1 Programmable pulse generator manual

This is a general overview over the programmable pulse generator, a versatile experimental control system for quantum information. Parts of this document are taken for the diploma thesis of Philipp Schindler at the University of Innsbruck[1]. The main source for documentation is the SourceForge project homepage[2].

Contents

Contents

[1] http://heart-c704.uibk.ac.at
[2] http://pulse-sequencer.sf.net

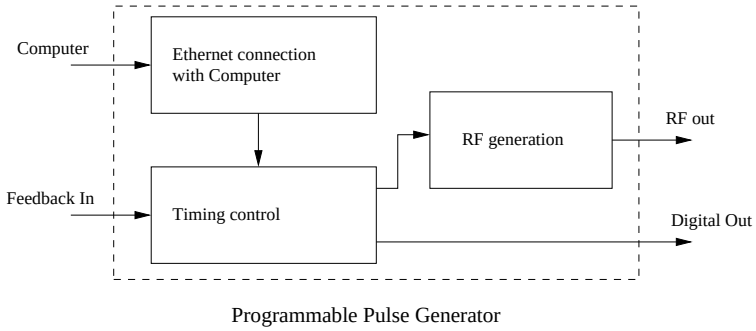

Figure 1: Logical block diagram of the programmable pulse generator. The communication, timing control, digital output subsystem is located in a programmable logic chip. The radio frequency is generated by an external synthesizer.

1.1 The programmable pulse generator

1.1.1 General Overview

The programmable pulse generator is a device which is designed to generate exactly timed digital (TTL) and radio frequency (RF) pulses. As shown in Fig. 1 the programmable pulse generator may be subdivided in four major blocks:

- Communication via Ethernet with the experiment control computer
- Timing control and program flow control
- Radio frequency pulse generation
- Digital output system

The heart of the programmable pulse generator is a complex programmable logic chip (field programmable gate array). A field programmable gate array (FPGA) is a reconfigurable logic device which consists of small logic blocks capable of carrying out arbitrary logic functions. The FPGA used in the programmable pulse generator[3] has over 12000 logical units, and therefore it is possible to realize complex designs (for example an entire microprocessor). The programming of the FPGA is accomplished in a hardware description language (HDL). The VHSIC[4] hardware description language (VHDL) is used in this project. The advantage of using a standard hardware description language is the ability to generate a design independent of the actual type of hardware and independent of the development environment and ensuring therefore portability. The steps for generating a design in an FPGA are:

- Hardware description (VHDL)
- Behavioral Simulation

[3] Altera Cyclone EP1C12
[4] Very-High-Speed Integrated Circuits

- Synthesis and Optimization
- Timing Simulation
- Device programming

For the complete design flow the freely available web edition of the Quartus II design suite from the FPGA manufacturer Altera[5] is used.

The hardware block diagram for the programmable pulse generator is shown in Fig. 2. All logical blocks except the radio frequency generation are integrated in the FPGA. The hardware for the FPGA was designed by Paul Pham and is described in his Master's thesis [1]. The communication with the computer is realized over a standard Ethernet interface via a custom protocol. Therefore no additional hardware on the experiment control computer is required. A more detailed overview of the programming of the FPGA is given in section(1.1.3).

The FPGA has only digital outputs and therefore the radio frequency pulses are generated by a direct digital synthesizer (DDS). Direct digital synthesizing is a technique of generating an analog radio frequency output from a stable digital clock. Due to its digital nature the direct digital synthesizer offers better control over the generated output than analog techniques to generate radio frequency signals (VCOs, PLL, etc). This makes phase coherent frequency switching within one sequence with only one synthesizer possible. For a more detailed description of direct digital synthesizer operation see section(1.1.2). The programmable pulse generator adds the ability to generate amplitude modulated radio frequency pulses. This is realized with a digital to analog converter (DAC) in combination with a variable gain amplifier (VGA). The digital to analog converter and the direct digital synthesizer are controlled by the digital outputs of the FPGA. With the help of additional addressing electronics, the FPGA is able to control up to 16 synthesizers and 16 digital to analog converters. A general purpose I2C serial bus is also implemented in the FPGA programming. This bus may be used to interface different microprocessors and measurement devices, but so far use of this bus has not been required.

