaluation kits should be configured according to the following table:

Device	Jumper	State
MAX22088 Ev. Kit	JP1	Open
	JP2	1-2
	JP3	1-2
	JP4	1-2
	JP5	Open
	JP6	Open
	JP7	Open

Device	Jumper	State
	JP8	1-2
	JP9	1-2
	JP11	Open
	JP12	1-2
	JP13	Open
	JP14	Open
MAX32670 Ev. Kit	JP4	TX0
MAX32670 Ev. Kit	JP1	1-2
MAX32670 Ev. Kit	JP2	1-2

The hardware should be wired according to the connection diagram shown in Figure 5. The power supplies are connected to the MAX22088 master circuit. The master and remote circuits are to be connected via twisted pair cables. The MCUs are connected to the MAX22088 evaluation kit on either the master or remote sides. The MCUs are powered by a PC via the Micro-USB port. This connection is also used by the MCU to send information at 115200 baud, which can be read by a terminal application.

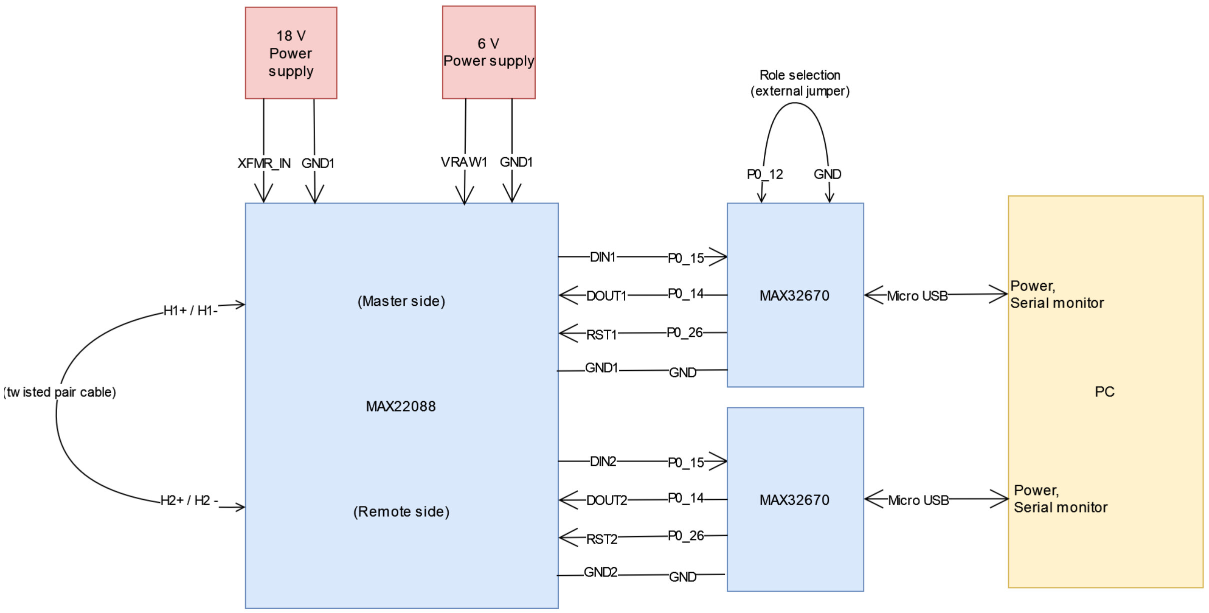


Figure 5. Connection diagram.

Once the setup is configured and wired, the power supplies can be turned on. The MCU with the external role selection jumper will be the master, identified with the green LED (P0_23), and the other device will be the slave, identified with the red LED (P0_22). Additionally, each device will send textual information confirming the initialization. See the following subsection for details.

9.2 APPLICATION

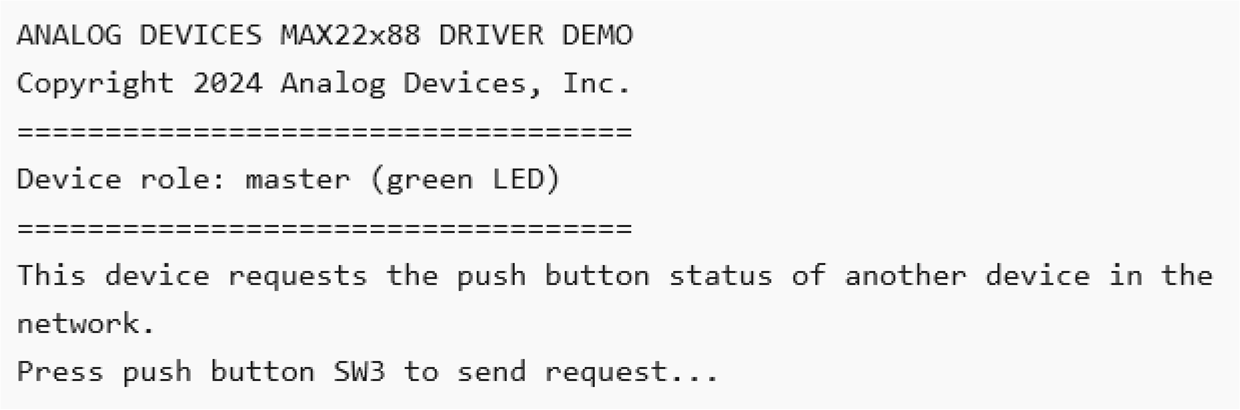
The two devices connected via Home Bus take on different roles (master or slave) and demonstrate the exchange of data upon user interaction. During execution, both devices provide text feedback via UART at 115200 baud.

