Python

Release

February 07, 2015

Contents

This documentation covers the PBox experimental control system that is currently used at University of Innsbruck. The two most relevant repositories are hosted on github. For specific documentation refer to the README file of the corresponding repository:

The sequencer2 compiler

https://github.com/pschindler/sequencer2

The VHDL code for the DDS board

https://github.com/pschindler/vhdl-ad9910

The following gives a quick tour to the Hardware and Software options of the various versions of the PBox.

CHAPTER 1

The MainBoard:

1.1 Introduction to the PBox

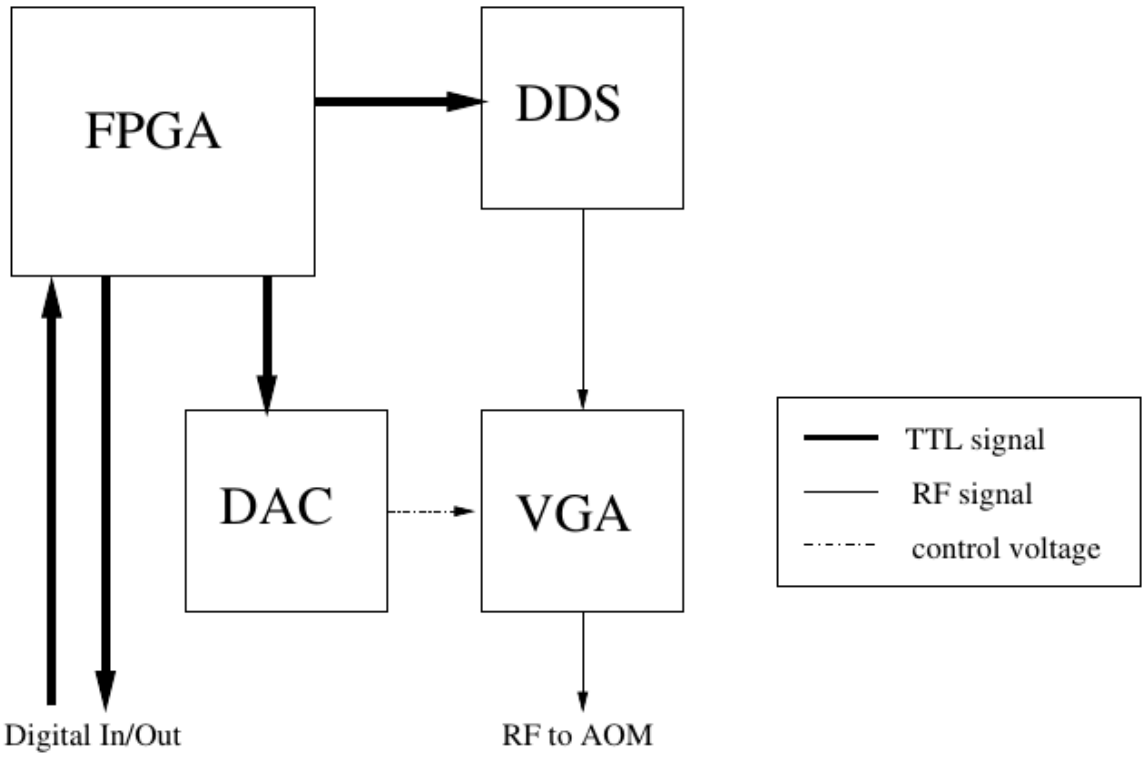
1.1.1 Basic working principle

The controls in an experiment are divided into synchronous and asynchronous signals.

Synchronous	Determinstcally switched in one experiment cycle
Asynchronous	Switched between two experiments (may be varied in a scan)

The Box is responsible for all synchronous signals in the experiment but can also generate asynchronous signals. The signal can be either digital (3.3V TTL) or analog radio frequency (RF) pulses. It features a special purpose microcontroller realized in an field programmable gate array (FPGA). This FPGA generates the digital signal and controls the direct digital synthesizers (DDS) that generate the RF pulses. The FPGA is connected to the experimental control computer (PC) with a standard ethernet connection.

Software wise, the PBox is controlled by a program called “sequencer2”. It is written in the python programming language. Note that a running sequencer2 version is required to perform the basic testing of the PBox hardware.



1.1.2 Overview over the Hardware

The new version of the PBox is built up from 5 main components:

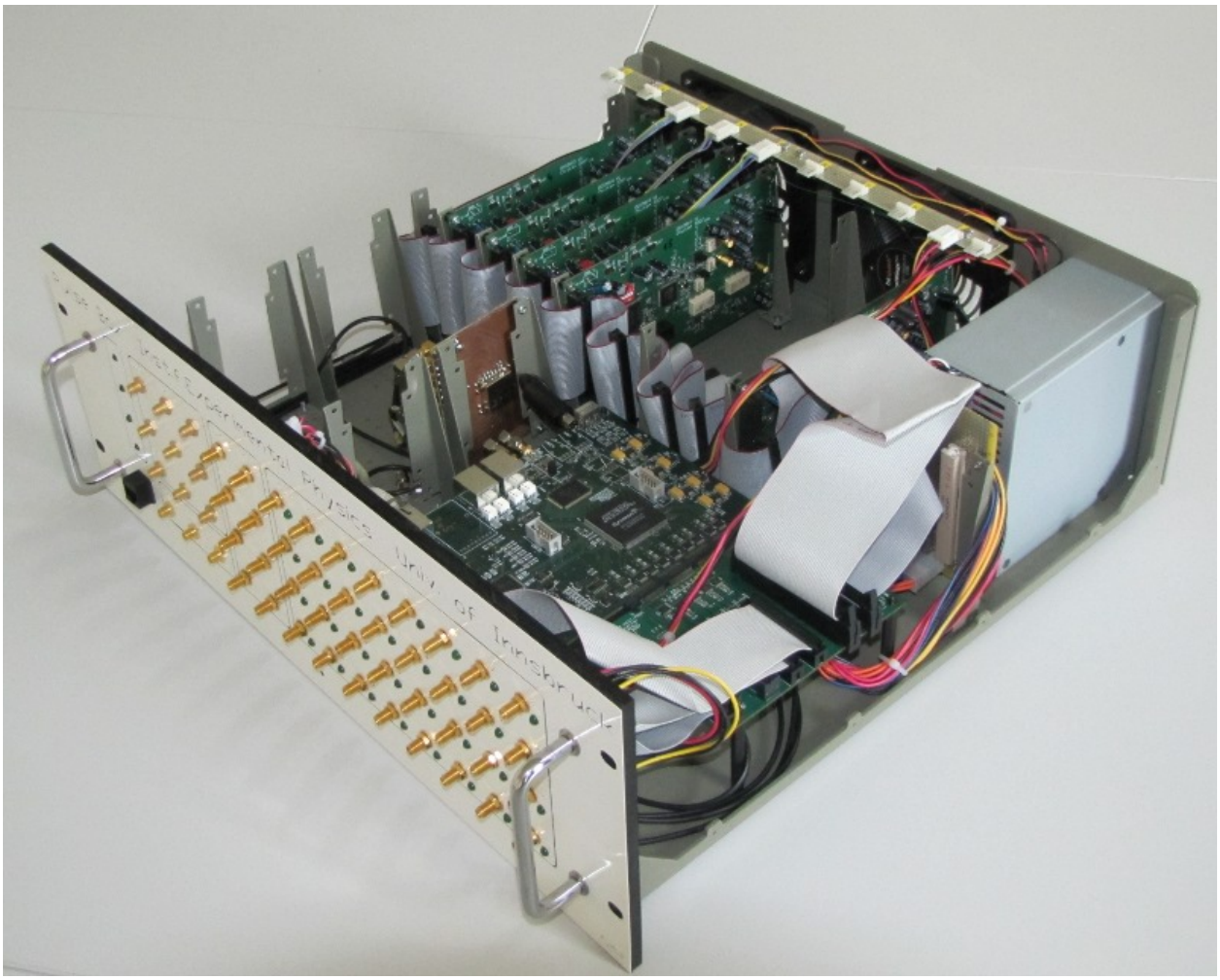
Mainboard	The main FPGA and the network connection
Synthesizer board	Auxiliary FPGA and DDS
Breakout board	Logic level converter and connectors
Digital IO board	Galvanically isolated connectors
Clock generator	Synchronizing all parts