Since several research groups are using the programmable pulse generator, it is managed as an open source project on the Sourceforge Project management homepage[6]. The software is released under the BSD open source license[7]. The most recent software source code and hardware design files are available on the project web-pages[8].

1.1.2 Frequency Generation

For the first version of the programmable pulse generator evaluation boards for the direct digital synthesizer, the digital to analog converter, and variable gain amplifier supplied from Analog Devices[9] were used. In order to use more than one frequency source simultaneously,

[5] http://www.altera.com
[6] http://www.sourceforge.net
[7] http://www.opensource.org/licenses/bsd-license.php
[8] http://pulse-sequencer.sf.net
[9] http://www.analog.com

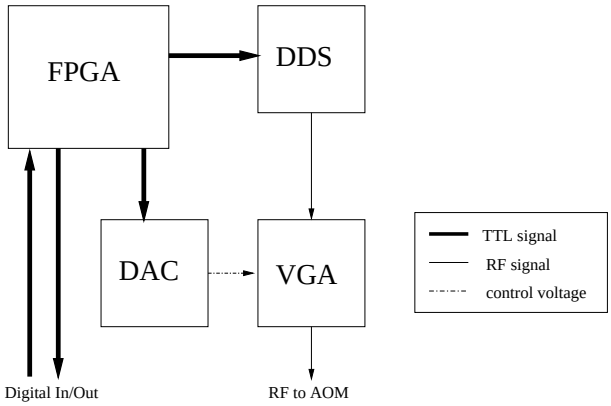

Figure 2: Hardware block diagram of the pulse generator. The core is a programmable logic device (FPGA). Radio frequency pulses are generated by the direct digital synthesizer (DDS). Amplitude modulation of these pulses is realized with the digital to analog converter (DAC) in combination with the variable gain amplifier (VGA). The digital inputs to the FPGA are used as triggers to synchronize the sequence to the mains line phase. The free outputs of the FPGA are used as synchronous digital outputs.

additional addressing electronics ("chain boards") are used for the synthesizer and the digital to analog converter. The variable gain amplifier is controlled by an analog voltage and therefore requires no additional addressing logic. It is possible to address up to 16 different radio frequency channels with four address bits for the synthesizer and the digital to analog converter. The schematics of these chain boards are available on the project web-page[10] .

In 2008 an integrated frequency generation component was developed in Innsbruck This module consists of a single PCB board and has following advantages over the old setup:

- Only a single PCB board instead of 3 per channel.
- Better signal to noise ratio due to the use of a different DDS IC[11]
- More flexibility due to an additional onboard FPGA[12]

Direct digital synthesis The direct digital synthesizer may be divided into digital and analog parts. A block diagram of a direct digital synthesizer is shown in Fig. 3. The digital part consists of a phase accumulator and a sine lookup table. The phase accumulator increases its value every time it receives a positive slope on the clock input. The value by which the phase register is increased determines the frequency of the generated sine and is called the frequency tuning word (FTW). From the actual value in the phase accumulator the amplitude is obtained with a sine lookup table. The width of the phase register is 32 bit, but since a 32 bit lookup table would require over 400 million entries, only the 12 most

