QUICK START GUIDE

I2C-Interface

# Introduction

This guide is supposed to provide a rough overview how to quickly access and use the   
I2C-Interface consisting of a composition of an Arduino- and Python-Implementation of the FluidiscopeApp. The interface, in turn, accesses external libraries to establish the actual I2C-Bus connection. In particular, these libraries are SMBus on Python-side and the Wire-Library on Arduino side. This guide is not meant to be a regular manual or complete reference. However, it will show how

* to send commands to the Arduino (Slave)
* to send responses to the RaspberryPi (Master)
* received messages are processed on each side
* the general command-syntax is structured
* commands are processed on Arduino side
* to modify the Arduino-file (\*.ino) in order to implement custom behaviour

Overview

Since the I2C-specification does not provide any possibility of the slave to send data as long as not requested from the master, the whole control over the I2C-Bus is carried out in the Class I2CDevice in the file *I2CDevice.py* which is written in Python 2.7 and is supposed to be run from an I2C-capable master device (e.g. RaspberryPi). I2CDevice provides an object-oriented interface, thus, each device connected to the master over the Bus (i.e. slave), can be controlled via an instance of I2CDevice. Each slave has to be properly prepared according to the protocol introduced by this quick guide, in order to establish communication.

The separate individual preparation of each slave is unavoidable, considering the slave – which usually is a microcontroller-board – has to be flashed and is actually carrying out the desired behaviour at the terminal. However, the implemented communication protocols blueprint has taken that into account and targets to make user-modifications as easy as possible. The \*.ino -blueprint is given in Arduino language, a C-Dialect, and will be explained along an example of an actual implementation which controls a stepmotor.

|  |
| --- |
|  |
| **Fig. 1:** Schematic Diagram of Communication-Flow. |

This guide will mainly focus on the first nodes away from the user in Fig. 1 and how modifications on this level take effect within the system.

The standardized I2C-Protocol enables communication between devices in terms of electrical engineering. Any semantics considering the bus-subscribers have to be implemented by the user. In order to establish a stable communication between different platforms connected to the same bus, an additional communication protocol has been introduced which sets a few certain rules. In particular, it will define which vocabulary has to be known on each side, how commands have to be formatted and which syntax has to be applied.

General Command Syntax

In order to identify user-intents unambiguously, especially when sending more than one command, the delimiters have to be placed appropriately. Table xx shows in pseudo-code how messages are constructed. The table columns separate

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | start signal | command 1 | instruction delimiter | instruction 11 | instruction delimiter |
| ⇨ | instruction 12 | instruction delimiter | instruction 13 | command delimiter | command 2 |
| ⇨ | command delimiter | command 3 | instruction delimiter | instruction 31 | stop signal |
| **Tab. 1** General command syntax of communication protocol implemented by I2CDevice class. Pseudo-code reads from left to right in descending row order. | | | | | |

Actual example with the structure given in Tab. 1 for the delimiter set specified in table xx.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \* | CIRC | + | 4 | + | 5 | + | 2 | ; | STATUS | ; | NA | + | 4 | # |

Any characters sent prior to the start-signal or after the stop-signal are ignored, thus messages without start-signal are considered as empty by the communication protocol. If the stop signal is missing the message is treated as corrupted and no command is executed even if it is a valid command apart from the missing stopsignal. Since start and stopsignal are obligatory, the effectively useable maximum message length reduces to 30 characters.

Python-Implementation

This chapter will guide through the interface implementation on Python side (resp. Raspberry Pi, see Fig. 1). After shortly introducing all variables and methods of the I2CDevice class, it will give an example on how to use class-instances as common user.

Static Variables

The term static variable denotes that a class variable is shared over all instances of this specific class. Therefor changing static class variables will affect all instances of this specific class. It is obvious that parameters like the start signal should not be set individually for each bus-subscriber.