1.2 Which version of the Box do I have?

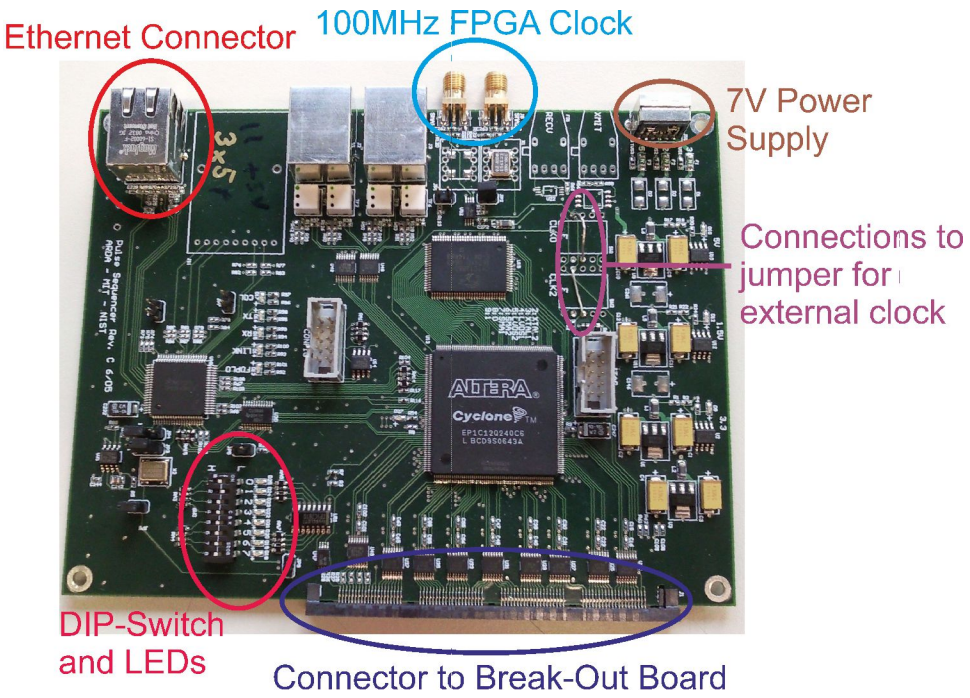
There exist two versions of the PBox. Their main difference that they use different RF-generation hardware (DDS). The mainboard is identical in hardware and firmware.

The versions can be distinguished by checking the DDS PCB boards. The old version uses an analog devices evaluation baord which has a blue PCB. The new version uses custom made DDS boards with a green PCB.

In general, the distinction can be made by looking at the cases. An image of the new version:



1.3 The MainBoard



1.3.1 100MHz FPGA Clock

Our experimental setups require the FPGA to operate at 100MHz. And the ethernet connection operates at 100MHz, too. There is a 100MHz quartz on the PCB. In standard configuration, one clock is set to external and one to the quartz on the PCB. Hence, one needs to solder the connections, so that they are both running at the external 100MHz clock because the external clock is most likely much more accurate.

In the picture of the main board, one can see the "Connections to jumper for external clock". This is the way, it must look like. If one has to solder something there, use the wires that are already on the PCB. If the wires are to short, do not use a flexible wire (=litz wire)!!! Use, for example, a the connection wire of a resistor instead!

1.3.2 7V Power Supply

This PCB has a 5V voltage regulator on it. Hence, the input voltage has to be at least 5.8V. It's recommended to use 7V as input voltage. The current consumption of the main board can be estimated with about 1A.

Pin 1 of the DIP Switch is the reset. So, in order to invoke a reset, just move the Switch 1 to one other side and then back. When the main board is powered up, or reseted, the LEDs next to the DIP Switch will start to blink for some time. If they don't blink but the 3 power LEDs are on, there will be something wrong with the clock.

1.3.3 Ethernet Connector

This system is programed through an ethernet connection with a computer. Unfortunately, the ethernet client is programmed very poorly. Hence, one has to take some things into account, when setting up the connection.

The pin 8 of the DIP Switch determines whether the ethernet client is running in DHCP mode or not. IT MUST ALWAYS RUN IN DHCP MODE! Hence, pin 8 must set to off, as shown in the picture. All pulse boxes, or main

boards of the pulse boxes, have the same MAC address (= physical network address in a subnet). This address must be unique in a subnet. So, if one has more than one pulse box connected to one router, one has to change the MAC address to become unique. The pins 2-7 of the DIP switch determine the last 6 bits of the MAC address. In order to work, set all pulse box MAC addresses to unique values. If pins 2-7 are all low, the MAC address of the pulse box is 00:01:ca:22:22:20. Accordingly, if pin 2 is high and the rest low, the MAC address is 00:01:ca:22:22:21, and so on. Since the ethernet client works with DHCP, one has to assign the pulse box a certain IP address in the DHCP server. (The DHCP is usually some part of the router). The IP address must be in the 192.168.0.x subnet and the last number has to be 220 or greater!!! If it is not, it won't work! Since the IP address of the pulse should not change (for convenient operation), log on to your router and assign the MAC address of the router, e.g. 192.168.0.220. (To get the MAC address of the pulse box: If one is logged in at the router, one can see all devices connected to the router. One of them should be the pulse box -> MAC address of the pulse box, or main board of the pulse box.)

