RASPBUDI – Project Documentation

Michael Fellner, August 2017

reviewed by Maximilian Götsch, October 2017

1. *Introduction*

RASPBUDI is a python project, which implements a laboratory bus driver on a Raspberry Pi in order to control various modules such as direct digital synthesizers (DDS) or Digital to Analog Converters (DAC) via Ethernet. Therewith, the 50-pin cable connectors for the laboratory bus interface doesn't have to be drawn via long distances, as Ethernet cables or even wireless LAN can be used. The RPi is connected to the lab network and listens for commands on the Ethernet port. Hereby, multiple clients are allowed. In addition, the RPi is connected to the bus system via 25 GPIO pins (1 strobe bit, 8 address bits and 16 data bits) and GND. With this setup, the RPi is able to initialize and control a DDS on the bus system according to commands given via Ethernet connection. By now, functions for setting frequency, phase and attenuation of a DDS9858 have been implemented. Concerning the transfer rate, when directly connecting client and server via a router, a rate of ca. 500,000 commands per second can be reached. If there are any obscurities or questions concerning this documentation, please do not hesitate to contact me (Michael.Fellner@student.uibk.ac.at).

2. Specifications

The RASPBUDI software runs on a Raspberry Pi 3, which provides 26 GPIO-ports, 5 of which are used. It enables to control the parallel laboratory bus interface via Ethernet connection (either via wireless LAN or Ethernet cable). By now, the clock frequency of DDS9858 has been set to 800 MHz and for the DDS9854 to 400 MHz, which can be changed in the code (see below). Power supply for the RPi is provided via USB, for common use of the device, it might also be supplied via the 5V Pins and the bus system (BE AWARE: Those pins are not protected by capacitors or a current limiter!). The bus interface, and consequently the modules, are supplied by a 5V laboratory power supply (Current when using two DDS9858 modules: ca. 1.2A).

3. Hardware Setup

The Raspberry Pi must be connected to the bus interface via a self-made adapter module, which enables to connect the RPi's 40-pin header with

the 50-pin header of the bus interface. The RPi's pin configuration can be easily looked up on the internet (e. g. https://de.pinout.xyz/#); the pinout of the bus interface is provided in the file *docs/bus_system.pdf* on page 3. The pin mapping, i. e. which GPIO port connects to which control bit, is as follows:

STROBE – GPIO3	D04 – GPIO1	D09 – GPIO10	D14 – GPIO7	A3 – GPIO16
D00 – GPIO14	D05 – GPIO27	D10 – GPIO25	D15 – GPIO5	A4 – GPIO19
D01 – GPIO4	D06 – GPIO23	D11 – GPIO9	A0 – GPIO12	A5 – GPIO20
D02 – GPIO15	D07 – GPIO22	D12 – GPIO8	A1 – GPIO6	A6 – GPIO26
D03 – GPIO18	D08 – GPIO24	D13 – GPIO11	A2 – GPIO13	A7 – GPIO21

4. *Software Setup*

All necessary software configuration can be done by the files in the "*config*" directory. For detailed information see the comments in the config files.

4.1. The Raspbudi Server

The Raspbudi server runs on the Raspberry Pi and listens for commands. If a valid command is received, it sends the appropriate commands on the bus it is connected to, and then returns the value which has been set. On failure it returns an error message.

4.2. *Command Line Interface*

A simple command line interface is provided by the "*raspbudi_cli.py*" script. A starter is provided by the script "*cli.sh*" in the main directory. This interface accepts commands of the form *<raspberryID>:<address>,<type>,<command>,<value>* and sends the command to the corresponding raspberry.

4.3. *Webinterface*

4.3.1. Webinterface Server

The webinterface server provides clients with the browser interface and handles the AJAX-requests from this interface. It is based on the Flask-Framework. Flask is a web framework written in python, that can handle AJAX-requests, in our usage, via the POST-method. At the moment, the server runs on port 5000 and is provided in the script "*webinterface_server.py*" in the "*webinterface*" directory. To get it run, the flask library has to be installed by running the following commands on the command-line:

```
$ sudo apt-get install python-virtualenv
```

```
$ pip install Flask
```

For a detailed installation guide see http://flask.pocoo.org/docs/0.12/installation/.

With AJAX-requests, it is often necessary to allow cross origin requests (CORS). In order to do so, run

```
$ pip install -U flask-cors
```

to install the corresponding packages. To start the flask-server it is recommended to use the provided "*webinterface_server.sh*" script in the main directory.

4.3.2. Webapp

As the server provides AJAX-requests, the webapp is just a webpage including JavaScript. It uses the *rangeslider.js* library, which has to be included as well as the jQuery library. Installing *those libraries* can be done by running

```
$ npm install –save rangeslider.js
$ npm install jquery
```

in the file directory if they are not already present. The webapp is provided by the file "*webinterface.html*". This interface is delivered by the webinterface server if a client connects to the server via the webbrowser on port 5000.

On startup, this webpage requests a list of all modules from the server, and lists those modules in a table, including sliders and textfields to change the attributes of the module. Whenever a slider is moved or a textfield value is changed (onfocusout()), the new value will be sent to the server via another AJAX-request.