|  |  |  |  |
| --- | --- | --- | --- |
| **Static Variable** | **Type** | **Value (default)** | **Description** |
| clock\_frequency | int | 100000 | Specifies at which frequency in kHz the Serial Clock Lane (SCL) of the I2C-Bus is driven. Changing the value of this variable will **not** affect the actual operating speed, which typically has to be set on the device itself. For example for the RaspberryPi in */boot/config.txt* with *dtparam=i2c\_arm\_baudrate=100000* and only takes effect after a reboot. The variable is only used to specify timings as fractions of the bus speed (e.g. certain delays). 100000 kHz is I2C specification standard and operating standard of the Raspberry Pi. However, I2C High Speed Mode provides up to 400000 kHz. |
| max\_msg\_len | int | 32 | Sets the maximum length of the data block which is sent and expected to receive on call of *write\_i2c\_block\_data()* and *read\_i2c\_block\_data()* 32 Byte-characters (28 Bit) is I2C-Specification maximum per transmission. Exceeding this limit results in an error. |
| delim\_strt | string | \* | Sets the delimiter to be recognized as the begin of an user message. |
| delim\_stop | string | # | Sets the delimiter to be recognized as the end of an user message. |
| delim\_cmds | string | ; | Sets the delimiter to be recognized as the separator between commands of an user message in case multiple *commands* are sent in one block. |
| delim\_inst | string | + | Sets the delimiter to be recognized as the separator between *instructions* of an user message in case multiple commands are sent in one block. |
| delim\_tbc | string | … | Sets the delimiter to be recognized as query of the slave to the master indicating to trigger another *requestEvent()* in order to send more data than the maximum block size defined in *max\_msg\_len* permits. |
| busy\_msg | string | BUSY | Character sequence which signifies that the slave cannot process the received command due to currently performing operations. |
| com\_cmds | dict | {"STATUS": "STATUS", "LOGOFF": "LOGOFF", "NAME": "NAME"} | Convention of common communication commands for which all bus-subscribers should respond to. |

|  |  |  |  |
| --- | --- | --- | --- |
| busy | bool | False | Bool indicating that a busy message was received recently from slave and which controls execution of further actions triggered meanwhile. |
| n\_retry | int | 10 | Integer specifying how often the instance should retry to send the same message in intervals of *wait()* after “busy”-message was received. |

4.2 Instance Variables

|  |  |  |  |
| --- | --- | --- | --- |
| **Instance Variable** | **Type** | **Initial Value** | **Description** |
| name | string | “unknown” | Holds identifying string the slave device reported after *announce()*. |
| address | int | [none] | Address of the bus-subscriber. Specified by user on object-initialization. |
| cmdlist | stringlist | [none] | Holds possible commands and the corresponding number of arguments the slave device reported after *announce()*. |
| inBuffer | string | [none] | Holds incoming messages the slave sent after querying requestEvent() |
| outBuffer | string | [none] | Holds outgoing messages supposed to send to the slave. Is always argument to call of *write\_i2c\_block\_data()*. The user has to take care by himself that outBuffer does not exceed the maximum allowed message length (with all delimiters) in the current implementation. |
| registered | bool | False | Indicates that the handshake in announce was successful and, in case the slave did send appropriate data, that *name* and *cmdlist* are available. Does also affect the timing in *wait()*. |
| nticks\_wait | int | 100 | Specifies how long to wait as fraction of the clock\_frequency on call of *wait()*. |
| n\_retry | int | 10 | Specifies the number of attempts to trigger an requestEvent() again if a “busy”-message was received in response. Is decremented in requestEvent(). Does not affect *sendEvent()*. |
|  |  |  |  |

Instance Methods

Instance Methods are class functions that perform operations on instances of this specific class. Therefor a class instance must be passed on call.

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Arguments** | **Returns** | **Description** |
| \_\_init\_\_() | address: int16 | void | Constructor; Creates I2C-object.  Expects identifying address assigned to the slave connected to the I2C-Bus. Argument *address* has to be provided as hex-input, e.g. 3110 = 0x1F |
| requestEvent() | void | msg: string | Queries response from slave at address assigned to instance via smbus library, delimits it and translates the ASCII-response into a human-readable string given that the slave responds with a string message. Maximum number of characters to receive per call is 32. However, it effectively bypasses this limit, through subsequent calls of itself, if signalled by slave with *delim\_tbc* (see chapter xx). Calls *read\_i2c\_block\_data()* of *smbus* library. |
| sendEvent() | msg: string |  | Translates string message specified by *msg* into ASCII and sends it via *smbus* library to address assigned to instance. Maximum number of characters to transfer per call is 32. Calls *write\_i2c\_block\_data()* of *smbus* library. |
| extractCommands() | msg: string | void | Separates commands and instructions from string in msg according to the delimiters specified in *I2CDevice* class header and stores them in variable *cmdlist* of the instance. |
| availableCommands() | void | void | Prints a list of available commands of the device assigned to instance after *announce()* has been successfully run and instance is in *registered* state. |
| clearInBuffer() | void | void | Clears out the buffer for incoming messages. |
| clearOutBuffer() | void | void | Clears out the buffer for outgoing messages. |
| isEmpty() | msg: bytelist |  | Expects a list of ASCII characters as byte in range from 0 to 255 and checks whether the bytelist contains any message at all or any message beside start and stop signal. It is called by *requestEvent()* prior to translation in order to check whether the slave response has failed. |
| queryStatus() | void | msg: string | Sends “status”-command to the slave. (see chapter xx) |
| queryName() | void | msg: string | Sends “name”-command to the slave. (see chapter xx) |
| announce() | void | void | Implements a handshake protocol between devices connected to the bus. Within a sequence of *requestEvent()* status, name, commands and instructions provided by the device at the instances address are queried and assigned to the instance. A successful handshake results in a *registered* status on each side. To ensure a definite synchronous state between slave and master, announce will *logoff()* the slave previous to the handshake if the slave reports unexpectedly *registered* status on call (see chapter xx). Thus, *announce()* should be called only once per application runtime. |
| logoff() | void | void | Sends “logoff”-command to the slave and results in a false registered state. (see chapter xx) |
| wait() | void | void | Introduces a specific but dynamically adjustable delay dependent from settings in instance variable *nticks\_wait* and static variable *clock\_frequency*. |
|  |  |  |  |
|  |  |  |  |