1.3.4 Connector to the Break-Out Board

On the main board itself, there are no connectors for our experimental setup. These are all on the break-out board. On the main board, there is just one connector to the break-out board through which all control signals have to go.

1.3.5 Troubleshooting

A first test can be performed by observing the LEDs near the DIP switches after switching on the Box. They should change from one to the next rapidly. If the LEDs take about one second or longer to switch, the clock of the FPGA is not applied.

The configuration of the ehternet can be tested by starting the server of the sequencer2 software. If no ‘‘pulse transfer protocol error‘ is displayed, the ethernet configuration works.

The main sources of errors are:

- The clock is not connected
- The FPGA core clock and the ethernet clock are not the same
- The firmware is not programmed

1.3.6 Reprogramming the firmware

The firmware for the can be found at

http://pulse-sequencer.sourceforge.net/firmware/sequencer_top-v0_29.pof

```
Local copy of the programming file
```

1.4 The PBox on the network

1.4.1 The Network Discovery Problem

The pulse sequencer's Ethernet interface allows flexible decoupling from the control computer. However, the Python sequencer running on the control computer must still determine what sequencer devices exist on the local network, what their addresses are, and which one(s) to address for any given pulse program. These problems and their solution we will collectively call ‘‘discovery’‘.

We will more clearly define what we mean by a ‘‘device address’‘ in the next section, but for now we use it to mean a generic way of distinguishing devices on a network and of sending packets to one or more specific devices.

The current solution to the discovery problem is to hardcode all addresses, both in firmware and in software, so that they agree. This places the burden on a human user to make sure that all sequencer devices on the same network have unique addresses, by setting DIP switch settings on the sequencer boards and writing a software settings file accordingly. While this provides a deterministic solution that works every time, it removes a lot of flexibility promised by having a network interface in the first place.

The main disadvantages are:

- Requiring a separate router to provide a "walled garden" (private subnet) containing the control computer and the sequencer devices. * Although a router may still be desirable in case there are other problems with letting sequencer devices live on your main subnet with Internet access. For example, your main DHCP server may require every device to register its MAC address before it can receive an IP address and before the router will route IP packets for it.
- Requiring both the router and the control computer to be on the same subnet that is hardcoded in the sequencer firmware.
- Having to hardcode the desired device address in the Python software, and only being able to address on sequencer device at a time.

1.4.2 Network Addresses

A sequencer device currently has three different, but related, addresses associated with it.

Below, x represents the 4-bit device ID set by DIP switches on the sequencer circuit board. See *The MainBoard* how to set these switches.

- Ethernet medium access control (MAC) address, of the form 00:01:ca:22:22:2x
- Internet protocol (IP) address, which when statically configured, is of the form 192.168.0.22x
- The device ID for pulse transfer protocol daisy-chain transfer, the same as x.

The device ID and Ethernet MAC address are taken from the DIP switch. The IP address can either be statically configured or dynamically configured, depending on DIP switch 6 (nswitch5 in firmware).

The sequencer always goes through a DHCP handshake process to discover the DHCP server and router on its subnet. However, if it is using a static IP address, it will decline the offered DHCP lease address.

1.4.3 Troubleshooting the network configuration

The interaction between the experiment control computer, the router and the PBox can be investigated with the ethernet packet analzing tool wireshark.

http://www.wireshark.org

Install the tool on the experiment control computer and listen on the interface that is connected to the router. After switching on the PBox, a DHCP request from the PBox's MAC address (00:01:ca:22:22:2x) should be detected. If the router is configured correctly, it should offer the PBox the corresponding IP address (192.168.0.22x).

When the sequencer2 tries to program the PBox it performs an ARP request, asking for the MAC address of the PBox. Then it should send UDP packets to the PBOx. The PBox should respond back.

1.4.4 DHCP Support

Dynamic host configuration protocol allows IP hosts to retrieve network parameters dynamically over the network without user intervention. The sequencer has a limited DHCP client implemented as a firmware module that is compliant with the following IETF RFC:

http://www.ietf.org/rfc/rfc1531.txt?number=1531

However, it has not been fully tested with a wide range of routers. For example, the sequencer is known to work with Linksys routers, but not a Dynex router. The problem is not known, but it is probably due to incomplete or incorrect implementation of the specification on either the client side (our fault) or server side (manufacturer's fault).

The DHCP module currently is able to accept a lease address and a gateway from the DHCP server response. Its main limitations right now are:

- it does not currently store the lease time.
- it does not request a new IP address when its lease time has expired.

Once lease time support is added, it will need to be tested by actually waiting past the lease time and seeing if another DHCP negotiation occurs. A packet sniffer such as Wireshark is useful for this purpose.

Finally, the Python software will need to know how to find the dynamic IP addresses of any sequencers on the network, as described in the next section.

1.4.5 Python Software Support

By default, the Python sequencer (both v1 and v2) assumes it is on the same network as a sequencer with device ID 9 configured on its DIP switches. The Python sequencer settings file contains the hardcoded, statically configured IP address 192.168.0.229 by default.

Once the firmware has been modified to handle DHCP dynamic IP addresses and lease times correctly, the connection still needs to be made between the software and the sequencer.

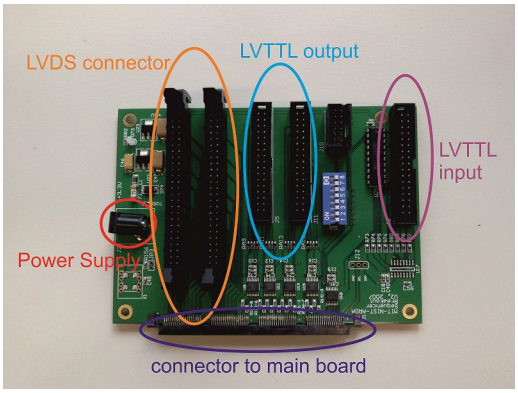
This is currently done in the Python v1 sequencer by sending a PTP Broadcast Request before every pulse program is compiled and run. However, this software has the following limitations:

- the corresponding PTP Broadcast Replies are not collected anywhere, and the results are not stored.
- there is currently no way of getting the lease time remaining from a sequencer device via PTP.
- there is no way to specify in the API which device ID to use, even if multiple devices were detected.
- there is no support for rediscovering sequencer devices whose lease IP addresses have expired.