[10] http://pulse-sequencer.sf.net
[11] AD9910
[12] altera cycloneII

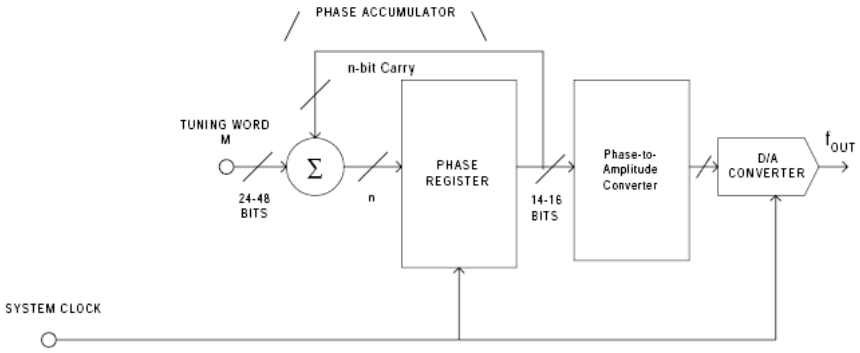

Figure 3: Block diagram of the direct digital synthesizer. It consists of a phase accumulator, a sine lookup table and a digital to analog converter. (image from [3])

significant bits are used. This constraint has no effect on the accuracy of the frequency, which is still determined by the phase and therefore the 32 bit register. The sine amplitudes are converted into an analog current with a 10 bit digital to analog converter inside the chip. Therefore there is almost no loss of information due to the reduced width of the sine lookup table. On the analog side there are impedance matching and filter networks.

As for all digitally sampled signals, the Nyquist sampling theorem has to be applied on the output of the synthesizer [2]. This theorem states that every frequency f, generated with a sampling frequency f_s, produces an additional mirror frequency $f_m = f_s - f$. This limits the output frequency f to a maximum frequency $f_{max} \leq f_s/2$. The unwanted mirror frequencies have to be filtered out using analog filters. In practice the maximum frequency is determined by the steepness of this output filters to $f_{max} \approx 0.4 \cdot f_s$. A more detailed introduction to direct digital synthesizer operation is given in references [3, 4].

Performance of the direct digital synthesizer The main advantage of a direct digital synthesizer over analog frequency generation methods is that the switching between frequencies and even switching on and off the output may be achieved in a few hundred nanoseconds. The frequency drift of the output signal is only dependent on the drift of the reference frequency which may be obtained from a precise frequency standard. Given the fact that the phase is controlled digitally and switched with great accuracy and repeatability, phase coherent switching between several frequencies is possible within a few microseconds.

The limitations of direct digital synthesizers are due mainly to the limited resolution of the built in digital to analog converter and appear as spectral impurities and quantized digital to analog converter noise. Another fundamental disadvantage of digital to analog conversion is the presence of mirror frequencies. The higher harmonics generated by nonlinearities of the output are also subject to the Nyquist theorem and mirror frequencies of them may be near the fundamental frequency. To estimate the spectrum of the DDS for all possible clock and output frequencies an on-line simulation tool is available at the

Maximum output frequency	$\approx 350\,MHz$
Minimum output frequency	$< 10\,MHz$
Number of coherent frequencies	16
Frequency switching time	$\leq 150\,ns$
Phase offset accuracy	$2\pi\ 2.4 \cdot 10^{-4}$
Signal to noise ratio	$50\,dBc$
Frequency resolution	$0.18\,Hz$
Maximum output level	$\approx -1\,dBm$

Table 1: Figures of merit for the radio frequency generation of the programmable pulse generator for a clock frequency of 800MHz.

Analog Devices website[13].

The measured figures of merit for the synthesizer used in the programmable pulse generator are presented in Tab. 1. The frequency resolution of the synthesizer depends on the register width of the phase accumulator and the reference frequency. The minimal frequency step is equivalent to a change in the least significant bit of the frequency tuning word. This leads to the following resolution for a reference frequency of 800MHz:

$$\Delta f = \frac{f_{ref}}{2^{32}} = 0.18\,Hz$$

If a higher frequency resolution is required, it is possible to decrease the reference frequency, however this also decreases the maximum achievable output frequency.