4.4. Initialization of an I2C Instance

4.5 Calling Syntax

4.6 Minimal Working Example

4.7. Drawbacks

The usage of the python method time.sleep() to control timings of the I2C-Communication turned out to be ill-advised since each sleep will cause the entire fluidiscopeApp to be unresponsive for the duration of the call. Although the introduced sleeps are small (milliseconds), they are most probably the reason of major responsiveness lacks, which appear to occur out of context while using the touch-UI of the App. The primary cause for the occasionally unresponsive UI was only observed and has not been investigated yet. Thus, there might be another or different reason for the lacking responsiveness. However, the usage of sleeps should be replaced by appropriate event schedulers that do not introduce blocks, as soon as possible.

Arduino-Implementation

Variables

Functions

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Arguments** | **Returns** | **Description** |
| setup() | void | void | Built-in Arduino function required to be present in each \*.ino-file. Supposed to initialize variables if needed. Might be left empty of |
| loop() | void | void | Built-in Arduino function required to be present in each \*.ino-file. This function loops everything inside it infinitely, therefor every user operation has to have its origin inside this loop (except interrupts). Subsequent function calls are executed in a regular processing chain. |
| announce() | void | void | Pushes *commandset* and *instructs* according to general syntax into *sendBuffer*. |
| requestEvent() | void | void | Callback function of Wire-Library working as interrupt, that is *requestEvent()* is always called when a *requestEvent()* was triggered by the master. Only function which is effectively able to send a response to the master. Does handle the “busy”-notification to the master. (cf. chapter xx) |
| receiveEvent() | bytecount: int | void | Callback function of Wire-Library working as interrupt, that is *receiveEvent()* is always called when a *sendEvent()* was triggered by the master. Pushes received messages into *receiveBuffer*. |
| separateCommand() | void | count: int | Handles the splitting of the message which is currently inside receiveBuffer according to general command syntax. In particular, the receiveBuffer is scanned successively for delimiters. One command at a time is stored in variable *CMD* while the corresponding instructions are pushed back into array *INST*. [separateCommand() processes data until first occurrence delim\_cmds] |
| executeCommand() | nInst: int | void | Does matching of read-in command with commands known from predefined *commandset*. If successfully matched, it executes any user defined behaviour inside of the condition block. executeCommand() is the **only function** in the blueprint-file which is supposed **to be edited by the user** in order to implement custom behaviour. |
| shiftOutBuffer() | shiftLength: int | void | Deletes the number of characters specified by *shiftLength* from the beginning of *outBuffer* and shifts the remaining characters to the beginning. Called in case the message to deliver inside outBuffer exceeds the I2C maximum number of characters per transmission. |
| shiftInBuffer() | shiftLength: int |  |  |
| prepareSend() | void | void | Copies *outBuffer* into the *sendBuffer* and formats the message in *outBuffer* with start- and stopsignal. Depending on the message length in *outBuffer* it inserts *delim\_tbc* and manages the queue of subsequent outgoing messages until the *outBuffer* is empty. |
| cleanUpReceive() | void | void | Clears *receiveBuffer*. |
| numberOfSends() | void | n: int | Determines how many characters are currently in the *outBuffer* and calculates the number of required transfers to send the whole content dependent from *max\_msg\_len*. Is called by *prepareSend()* |
|  |  |  |  |

Preparation

Creation and Integration of Custom Classes

Example

Drawbacks

Although the usage of memory fragmenting (Arduino-)Strings was deliberately avoided in favour of C-Strings, the memory utilization on Arduino-side can most likely be improved, since a lot of character buffers with partially redundant content were introduced in order to manage incoming and outgoing messages. However, the focus was and will remain on readability and easy realization of user functionalities, even at the expense of slight performance hits.