CHAPTER 2

Additional Hardware

2.1 The BreakoutBoard



The breakout board provides the connectors for the various Bus systems. The LVTTL inouts and outputs are connected to the digital input and output cards, whioch provide a galvanic isolation. The LVDS bus is connected to the DDS devices that provide an analog RF output.

2.1.1 The connectors

The power conenctor needs to be soldered the connector to the PCB. The problem with this connector is that it is very space-consuming in the pulse box. Since along this side of the main and break-out board, there are no other connectors. One other idea would be to solder the connection wires directly to the PCB. The voltage of this PCB is the same as for the main board. That means 7VDC!

The connection to the mainboard is performed by a SAMTEC connector The company that built the 30 break-out boards was not capable of soldering this connector, too. Hence, one has to solder it oneself. Since there are 128 connections to solder, it's kind of a pain. One easier way of doing it is to use solder paste and a heat gun, available at the Kasper-Hauser room or at the IQOQI with Gerhard Hendl. It's very likely that not all connections at soldered perfectly, therefore one needs to check the connections and resolder. On how to check the connections for the individual connectors, please refer to the corresponding section.

2.1.2 Troublehooting

The main reasons for failures on the breakout board are:

- The connector to the mainboard is not soldered correctly A LVDS to

- A LVDS-TTL converter is not soldered correctly
- A LVDS-TTL converter is broken

If the breakout board is working correctly and the LEDs of the output ports do not work when testnig the TTL output channels, the error is located in the digital output board.

The digital out board

2.1.3 LVTTL Input Connectors

A basic test on the 8 trigger inputs is performed by checking the 16 corresponding connections at the main board connector with a continuity tester. One can, of course, repeat the same trick with the voltmeter again but this time in the other direction.

2.1.4 Testing the LVTTL Output Connectors

All pins on the left side (closer to the LVDS connectors) are GND. And the 16 data pins are on the right side starting at the bottom (closer

to the main board connector). One can connect these pins with a test

device, such as the PCBs for the TTL-output signal or a logic analyzer. At this point, Python and the Sequencer are probably already installed and configured. Open a shell, change to the directory of the sequencer and start Python from there. Then type:

```
from tests.test_hardware import *
```

and to test if the connection to the pulse box works, type:

```
ht = HardwareTests()
```

In order to set all TTL output pin to high, type:

```
ht.test_ttl_set()
```

or:

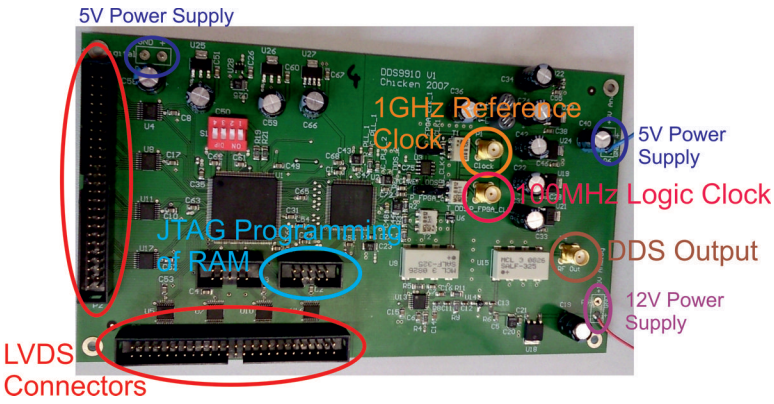
```
ht.test_ttl_set(65535)
```

To set them all to low, type:

```
ht.test_ttl_set(0)
```

With the test device, one should now be able to see if all connections correctly transport LOWs and HIGHs. Since it can be hard to find the right connections from a specific pin number to the pin on the main board connector, there is also the possibility to check the voltages at the resistors next to the main board connector with a voltmeter. After checking some connections, one will easily be able to determine between good and bad connections. (But don't forget this PCB has a top and a bottom side.)

2.2 The DDS board



2.2.1 LVDS connectors

They are used to program the DDS output from the FPGA on the main board. Each DDS needs to have a different number on the bus. The number set at the DIP-Switch, the red box with the 4 switches, represents the number on the bus and is therefore important for programming the DDS output frequency. 5V Power Supplies and 12V Power

2.2.2 Supply

These have to be connected with the 5V, or respectively the 12V, power supply of the pulse box. The current consumption of a DDS board at normal operation can be estimated with about 1A.

Unfortunately, the holes at too small to use standard connectors for theses. Hence, it's the easiest thing to directly solder the wires to the pads. But make sure that when you mount the DDS board, there is no stress on the cable. Otherwise the cable might break after some time, and checking each power supply of every DDS board is probably to first thing one might do when looking for the bug. 1GHz Reference Clock

This is the input of the reference clock signal, and its frequency has to be 1GHz. This frequency is supplied by the Clock Generator Board, or the Clock Distributor Board respectively. 100MHz Logic Clock

This is the input clock for the logic system on the board, and its frequency has to 100MHz. This frequency is supplied by the Clock Generator Board, or the Clock Distributor Board respectively, too.

2.2.3 DDS Output

This is the output connector of the generated frequency. Unfortunately, this output is not galvanicly isolated. Hence, in order to galvanicly isolate all pulse box outputs, one needs to add a transformer between the DDS output and the actual pulse box output.

2.2.4 Programming the FPGA

The FPGA on the DDS board can be programmed with two different protocols:

- JTAG - Temporary programming - Connector J1
- Active serial programming - Permanent programming - Conenctor J2

The JTAG port is used to convert the FPGA into a logic analyzer to test the LVDS bus. The permanent program of the FPGA is programmed via active serial programming. The firmware can be downloaded from the project homepage:

http://sourceforge.net/projects/pulse-sequencer/files/ad9910_firmware/RC1/

2.2.5 Troubleshooting

If the DDS gives a slightly wrong frequency (e.g. 100MHz instead of 80MHz) then the reference frequency in the software is set incorrectly.

If the DDS does not react at all, the LVDS bus needs to be investigated.

Testing the LVDS Connectors

2.3 LVDS Terminators

2.3.1 General description

LVDS is a bus where a current flows. So, if there is no terminator with a resistor, the right amount of current can't flow. Hence, a terminator is mandatory.

The best thing is to connect one side of the cable with the source, e.g. the break-out board, and the other side with the terminator. In between, the DDS boards can be connected to the cable.

2.3.2 Building a terminator

First, one needs to order the PCB at a manufacturer. REMEMBER ONE NEEDS TWO TERMINATORS FOR ONE PULSE BOX! The required PCB file can be found in the folder \anna-c704calcium40ControlProgramsPBox documentslvds-terminator and is called lvds-terminator.PcbDoc. For each PCB, one needs two resistor arrays. The RS-Components order number can be found in the same folder in the file resistor_order.txt. Additionally, one needs a 50-pin-connector for each PCB, too.