The application defines two roles for the microcontrollers. MCU 1 is configured as the slave, and MCU 2 is configured as the master. The master sends requests to the slave, and the slave sends responses back to the master.

The user interacts with each device via the SW3 general purpose switch on the Max32670 evaluation kit. By pressing the switch on the master, a request will be sent. The response sent by the slave depends on the state of the switch when the request is received.

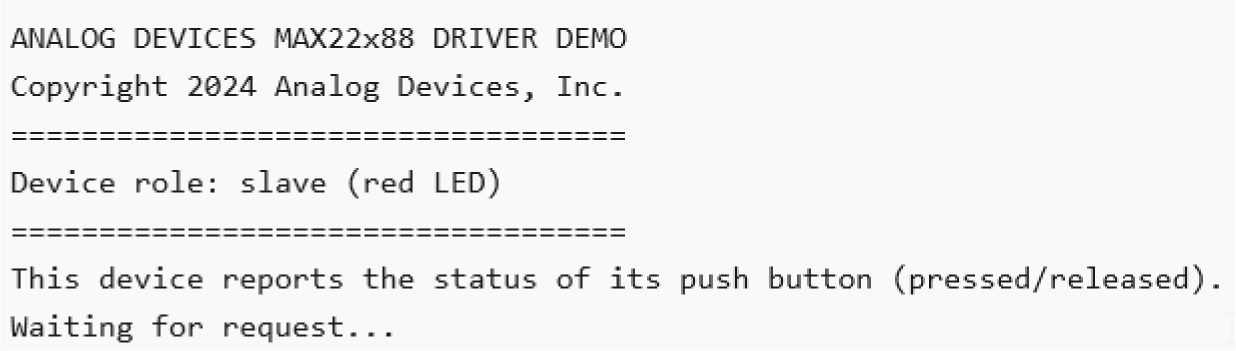
The execution flow is as follows:

- MCU 1 and MCU 2 boot up and the application starts. Each device confirms that initialization was successful via UART (Figures 6 and 7).



```
ANALOG DEVICES MAX22x88 DRIVER DEMO
Copyright 2024 Analog Devices, Inc.
===================================
Device role: master (green LED)
===================================
This device requests the push button status of another device in the
network.
Press push button SW3 to send request...
```

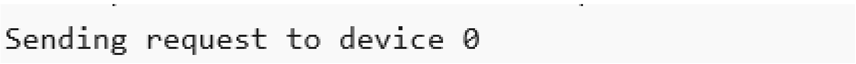
Figure 6. Master device information



```
ANALOG DEVICES MAX22x88 DRIVER DEMO
Copyright 2024 Analog Devices, Inc.
===================================
Device role: slave (red LED)
===================================
This device reports the status of its push button (pressed/released).
Waiting for request...
```

Figure 7. Slave device information

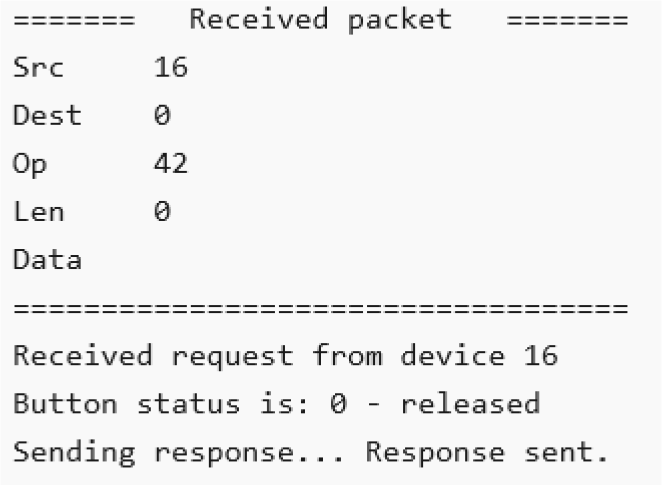
- User presses SW3 on master.
- Master sends a request and logs the information via UART (Figure 8)



```
Sending request to device 0
```

Figure 8. Master sends request to slave after button press.

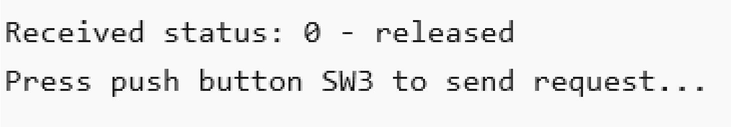
- MCU 1 processes the request, reads the state of its SW3 button.
- MCU 1 replies to MCU 2 with the button state and logs the information via UART (Figure 9)



```
=======   Received packet   =======
Src     16
Dest    0
Op      42
Len     0
Data
===================================
Received request from device 16
Button status is: 0 - released
Sending response... Response sent.
```

Figure 9. Slave device reads the state of its push button and sends reply to master device.

