Raspbudi Documentation

Release

Michael Fellner, Maximilian Götsch

Oct 31, 2017

CONTENTS:

CHAPTER

ONE

MODULE RASPBUDI

class raspbudi.raspbudi_server.**RaspbudiServer**(*port=50000*, *DBG=False*)
This class implements a server to run on the Raspberry. This server then listens for incoming connections and commands and executes them.

dataReceived(*data*)
This function handles received data and passes it on to the execute function.

execute(*data*)
This function tries to execute the command given by data. If it succeeds, it returns the value that has been set to the client. On failure it returns an error message. data is expected to be a list containing device address, device type, command and command arguments, separated by commas.

run()
This function is the main loop of the server. It listens for connections and accepts commands from clients.

class raspbudi.raspbudi_client.**RaspbudiClient**(*config_file=None*)
This class is used to communicate with a Raspberry Pi on which the raspbudi server is running.

read_config(*filename*)
Parses the config file 'filename' to get the raspberry pi ip-addresses.

send_command(*rpi_id*, *command*)
Connects to the specified Raspberry and sends the command string.

1.1 Module raspbudi.bus

class raspbudi.bus.bussystem.**BusSystem**(*name=None*, *DEBUG=False*)
This class represents the whole bus system, containing the bus driver and the available modules. The available modules are determined by the dict provided_modules in the __init__.py script of the modules subpackage.

execute_data(*data_array*)
This function tries to execute the given command with the right module driver.

class raspbudi.bus.busdriver.**BusDriver**
Provides a base class for a bus driver.

class raspbudi.bus.busdriver.**DummyBusDriver**
This class provides a fallback dummy bus driver which just outputs the command to the logfile instead of sending it somewhere.

class raspbudi.bus.busdriver.**RaspberryBusDriver**
Implements a bus driver that uses the Raspberry Pi GPIO pins to control the bus.

send_command(*command*)
Sends the command (which is a 24 bit number) on the bus and then sends the strobe signal.

send_command_stack(*command_stack*)
Sends the commands in command_stack one after another.

set_gpio(*pinnumber*, *val*)
Sets the GPIO pin pinnumber to val

1.2 Module raspbudi.modules

class raspbudi.modules.busmodule.**BusModule**(*bussystem*, *initialize=False*)
Base class for module drivers.

get_commands()
Returns a list of the commands that are allowed to be executed by external programs.

send_commands(*command_list*)
Sends command_list one after another on the bus. To send a single command, provide this method with a list containing only this command.

class raspbudi.modules.dds.**DDS**(*bussystem*, *initialize=False*, *clock_frequency=800*)
Base class for DDS modules.

class raspbudi.modules.dds9858.**DDS9858**(*bussystem*, *initialize=False*, *clock_frequency=800*)
Driver for the DDS9858 module. It provides functions to initialize and reset the module, and to set the frequency, phase and attenuation of it.

calculate_attenuation_word(*attenuation*)
Calculates the binary representation of the given attenuation in respect to the resolution of the device.

calculate_frequency_tuning_word(*frequency*)
Calculates the binary representation of the given frequency in respect to the resolution of the device.

calculate_phase_offset_word(*phase*)
Calculates the binary representation of the given phase offset in respect to the resolution of the device.

get_real_attenuation(*attenuation_word*)
Calculates the actual attenuation from the binary representation in respect to the resolution of the device.

get_real_frequency(*frequency_tuning_word*)
Calculates the actual frequency from the binary representation in respect to the resolution of the device.

get_real_phase_offset(*phase_offset_word*)
Calculates the actual phase offset from the binary respresentation in respect to the resolution of the device.

initialize(*address*)
Initializes the device. This also sets frequency, phase and attenuation to zero.

reset(*address*)
Resets the device. This also sets frequency, phase and attenuation to zero.

set_attenuation(*address*, *attenuation*)
Set the attenuation of the device to attenuation (in dB).

set_frequency(*address*, *frequency*)
Set the output frequency of the device to frequency (in MHz).

set_phase(*address*, *phase*)
Set the phase offset of the device to phase (in degree).

set_registers(*address*, *registers*, *value*, *flags=0*)
Sends value to the specified registers of the module at address. registers is a list of addresses to the registers,

and value is splitted into bytes which are sent to the registers from least significant to most significant byte. The flags parameter allows additional bits to be set (it is OR'd to the command).

class raspbudi.modules.dds9854.**DDS9854**(*bussystem*, *initialize=False*, *clock_frequency=400*)
Driver for the DDS9854 module. It provides functions to initialize and reset the module, and to set the frequency and the phase offset of the device. Also, the output amplitude for each output channel can be set individually.

calculate_amplitude_word(*amplitude*)
Calculates the binary representation of the given amplitude in respect to the resolution of the device.

calculate_frequency_tuning_word(*frequency*)
Calculates hte binary representation of the given frequency in respect to the resolution of the device.