When soldering the orientation of the resistors is not import, but the orientation of the 50-pin-connector is important!!! The first pin on the PCB (the only squared pin) has to be on the same side as the single slit (in the middle) of 50-pin-connector.

2.4 The Clock Generator

Im Clock Generator kommt der AD9517-3 zum Einsatz zusammen mit einem 1GHz Oszillator CVCSO-914 von Crystek. Die Ansteuerung erfolgt über einen Attiny25 von Atmel. Eigenschaften des Clock Generators

4 Kanäle bis 1.4 GHz 4 Kanäle bis 800 MHz

Amplitude und Frequenz jeweils paarweise programmierbar

2.4.1 Benutzungshinweise

Der Clockgenerator detektiert die Referenzclock (10MHz) beim Einschalten. Deshalb 10MHz Referenzsignal vor dem Einschalten der Spannungsversorgung anlegen. Programmierung

2.4.2 Benötigt:

Programmiergerät, z.B. AVR-ISP AVR-Studio (kostenlos zum Runterladen nach Registrierung) Quellcodes Clockgenerator mit Spannungsquelle (5V) Idealerweise 10MHz Referenz und Spektrum Analysator um das Ergebnis zu verifizieren

2.4.3 Programmierung:

Quellcodes in AVR-Studio öffnen. Neues Projekt anlegen und Quellcodes einbinden. Gewünschte Anpassungen vornehmen s.u.(hinweise im Quellcode beachten) Kompilieren (F7) (Fehlermeldungen beachten!) Erstellte .hex Datei zum Generator übertragen Gegebenenfalls Generator aus und wieder einschalten, damit neue Einstellungen geladen werden

2.4.4 Funktionsweise

Referenz: Der AD9517-2 verwendet wahlweise ein externes oder internes 10MHz referenz Signal. Referenzdividierer: Das Referenzsignal kann durch einen Referenzdividierer R geteilt werden. (default R=1) Multiplikator: Danach wird es mit einem Multiplikator N=P x B +A multipliziert, der sich aus einem Prescaler P sowie zwei Zählregistern B und A zusammensetzt. VCO: Somit ergibt sich die Frequenz des VCO zu fVCO =fRef*N/R VCO Teiler: Diese Frequenz die typischerweise um 2GHz gewählt wird, wird hernach durch einen VCO Teiler geteilt. Kanalteiler: Der Ausgang des VCO Teilers geht dann an 4 Kanalteiler, die Teilungen von 2 bis 16 ermöglichen und für je 2 Kanäle gelten.

2.5 The digital out board

This board was designed to get rid of ground-loops created by the pulse box. The board has 16 galvanically isolated grounds, each created by a DC-DC-converter. A digital isolator transfers each output signal from the common ground system of the pulse box to the signal's seperate ground system.

The PCBs and parts list can be found on

\Anna-c704sqipPulse BoxDigital Out

Attention: This PCB cannot drive 50 Ohm. The input resistance of each connected device has to be in the order of 1kOhm or higher. So, DO NOT CONNECT 50 Ohm DEVICES TO THE DIGITAL OUTPUTS! It can't provide enough current for 50 Ohms because the drivers are very fast and they have to drive a 5V-TTL signal. Hence, there will be rather big currents. Since we are using 50 Ohm-SMA cables and cannot terminate with 50 Ohms, there will be back-reflections. If you are using a short cable (= low load-capacitance), there may be some voltage overshoot. With a 1m cable, the overshoot can be 2V for 10ns. If that damages your circuits or try to avoid overshoots generally, you have to add a capacitor parallel to the output. To get an idea, what capacitor to use, have a look at the list below.

- no capacitor parallel to the output, 1m coaxial cable, high-Ohm input resistance: overshoot 2V (7V in total), rise time <1ns
- 330pF parallel to the output, 1m coaxial cable, high-Ohm input resistance: overshoot 1V (6V in total, rise time ~9ns
- 1.5nF parallel to the output, 1m coaxial cable, high-Ohm input resistance: hardly any overshoot, rise time ~60ns

2.5.1 Troubleshooting

- The cable to the breakout board is not assembled correctly

- The supply voltage is not applied
- A DC-DC converter is broken

2.6 Digital In board

This board was designed to get rid of ground-loops created by the pulse box. The board has 8 galvanically isolated grounds, each created by a DC-DC-converter. A digital isolator transfers each in signal from its seperate ground system to the common ground system of the pulse box.

The PCBs and parts list can be found on \Anna-c704sqipPulse BoxDigital In .

The PCBs has a high-Ohm input. This does not match 50 Ohm coaxial cable, therefore back-reflections will occur. If you have a device connected to one input channel that can drive 50 Ohm, you can put a 50 Ohm resistor parallel to your input to get rid of the reflections. BUT, if you connect a device that drives 50 Ohm to a regular high-Ohm input, the input channel might be damaged because the input voltage will then be 10V. So, don't do that.

2.6.1 Troubleshooting

- The cable to the breakout board is not assembled correctly
- The supply voltage is not applied
- A DC-DC converter is broken

2.7 Housing for the Box

The SolidWorks files can be found under \Anna-c704sqipPulse BoxSolidWorks. With these files, one can order anywhere. The cheapest company (by much) so far was EGU Metall. Their quote can be found under \Anna-c704sqipPulse BoxSolidWorksQuote. Since the inputs and outputs have to be galvanically isolated and there are isolating SMA connectors, one has to use a non-conducting material for the front panel. For example, one can order 6mm POM (plastic) at "Technische Kunststoffe" and the guys from the mechanical workshop will mill it. Some additional things have to be ordered as well.

> Handles: RS-Components part number 667-0365 Network cable connector for front panel: RS-Components part number 186-3149 Power supply: RS-Components part number 177-804 Fans: if desired, any 12cm fan will do

2.7.1 Description

The main board and the break-out board are mounted above the frequency generator to save room in the case. Up to 10 DDS boards could be mounted in the back (but only 9 connector holes in tqhe panel). The frequency distributors can be mounted in the front or in the back. If additional PCBs have the same size as the frequency distributor (100mm x 70mm), they can be mounted too.