- MCU 2 processes the response from MCU 1 and logs the information via UART (Figure 10)



```
Received status: 0 - released
Press push button SW3 to send request...
```

Figure 10. Master device receives button status from slave device.

9.3 PROTOCOL STACK

The protocol stack sits on the network layer of the OSI model. It serves as the interface between an application and the raw data handled by the driver.

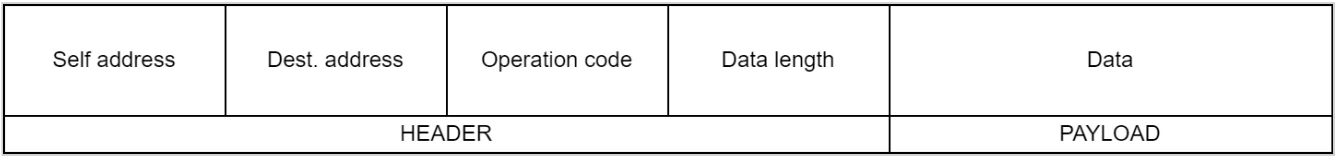
The stack implements a communication protocol that introduces the concept of device addresses and operation codes, enabling the development of applications on multidrop networks with unicast addressing. The underlying protocol implemented by the stack is transparent to the application, to be seen as an implementation detail.

9.3.1 The protocol

The protocol organizes the frames into packets formed by a 4-byte header and a payload of up to 255 bytes, represented in Figure 11.

Each byte in the header is a different field and serves a different purpose. The following is a description of each field, in the order that they're transmitted:

- Self-address: the address of the device transmitting the packet.
- Destination address: the address of the destination device
- Operation code: an application-defined value that signifies the purpose of the packet. Can be used by the application to decide how to interpret the payload.
- Data length: the number of bytes to follow as the payload. Can be 0.
- Data: the payload of the packet, application defined. The number of bytes in this field is equal to the value in "Data length".



Self address	Dest. address	Operation code	Data length	Data
HEADER				PAYLOAD

Figure 11. Packet format.

9.3.2 Using the stack

The stack is implemented in terms of the **adi_hbs_t** structure and supporting functions. To use the stack, and object of type **adi_hbs_t** must first be instantiated and initialized. Afterwards, a pointer to this object is passed as an argument to the functions defined in the stack API.

As the interface between the application and the driver, the stack goes through an integration step, for both inbound and outbound traffic. The outbound integration is handled by the initialization API, while the inbound integration happens on the user application. The rest of the section explores the initialization procedure, and how the stack is used in the application.

The stack object can be initialized with the **adi_hbs_InitMax22x88** function. This arguments to this function are: a pointer to the stack to be initialized, an application-defined 8-bit address, and a pointer to an instance of an adi**_max22x88_t** driver. The 8-bit address should be unique in the network. It is used by the stack to prepare outbound packets, and filter inbound packets. The driver instance is used for the outbound integration: any packets being sent through the stack will be forwarded to this instance of the driver.

Once the stack is initialized, the user application can register an Rx callback with the **adi_hbs_RegisterRxCb** function. This callback is invoked by the stack once per incoming packet. It is the intended way for the application to process incoming data.

The last step is to write the inbound integration. It involves reading incoming data directly from the driver and forwarding it to the stack. As an example, refer to the following snippet. Error checking and driver initialization have been omitted for simplicity.

```
#define SELF_ADDRESS 0x01 // arbitrary, unique value

hbs_err_e rx_callback(adi_hbs_t* hbs, adi_hbs_Packet_t* packet) {
    // Application-specific code
    // `hbs` is the stack that triggered the callback
    // `packet` is the a struct containing the incoming packet data
}

int main(void) {
    adi_max22x88_t driver;
    // ... driver initialization omitted

    adi_hbs_t hbs; // create stack instance
    adi_hbs_InitMax22x88(&hbs, SELF_ADDRESS, &driver);  // initialize stack
    adi_hbs_RegisterRxCb(&hbs, rx_callback); // register rx callback
    while (1) {
        if (adi_max22x88_IsAvailable(&driver)) {
            uint8_t data;
            adi_max22x88_Read(&driver, &data);  // read incoming data directly from the driver's SW buffer
            adi_hbs_Received(&hbs, data);  // forward the data to the stack.
        }
    }
}
```

10 EXTERNAL DOCUMENTS

MAX22088 https://www.analog.com/en/products/max22088.html

MAX22288 https://www.analog.com/en/products/max22288.html

MAX22088 Evaluation Kit https://www.analog.com/en/resources/evaluation-hardware-and-software/evaluation-boards-kits/max22088evkit.html

MAX32670 Evaluation Kit https://www.analog.com/en/resources/evaluation-hardware-and-software/evaluation-boards-kits/max32670evkit.html