Phase coherent switching For coherent qubit manipulation the phase of the laser light is crucial as this determines the axis of rotation within the XY plane of the Bloch sphere. The phase reference for a particular transition is set by the first laser pulse exciting that transition. All subsequent pulses have to be phase-coherent with respect to the first pulse in order to achieve fully controllable coherent state manipulation. As the laser is driven by an AOM, the phase may be controlled by setting the phase of the radio frequency driving the modulator. Additionally a single sequence may include pulses on more than one transition (for example carrier and sideband transitions) therefore phase coherent switching of multiple frequencies is required within one sequence. A phase offset is required to realize rotations around the Z axis of the Bloch sphere. The synthesizer has only one phase accumulator built in, therefore phase continuous switching between different frequencies is not possible with a single synthesizer. Fig. 4 gives an overview of the different switching modes.

Because the synthesizer has only a single phase accumulator, keeping track of the phase of multiple frequencies is handled by the FPGA which has 16 phase accumulators similar to the ones built into the synthesizer. These phase accumulators have the same width as the phase accumulator in the synthesizer. The FPGA can only be clocked with maximum

[13] http://designtools.analog.com/dtDDSWeb/dtDDSMain.aspx

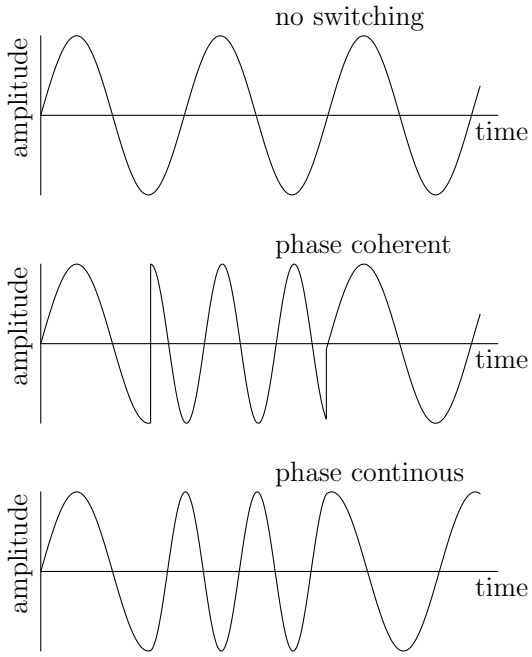

Figure 4: Different switching methods. For phase coherent switching the phase is set to the value as if there was no switching at all. For phase continuous switching there are no phase jumps.

frequencies of around 100 MHz and therefore the FPGAs frequency tuning word is always the tuning word of the synthesizer multiplied by eight. In a phase coherent frequency switching event, the content of one of these registers is written into the phase accumulator of the synthesizer. As there are different relative phases required for a single frequency it is possible to add a constant value to the current phase accumulator in the FPGA. The timing of the writing process determines the quality of the phase coherent switching. As long as the delay between reading out the phase accumulator in the FPGA and setting the phase accumulator in the synthesizer is constant for every switching process, the delay may be left uncompensated as it leads to a constant phase offset. For writing the phase word to the synthesizer the 32 bit word is split into four eight bit words. To keep the phase word constant during the switching event, the value of the phase accumulator in the FPGA is copied into a dedicated register. This register is then transferred to the synthesizer. After this writing process the synthesizer takes over the new phase word with the rising slope of a digital control signal generated by the FPGA. This process ensures a constant time delay between writing the phase accumulator into the register in the FPGA and updating the phase register in the synthesizer.

Amplitude shaping In order to create arbitrary pulse shapes for reducing the off-resonant excitation, a variable gain amplifier controlled by a digital to analog converter is used. The digital to analog converter is itself controlled by the FPGA. An amplitude shaped pulse is generated in five steps:

- Phase coherent switching to the desired frequency and relative phase.