2.8 Miscellaneous hardware stuff

2.8.1 Firmware download

The working firmware for the PBox generated by Paul is available at

http://pulse-sequencer.sourceforge.net/firmware/sequencer_top-v0_29.pof

2.8.2 Building the firmware in quartus II

make sequencer_top.vhd

make sequencer_top.map.eqn

Open the project sequencer top.qpf in Quartus. Press Ctrl+L

2.8.3 How to set up the clock switches

Set both clock selection switches so that they point towards the sma clock connectors. Connect the 100 MHz clock to the left clock input.(The one which is besides the ethernet plugs)

2.8.4 How to set up the ip address

To set the ip address on the fpga board use the red dip switches. The pinout is:

|reset|ip1|ip2|ip3|ip4|dhcp|

set dhcp to OFF

The ip address is 192.168.0.X where X=220 + ip1 + 2 * ip2 + 4*ip3 + 8* ip4

Testing the Hardware

2.9 Testing the hardware of a complete PBox

go to the sequencer2 directory and run following command:

```
python test_pbox_hardware.py
```

This should then open a window with a webbrowser displaying this page.

The tests will run automatically and you will be asked to check whether the hardware will perform correctly. If a test fails, a browser window will be opened to lead you towards troubleshooting information.

2.9.1 Requirements

- The PBox needs be connected to the ethernet
- The router must be set-up accordingly
- The clock generator needs to be locked

Required equipment

- A spectrum analyzer to test the DDS
- A 300MHz oscilloscope

In order to diagnose errors in the LVDS bus following additional equipment is required:

- A USB Blaster to program the FPGA
- A tested DDS board

2.10 Testing the LVDS Connectors

The LVDS bus consists of two 50 pin flat ribbon cables conencting the DDS boards with the breakout boards. Unlike the digital output board there exists no board to test the conenctions visually with LEDs. The LVDS bus can be tested directly with a DDS board. For this the DDS board is connected to the breakout board. Then, a logic analyzer is programmed into the FPGA on the DDS board.

For this, one needs to install QUARTUS II, which is a software from Altera and a FPGA programming device USB blaster. The logic analyzer in QUARTUS II is called SignalTap. More information about SignalTap can be found in:

http://www.altera.com/literature/hb/qts/qts_qii53009.pdf

The SignalTap file is `test_lvds_bus.stp`. In order to download the logic analyzer into the FPGA, the USB blaster needs to be connected to the JTAG port on the DDS board (labelled J1). The correct JTAG file is `dds_controller_stp_test_lvds_bus.sof`

Now the test `Test LVDS bus (Logic analyser)` should be performed (see *Testing the hardware of a complete PBox*).

After applying the test, the state of the bus should be `0xaaaaaaaa`. So all lines of the bus should be alternating high/low.

With the next test, the timing of the bus can be investigated. The SignalTap should cycle between the value `0xfffffffe` and `0x00000000`. The delay between two cycles hould be roughly 15 clock cycles.

If these tests are performed succesfully, the DDS board should work immediately after applying the 1GHz clock.

2.10.1 Troubleshooting

Most common hardware errors are:

- The terminators of the LVDS bus are not isntalled
- The connector between the mainboard and the breakout board is incorrectly soldered.
- The connectors of the flat ribbon cable are not properly attached.
- The clock of the DDS board is not connected
- The LVDS-TTL converters on the DDS board are incorrectly soldered.

About the programming:

2.11 Sequencer2 main documentation

2.11.1 Overview - About the sequencer2

The sequencer2 is a compiler for the FPGA experiment control system known as the Programmable pulse generator (PPG).

The home page of the PPG is at:

http://pulse-programmer.org

The sequencer2 can be downloaded from the mercurial (hg) repository on the source-forge page. With a command line hg tool just type:

```
hg clone http://pulse-sequencer.hg.sourceforge.net:8000/hgroot/pulse-sequencer/sequencer2
```

With a GUI tool (Tortoise HG) just select clone and add the following URL

http://pulse-sequencer.hg.sourceforge.net:8000/hgroot/pulse-sequencer/sequencer2

This high level documentation is written in the restructured text format and can be converted in a HTML or PDF document.

2.11.2 Structural overview

The sequencer2 source code consists of two different modules:

sequencer2	The Bytecode assembler
server	A TCP server for interfacing with LabView

Where the sequencer2 module generates the binary code for the PPG and transmits it via an ethernet connection.

The sequencer2 module can be used without the server module

The server package is a high level interface for the sequencer module and handles the communication with the experiment control software.

2.11.3 Installing the software

For installing the server the Python programming language in version 2.4 to 2.6 is required.

The sequencer2 does not work with python 3.0 or higher

The python programming language can be downloaded at:

http://python.org

The sequencer2 itself does not need an installation. Just copy to files to a directory and run the server from a command line.

2.11.4 Configuring the server

The default configration is stored in the file config/sequencer2.ini. Do not edit his file.

The site configuration is stored in the file user_sequencer2.ini.

To generate this file run the interactive script configure.py:

```
python configure.py
```

This script will generate the file config/user_sequencer2.ini

For a new setup make sure that at least following settings are correct:

Setting	Value
box_ip_address	The network address of the sequencer.
DIO_configuration_file	Your hardware configuration file
sequence_dir	The directory of your sequence files
include_dir	The directory of your include files
nonet	False
reference_frequency	Your DDS reference frequency

The network address of the sequencer is set by the DIP switches on the sequencer PCB board

2.11.5 How to set up the ip address

To set the ip address on the PPG main board use the red dip switches.

The pin usage on the board is:

reset	ip1	ip2	ip3	ip4	dhcp

set dhcp to OFF

The ip address is 192.168.0.X where:

```
X = 220 + ip1 + 2 * ip2 + 4 * ip3 + 8 * ip4
```

2.11.6 Testing the PBox Hardware

An interactive testing program that uses the webbrowser to give directions for troubleshooting is available:

```
python test_pbox_hardware.py
```

It is able to investigate

- The network configuration of the PBox and the sequencer
- The TTL output modules
- The DDS board and its connection

2.11.7 Starting up the server

After that the server may be started with:

```
python test_sequencer2_server.py
```

2.11.8 Configuring the logging module

Generally the logging module is set in the startup file. For a default installation this file is test_sequencer2_server.py

The logging is enabled by the line:

```
logger=ptplog.ptplog()
```

The log level is determined by the configuration file user_sequencer2.ini.