calculate_phase_offset_word(*phase*)
Calculates the binary representation of the given phase offset in respect to the resolution of the device.

get_real_amplitude(*amplitude_word*)
Calculates the actual amplitude from the binary representation in respect to the resolution of the device.

get_real_frequency(*frequency_tuning_word*)
Calculates the actual frequency from the binary representation in respect to the resolution of the device.

get_real_phase_offset(*phase_offset_word*)
Calculates the actual phase offset from the binary representation in respect to the resolution of the device.

initialize(*address*)
Initialize the device. This also sets frequency, phase and amplitude to zero.

reset(*address*)
Resets the device. This also sets frequency, phase and amplitude to zero.

set_amplitude_cosine(*address*, *amplitude*)
Set the amplitude of the cosine output channel of the device to amplitude (in %).

set_amplitude_sine(*address*, *amplitude*)
Set the amplitude of the sine output channel of the device to amplitude (in %).

set_frequency(*address*, *frequency*)
Set the output frequency of the device to frequency (in MHz).

set_phase(*address*, *phase*)
Set the phase offset of the device to phase (in degree).

set_registers(*address*, *registers*, *value*, *flags=0*)
Sends value to the specified registers of the module at address. registers is a list of addresses to the registers, and value is splitted into bytes which are sent to the registers from least significant to most significant byte. The flags parameter allows additional bits to be set (it is OR'd to the command).

class raspbudi.modules.dac.**DAC**(*bussystem*, *initialize=False*)
Base class for DAC modules.

class raspbudi.modules.dac7744e.**DAC7744E**(*bussystem*, *initialize=False*)
Driver for the DAC7744E module. It provides functions to initialize the module and to set each of the four output voltages individually (in %).

calculate_real_voltage(*N*)
Converts the digital two byte word to the actual voltage.

calculate_voltage_word(*voltage*)
Converts the user input to the appropriate two byte word for setting the voltage.

initialize(*address*)
Sets all voltages to 50 %.

set_voltage(*address*, *address_mode*, *N*)
The method set_voltage is meant for internal use only to send the appropriate command for setting the voltage of one of the four outputs. Users of this class should use the wrapper methods set_voltage_A/B/C/D.

set_voltage_A(*address*, *voltage*)
Sets the voltage of the first output to voltage (in %).

set_voltage_B(*address*, *voltage*)
Sets the voltage of the second output to voltage (in %).

set_voltage_C(*address*, *voltage*)
Sets the voltage of the third output to voltage (in %).

set_voltage_D(*address*, *voltage*)
Sets the voltage of the fourth output to voltage (in %).

CHAPTER

TWO

INTERFACES

2.1 Command line Interface

class cli.raspbudi_cli.**RaspbudiCLI**(*config_file=None*)

This class provides a simple command line interface to control the raspbudi server.

run()

This is the main loop of the command line interface. It accepts the user input and forwards the commands to the Raspberrys.

2.2 Webinterface

class webinterface.flask_client.**FlaskClient**

The FlaskClient class extends the RaspbudiClient to support all needed functions the Flask server needs.

get_channel_list()

This function returns a string of the form <channel1>,<type1>;<channel2>,<type2>;... This is needed to provide the webapp with the channel information.

get_channel_values()

This function returns a string of the form <channel1>,<value1>;<channel2>,<value2>;... This is needed to provide the webapp with the current channel values.

initialize_channel(*channel_id*)

This function should be called if a channel is initialized. It then sends the appropriate command to the Raspberry and resets the stored value.

parse_config_file(*filename='/home/max/raspbudi/config/paths.cfg'*, *section='webinterface'*)

Reads the main config file 'config/paths.cfg' to retrieve the paths to the other needed config files.

read_settings(*filename*)

Parses the webinterface settings file specified in 'config/paths.cfg' to get the channels which are present to control.

update_channel(*channel_id*, *value*)

This function should be called if the value of a channel is changed in the webapp. It takes care of sending the appropriate command to the Raspberry and of storing the current vavlue of the channel.

2.3 TRICS

class trics.trics_configparser.**TRICSConfigParser**(*config_file=None*)

This class parses a channel settings file stored by the trics laboratory software and extracts information needed for the raspbudi UserAPI script to identify the channels it has to control.

create_device_info(*channel_args*, *additional_data*)

This function creates a dictionary containing the Raspberry ID and the command to be sent if the channel is accessed. One has to append the value which the channel should be set to the command.

parse_additional_data(*data*)

This function creates a dict from the additionalData field of the settings file. In this field the key-value pairs are separated by a semicolon, and key and value are separated by a comma.

read(*filename*)

This function reads in a channel settings file and organizes its information into a dict containing the information necessary for the UserAPI script to control the raspbudi channels.

PYTHON MODULE INDEX

INDEX

P

R

S

T

U

W