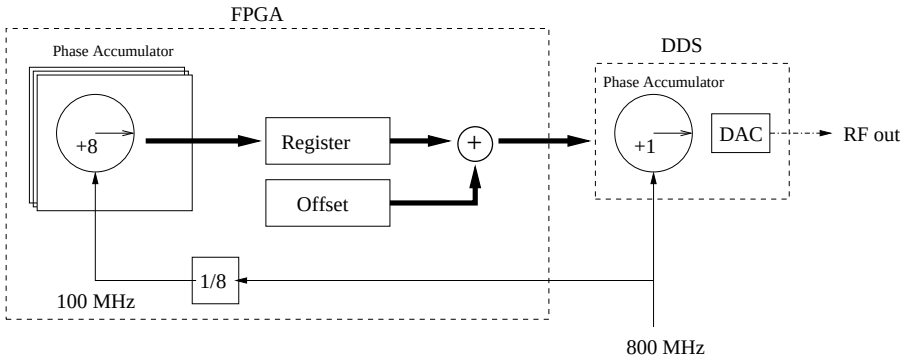

Figure 5: Block diagram of the phase registers inside the FPGA. The FPGA clock frequency is the synthesizer reference clock frequency divided by eight. A phase offset is added to the value of the phase register and the sum is written to the synthesizer.

- Generation of the rising slope with the digital to analog converter.
- Wait cycles for realizing the desired pulse length.
- Generation of the falling slope with the digital to analog converter.
- Switching off the synthesizer.

The variable gain amplifier is an analog devices AD8367 which is logarithmic in control voltage. Therefore the digital to analog converter has to compensate this behaviour. This is done while compiling the sequence in the experiment control computer. The digital to analog converter is an AD9744 which has a resolution width of 14 bits and a maximum clock rate of 100 MHz. Addressing of multiple converters is realized with chain boards which enable the clock of the converter depending on the addressing word. The resolution of the amplitude shaping is $0.01\,dB$ (the resolution is given in dB because the VGA is logarithmic in control voltage). Therefore the resolution of the (linear) pulse shape changes with the actual radio frequency power used.

1.1.3 The FPGA Core

The design of the FPGA system of the programmable pulse generator may be subdivided in three different layers: the hardware, the firmware and the software. The hardware layer was designed by Paul Pham for his Master's thesis at MIT [1]. The firmware layer controls the behaviour of the FPGA. The firmware currently used was also written by Paul Pham, whereas future versions are likely to include new components written by the author. The software layer is running on the experiment control computer and handles the generation and transmission of the machine code. It was written in a collaboration between Paul Pham and the author.

The hardware is designed to be able to generate accurate digital pulses. The Low Voltage Differential Signaling (LVDS) logic standard is used for external digital signals.

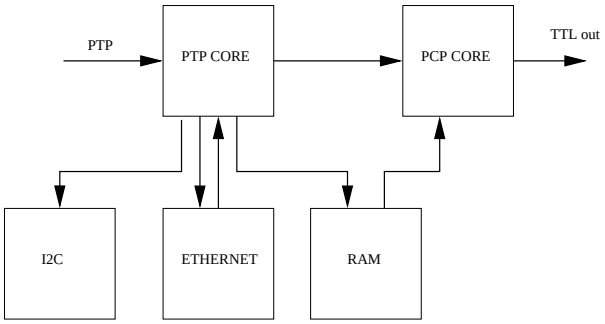

Figure 6: Simplified block diagram of the FPGA firmware. The transfer core handles the communication with the experimental control computer and writes the byte code to the memory(RAM). It also handles the start and stop commands for the timing core (PCP). The timing core reads out the program from the memory and decodes and executes it. The I2C core is controlled directly by the transfer core.

This standard was defined by National Semiconductor[14] and is widely used for high speed long distance digital signal transmission. Due to its differential nature it is very insensitive to electrical noise coupled into and reflections generated inside the transmission lines. The board has different options for reference clock signals. The two independent clock inputs of the FPGA may be either connected to external radio frequency connectors, connected to a clock derived from the external bus or connected to one of the two quartz oscillators placed on the board. To increase the possible program size an external memory chip is available on the board.