Following options are allowed:

```
log_filename = None
console_log_level = WARNING
combined_log_level = DEBUG
default_log_level = DEBUG
```

If log_filename is set to None, the sequencer logs only to the terminal with default_log_level

Following log levels are allowed:

```
DEBUG
INFO
WARNING
ERROR
```

2.11.9 Logging to files

The logging facilites are also able to log to a couple of files. To log to files the logger should be configured as:

log_filename = log/sequencer2

Now the logger will log to following files:

```
sequencer2_sequencer2.log
sequencer2_api.log
sequencer2_server.log
sequencer2_DACcontrol.log
sequencer2_all.log
```

The file sequencer_all.log will contain messages from all different logging parts with the log level set by combined_log_level

2.11.10 Viewing log files

The log files which are describd may be viewed with the help of the logtools The logtools are included in any sequencer2 distribution and may be invoked typing:

```
python debug/logtools.py
```

2.11.11 Writing pulse sequences

There are 2 possibilities of generating pulse sequences:

- Generate a pulse sequence directly from sequencer2
- Use the server and high level script files to generate pulse sequences

2.11.12 Writing pulse sequences directly in the sequencer2

A typical pulse sequence:

```
import sequencer
import api
import ptplog
my_sequencer=sequencer.sequencer()
my_api=api.api(my_sequencer)
my_api.dac_value(1, 12)
my_api.jump("test")
my_sequencer.compile_sequence()
```

This script is then directly executed in the sequencer2 root directory by typing:

An example is:

```
python [script_name]
```

2.11.13 List of API commands

Below is a table of commands which are available through the API interface

Command	Function
wait(time, use_cycles=False)	wait for a given time in Microseconds
label(label_name)	Insert a label
jump(target_name)	Jump to label with given name
jump_trigger(target_name,trigger)	Jump to label if trigger inputs match the Bitmask
start_finite(label_name,loop_count)	Begin a finite loop with given loop count
end_finite(target_name)	End a finite loop
begin_subroutine(sub_name)	Start a subroutine
end_subroutine(sub_name)	End a subroutine
call_subroutine(sub_name)	Calls a subroutine
ttl_value(value, select)	Sets the status of a 16Bit part of the digital IO
dac_value(value, address)	Sets the DAC on the DDS board with the given address
init_dds(dds_instance)	Control the AD9910 DDS
update_dds(dds_instance)	
set_dds_profile(dds_inst,profile)	
set_dds_freq(dds_inst,freq,profile)	
load_phase	Control the phase registers
pulse_phase	
init_frequency	

2.11.14 Using the DDS from the API interface

A simple example for testing the functionality of the DDS board:

```
my_sequencer = sequencer.sequencer()
my_api = api.api(my_sequencer)
dds_device = ad9910.AD9910(my_device, 800)
my_api.init_dds(dds_device)
my_api.set_dds_freq(dds_device, frequency, 0)
my_api.set_dds_profile(dds_device, 0)
my_api.update_dds(0, dds_device)
my_sequencer.compile_sequence()
```

For more examples on testing the DDS see the file:

tests/test_hardware.py

2.11.15 Writing pulse sequences with server component

The server component acts as an interface between the API component and an advanced experiment control program.

A sequence is executed as follows:

- Experiment control software sends script filename and sequence data to the server
- Server interprets this “command string” and loads the sequence file
- The sequence file is executed
- The server sends data back to the experiment control software.

An example sequence file:

```
# Define the sequence variables
<VARIABLES>
det_time=self.set_variable("float","det_time",100000.000000,0.01,2e7)
</VARIABLES>
```

```
# The save form specifies which data will be saved and how, when a scan is performed.
# If this is omitted a standard form is used
<SAVE FORM>
  .dat   ;   %1.2f
   PMTcounts;   1;sum;              (1:N);          %1.0f
</SAVE FORM>
# Here the sequence can override program parameters. Syntax follows from "Write Token to Params.vi"
<PARAMS OVERRIDE>
AcquisitionMode fluorescence
DOasTTLword 1
Cycles 1
</PARAMS OVERRIDE>
# The sequence itself
<SEQUENCE>
ttl_pulse(["397_det", "866"],det_time)
</SEQUENCE>
# Some spooky LabView stuff
<AUTHORED BY LABVIEW>
1
</AUTHORED BY LABVIEW>
```

This file contains chunk that is needed for the Innsbruck experiment control system also known as QFP ...

The format of the command string is described in the Innsbruck manual available on the sf.net download site.

2.11.16 Reporting Bugs

If you encounter any bugs in the software/hardware of your experiment write an email to:

philipp.schindler@uibk.ac.at

2.11.17 Testing the sequencer2 internals

The python code of the sequencer2 may be tested with following command:

```
python sequencer2_unittest.py
```

Note that this does not test the Hardware, it is intended for testing the source code of the seqeuencer2. It tests the generation of the machine code from the API layer commands and (to a lesser extend) the generation of the API commands from the end user layer.

2.11.18 More documentation

A general talk about the Box may be found on

http://pulse-sequencer.sourceforge.net/innsbruck/obergurgl-box.pdf

A sequence programming overview may be found on:

http://pulse-sequencer.sourceforge.net/innsbruck/box-cheat.pdf

The LabView interface documentation and a general overview may be found in the old documentation:

http://sourceforge.net/project/showfiles.php?group_id=129764&package_id=220283

Please note that this manual refers to the “old” sequencer and is not generally valid for the sequencer2. The LabView interface is identical in the two servers.

2.11.19 API documentation

The server uses the epydoc markup language inside the source code.

To generate the API-level documentation the epydoc interpreter is needed. It is available from: http://epydoc.sf.net

The documentation is generated with the command:

```
epydoc -v --top=server server sequencer2
```

2.11.20 Debugging the sequencer

This simple debugging procedure is able to test the API commands on the hardware itself.

Simple tests are collected in the file tests/test_hardware.py

For testing purposes an interactive python shell is needed. For better testing experince it is recommended to use the improved python shell http://ipython.scipy.org

If python is in your system path the shell may be invoked from the root directory of the sequencer2 in a command window by simply typing:

```
python
```

Then the test_hardware file has to be imported:

```
from tests.test_hardware import *
```

After that the HardwareTests class has to instantiated:

```
ht = HardwareTests()
```

To test the TTL subsystem of the Box type then:

```
ht.test_ttl_set()
```

It is advisable to play around with the commands defined in tests.test_hardware.py to get a feeling of how the commands work

Tips for trouble shooting can be found in the general box documentation:

http://sourceforge.net/project/showfiles.php?group_id=129764&package_id=220283

2.11.21 Debugging the server with ipython

This debugging method is for low level debugging of the server. This can be used to investigate unusual behavior of the hardware or to add more functionality top the API.