The code is basically straight-forward JavaScript, apart from a few things. For information about the source code, please see the source code comments.

4.4. *TRICS Client*

To be able to use the raspbudi framework with the laboratory software TRICS, a client has been created which provides communication between TRICS and Raspbudi. This client is located in the "*trics*" folder and is called "*trics_client.py*". It is a UserAPI script which can be imported in the TRICS system. This script receives a message if the value of any channel has changed. If the channel corresponds to a Raspbudi device, the appropriate command is sent to the right Raspberry Pi.

To obtain the list of all channels the script reads the TRICS settings file (the path to it is specified in the "*config/paths.cfg*" file), and parses it to filter all DDS and DAC channels. The Raspberry Pi IP-addresses are specified in the "*config*" directory too.

4.5. Software structure

At the moment, *RASPBUDI* consists of several classes, which will be shortly discussed in the following:

BusModule:

An object of the class BusModule represents any module that can be controlled via the software provided here. It has the attributes "bussystem", which specifies the bus system the module is attached to (see below), and "*command_list*", which is a list containing the commands that are allowed to be executed from a remote client. Functions not listed here, cannot be executed via the raspbudi client as access to them is denied by the bus interface due to safety reasons. The method "*get_commands()*", which is also implemented in BusModule, returns this list.

BusSystem:

An object of the class BusSystem represents a set of modules that can be controlled via the software provided. This class is used to execute commands on the bus, utilizing an appropriate BusDriver (see blow) and the available Module Drivers, which are specified in the *module* subpackage and loaded automatically. It provides the functions:

- *execute_data(data_array):* Executes the data specified by *data_array* on the bus. *data_array[0]* must contain the address of the device, *data_array[1]* the driver to use, *data_array[2]* the method of the driver to be executed and *data_array[3:]* additional arguments that are passed to this method.

BusDriver:

The bus driver provides methods to communicate with a module in a bus system. Here, the pin mapping between the RPi and the bus interface needs to be specified. By now, one specific bus driver, RaspberryBusDriver, is implemented. It provides the following functions:

- *set_gpio(pinnumber,val)*: Sets the specified GPIO-Port of the RPi to the value *val,* which must be *true* or *false*.
- *send_command(command)*: Sends a command (which is a 24-bit (binary) number) to the module. The module's address is stored in

the *command* argument (it is represented by the first 8 bits), which is generated in the implementation of the specified module. This includes setting the corresponding bits and sending the strobe signal. Before setting the strobe bit, a delay, the duration of which can be specified by changing the variable "*DataReadyTimeout*", is triggered.
- *send_command_stack(command_stack):* Requires a list as argument, sends the commands provided in the list one after another.

DDS:

The DDS class is inhering from BusModule. The classes DDS9858 and DDS9854 (which inherit from DDS), are implemented and provide the following functions to be called from other classes:

- *initialize(address)*: Initializes the DDS at *address* by sending the corresponding commands via the bus driver.
- *reset(address)*: Resets the DDS at *address*.
- *set_frequency(address, frequency)*: Sets the frequency of the specified DDS to the value *frequency.* The unit of frequency is MHz. The frequency must not exceed half the clock frequency (At the moment, clock frequency is set to 800 MHz for DDS9858 and to 400 MHz for DDS9854, this can be changed in the source code).
- *set_phase(address, phase)*: Sets the frequency of the specified DDS to the value *phase.* The unit of *phase* is degree.
- *set_attenuation(address, attenuation)*^{DDS9858}: Sets the frequency of the specified DDS to the value *attenuation.* The unit of *attenuation* is dB. Maximum attenuation is 31.75dB.
- *set_amplitude_sine(address, amplitude)*^{DDS9854}: Sets the amplitude of the sine output of the specified DDS in percent.
- *set_amplitude_cosine(address, amplitude)*^{DDS9854}: Sets the amplitude of the cosine output of the specified DDS in percent.

The set-functions calculate the binary command(s) necessary to pass the desired settings to the module. They must have a return value, if the module is controlled via Ethernet, as the return value of the method will be passed to the Ethernet client as server's response. For those functions, that set some attribute of the DDS (e. g. frequency), the return value contains the value, the attribute has actually been set to; this real value is returned by the functions "*get_real_<attribute>*", which is not meant to be called from other classes as well as any functions not listed above. This is[1] necessary, because rounding errors are likely to occur due to the limiting precision when converting the data to the binary key. To calculate the 8-bit word, which must be sent to the module, the functions use the methods "*calculate_frequency_tuning_word(frequency)*", "*calculate_phase_offset_word(phase)*" and

“calculate_attenuation_word(attenuation)”, which convert the input to corresponding 8-bit value, as given in the datasheet of the corresponding DDS.

DAC:

The DAC class inherits from BusModule and is the base class for Digital-to-Analog Converters. The DAC7744E class inherits from it and provides control for a DAC7744E device. It implements the methods:

- *set_voltage_A(address, voltage)*: Sets the voltage of the first output of the DAC to *voltage*. For the other outputs there exist similar functions, named from A to D.
- *initialize(address)*: Sets all outputs to zero voltage.