The firmware may be divided in the following blocks, as shown in Fig. 6

- PTP pulse transfer protocol: For transferring data between the computer and the FPGA.
- PCP pulse control processor: The core with the timing and program control logic.
- Interfaces to the external and internal peripherals.

The pulse transfer protocol is a custom protocol for controlling the programmable pulse generator. It is based on the User Datagram Protocol (UDP) which is a standard Internet protocol. The part of the firmware which implements this protocol is denoted in the following as the transfer core. The protocol supports directly the connection and synchronization of several FPGA boards. The most important commands are shown in Tab. 2.

The full specifications of the pulse transfer protocol may be found in chapter 4 of Paul Pham's Master thesis [1] or in appendix(3). The transfer core extracts the machine code from the received data and writes it to the memory (RAM) of the programmable pulse generator. After receiving the data the sequence is initiated with the start command.

After receiving the start command, the pulse control processor (PCP) fetches the first instruction from the memory. This part of the firmware is denoted as the timing core. It

[14] http://www.national.com

Name	Description
Discover	Searches for connected programmable pulse generators
I2C	Sends data to the general purpose I2C Bus
Start	Starts the current sequence.
Stop	Stops the timing core to initiate a new data transfer.
Write	Writes data to the memory of the programmable pulse generator.

Table 2: Important commands of the pulse transfer protocol

Name	Description
Pulse	Generates a digital pulse on the external bus.
PulsePhase	Writes the content of a phase accumulator to the external bus.
Jump	Jumps to a different memory address.
Branch	Jumps to a different memory address depending on triggers.
Wait	Waits for a certain number of clock cycles.

Table 3: Important machine code commands of the programmable pulse generator

decodes and executes the single commands of a program. In addition to basic program control structures, the instruction set also contains commands for setting the digital output and for phase coherent switching. Although the instruction set shows parallels with a microprocessor there are no general purpose registers, arithmetic or logic functions implemented. This leads to the fact that the only way to influence the running program is through trigger inputs. Small changes require a full recompilation. The most important commands are shown in Tab. 3. The phase accumulators required for phase coherent switching as described in section(1.1.2) are also part of the timing core.

The outputs on the external bus are separated in digital TTL outputs of the programmable pulse generator and control signals for the synthesizer and the digital to analog converter. In total the external bus is 64 Bits wide, where 20 bits are available for digital TTL outputs. For the implementation of the wait command, the timing core uses a register which is loaded with the number of clock cycles which correspond to the desired waiting time. For each rising edge of the clock the value of this register is decreased by one. The next command is not executed until the register value is zero.

Additional information about the pulse transfer protocol and the programming of the programmable pulse generator is given in appendix(3).

1.2 The Software

1.2.1 Overview

This section gives an overview over the software which is used for experiment control of the $^{40}Ca^{+}$ experiment. The main experiment control program is written in the graphical programming language of LabView. It was written by Wolfgang Hänsel and a detailed

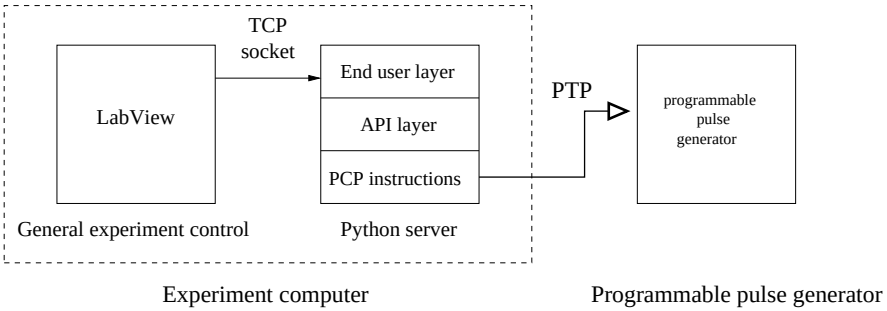

Figure 7: Experiment control software. The LabView control software and the Python server are connected via a TCP connection. The Python server generates byte code for the FPGA and transmits it via the pulse transfer protocol.