For debugging the server the ipython python shell is recommended. Ipython is available at:

http://ipython.scipy.org

Start the ipython shell in the root directory of the server. Generate the necessary variables and includes by running following command:

```
%run test_ipython.py
```

A simple sequence consisting of two ttl pulses is then generated with the commands:

```
ttl_pulse("1",10)
ttl_pulse("2",10)
```

The sequence list is generated with the command:

```
user_api.final_array = user_api.get_sequence_array(sequence_var)[0]
```

Compiling the sequence is done with the command:

```
user_api.compile_sequence()
```

The machine code of the sequence is displayed with the following command:

```
user_api.sequencer.debug_sequence(force=True)
```

2.11.22 COPYING

2.12 Writing sequences for quantum information experiments

2.12.1 Overview over the sequencer2 programming style

The sequencer2 software is divided up in three different layers:

PCP instructions (machine code)
API layer
End user layer

Here, only the end user layer will be covered. The sequence below performs Doppler cooling, a coherent pulse on a carrier transition and a detection cycle

```
DopplerCooling()
pulse_729(theta, phi, "carrier")
Detection()
```

This is then translated into an API layer program that describes the timing of the TTL channels and DDS commands:

```
# Doppler cooling
set_ttl("doppler", 1)
wait(doppler_time)
set_ttl("doppler", 0)
# Generate coherent pulse
switch_on_dds(frequency, phase, amplitude)
wait(rf_time)
switch_off_dds()
# Detection
set_ttl("detection", 1)
wait(detection_time)
set_ttl("detection", 0)
```

Which is then translated into a PCP instruction list that can be exectued directly on the special purpose micro-controller on the PBox mainboard.

A list of the most important end-user commands is available in the PBox cheatsheet.

```
The PBox cheatsheet
```

2.12.2 Writing a sequence file for use with QFP or TRICS

The end-user program is embedded in a Pseudo-XML file that is compatible with both experiment control programs, QFP and TRICS.

```
# Define the sequence variablesxt
<VARIABLES>
det_time=self.set_variable("float","det_time",100000.000000,0.01,2e7)
</VARIABLES>
# The save form specifies which data will be saved and how, when a scan is performed.
# If this is omitted a standard form is used
<SAVE FORM>
  .dat   ;   %1.2f
  PMTcounts;   1;sum;               (1:N);          %1.0f
</SAVE FORM>
# Here the sequence can override program parameters. Syntax follows from "Write Token to Params.vi"
<PARAMS OVERRIDE>
AcquisitionMode fluorescence
DOasTTLword 1
Cycles 1
</PARAMS OVERRIDE>
# The sequence itself
<SEQUENCE>
ttl_pulse(["3", "5"],det_time)
</SEQUENCE>
# Some spooky LabView stuff
<AUTHORED BY LABVIEW>
1
</AUTHORED BY LABVIEW>
```

The definition of the variables is performed as follows:

```
# Define the sequence variablesxt
<VARIABLES>
#Syntax Example:
sequence_var = self.set_variable("variable_type", "variable_name", \
                          default_val, min_val, max_val)

#More examples
float_var=self.set_variable("float","name for labview", 10, 0, 100.0)
int_var=self.set_variable("int","name for labview", 10, 0, 100)
bool_var=self.set_variable("bool","det_time")
</VARIABLES>

# Use the variables defined aboce direct in the python script
<SEQUENCE>
if bool_var:
    ttl_pulse(["3", "5"],det_time)
else:
    for item in range(int_var):
        ttl_pulse(["3", "5"],det_time)
</SEQUENCE>
```

2.12.3 TTL pulses

TTL pulses may act on a list of channels or on a single channel:

```
ttl_pulse(["channel name1", "channel name2"], pulse_duration)
```

For a pulse on a single channel there are two different possibilities for defining the pulse:

```
ttl_pulse("channel name", pulse_duration)
ttl_pulse(["channel name"], pulse_duration)
```

More complex series of pulses can be achieved by using the `is_last` and `start_time` parameters of the pulse methods. By default (when omitting it) `is_last` is set to True. This means that the pulses are attached one after the other. By manually setting `is_last` and `start_time` interleaved pulses are possible:

```
# Create a pulse from time 0 to 100
ttl_pulse(["3", "5"],100,is_last=False)
# Create a pulse from time 50 to 130
ttl_pulse(["1", "4"],80, start_time=50)
#set start time to zero after last pulse
#Create a pulse from 130 to 330
ttl_pulse(["3", "7"],200)
```

2.12.4 Adding a pause to the sequence

A pause between two operations in a sequence is generated by the command:

```
seq_wait(200)
```

2.12.5 Repeating an operations

A sequence can be simply repeated by the python built in for loop:

```
for i in xrange(300):
     # Create a pulse from time 0 to 100
     ttl_pulse(["3", "5"],100,is_last=False)
     # Create a pulse from time 50 to 130
     ttl_pulse(["1", "4"],80, start_time=50)
     #set start time to zero after last pulse
     #Create a pulse from 130 to 330
     ttl_pulse(["3", "7"],200)
```

This creates a machine code where the operations occur 300 times subsequently. This works safely, but the memory of the PBox mainboard is limited and the sequence size is increased:

```
for i in multiple_pulse(300):
     # Create a pulse from time 0 to 100
     ttl_pulse(["3", "5"],100,is_last=False)
     # Create a pulse from time 50 to 130
     ttl_pulse(["1", "4"],80, start_time=50)
     #set start time to zero after last pulse
     #Create a pulse from 130 to 330
     ttl_pulse(["3", "7"],200)
```

This creates a machine code where the repeated part is only compiled once and repetition is performed with the aid of a finite loop inside the FPGA micro-controller. This version saves memory and pulse-length. Please note, that the variable `i` cannot be used within the loop, as the micro-controller does not allow variables in finite loops.

2.12.6 Coherent manipulation - RF pulses

An RF pulse can be generated by the command:

```
rf_pulse(theta, phi, ion, "transition name")
```

where `theta` is the rotation angle in units of Pi. `theta=1` corresponds to a rotation from the ground to the excited state. `phi` is the relative RF (and also laser) phase that determines the rotation axis on the Bloch sphere, `ion` is an integer corresponding to the ion-number.

The actual parameters for the DDS device are frequency and RF-level. These two parameters are encoded in a "transition". In the actual end=user sequence only a string identifying the used transition is required. The data corresponding to this transition is sent from TRICS or QFP to the sequencer2. Normally this is NOT included in the VARIABLES or the sequence itself. See the TRICS documentation for more information ;-)

To insert a transition directly into the Pseudo XML file, put following code between the SEQUENCE tags:

```
transition1 = sequence_handler.transition("test1", {1:1}, 200)
# The transition is used by the following:
rf_pulse(1,0,1,transition1)
```

There exist operations that allow a frequency, amplitude or phase sweep during an RF pulse. These functions are not properly tested in Innsbruck, so use them at your own risk. Please see the file `server/user_function.py` for the function definitions.

2.12.7 Further documentation:

`Slides from an introductory talk to sequence programming`

This documentation is based on SPHINX and is contained in the doc/ folder in the sequencer2 repository.