description would be beyond the scope of this thesis. This section will concentrate on the part of the experimental control software which generates and transfers the machine code for the programmable pulse generator. This software was written entirely in the Python open source programming language[15]. It will be denoted as the Python server. An overview of the total experiment control software is given in Fig. 7. The Python server source code is available on the pulse sequencer project homepage[16]. Communication between the LabView program and the Python server is realized with a standard network protocol (TCP[17]) to ensure maximum portability. In the current setup the Python server and the LabView program run on the same computer and communicate over the Ethernet loopback interface. The Python server is itself divided into three different layers. It receives a human readable pulse sequence, converts this into machine code for the timing core inside the FPGA and then transmits this to the programmable pulse generator.

1.2.2 The Python server

The different Layers of the Python server are shown in Tab. 4. A layer provides functions to the layer above and uses functions from the layer below. The end user layer receives functions directly from the LabView program and converts this to API layer commands. Examples of functions at the end user layer layer would be "single qubit rotation" or "side-band cooling". The application programming interface (API) layer converts this functions to commands for the programmable pulse generator. Examples of functions at the API layer are "switch on synthesizer" or "set digital output to level high". The pulse transfer protocol is implemented in this layer. It is also used for writing small programs to debug the hardware. The compiler layer generates the binary machine code from these API commands.

There are two different ways of describing a pulse sequence in the end user layer. They are called the "parallel mode" and the "sequential mode". The parallel mode is used in the $^{40}Ca^{+}$ experiment whereas the sequential mode is used by all other experiments. In the

[15] www.python.org
[16] http://pulse-sequencer.sf.net
[17] Transmission Control Protocol

Name	Description
End user	Receives human readable pulse sequence.
API	Converts pulse sequence to events for the programmable pulse generator
compiler	Machine code generation for the timing core

Table 4: The Python server software layers.

Algorithm 1 Example end-user program

```
doppler_cooling()
R729(1,1,0,Carrier)
detection()
```

parallel mode the transmitted data is directly executed by the Python server whereas in sequential mode the LabView program sends a command string with the file name and the parameters of the current sequence. The difference between the two modes is described in appendix(2).

The syntax of the following program examples is not strictly correct but instead shows the concept of the different layers. In Algorithm(1) a simple end user layer program is shown. It consists of a simple experimental sequence with Doppler cooling, a single qubit rotation and qubit detection. The corresponding API layer program is shown in Algorithm(2). While the end user layer program is self explanatory the API layer program needs some further explanations.

The begin_sequence() function is mandatory at the beginning of all sequences. It generates the internal variables used for the API and the compiler layer and additionally generates events for initializing the external hardware. The begin_finite_loop() marks the start of the sequence to be repeated. The phase accumulators in the FPGA are initialized with the initialize_frequency(f_carrier) command. The set_ttl() and switch_frequency() commands are used to generate the digital and radio frequency pulses. The end_finite_loop(100) command marks the end of the repeated sequence. The end_sequence() command starts the compilation and the transmission of the sequence. A complete list of all available commands is given in the appendix of this thesis.

1.2.3 Limitations

One major limitation of the Python server is the compilation speed. Amplitude shaped pulses need 100 or more digital to analog converter events. When the amplitude shape is calculated for every pulse used in the sequence this leads quickly to a large sequence which takes over one minute to compile. In order to work comfortably the compile time should not be much longer than one second. Therefore the amplitude shapes are compiled once and stay in the FPGA memory until a new shape is compiled. In the program they are then invoked as subroutines. They are not overwritten when a new sequence is generated, and this leads to short compilation times when the parameters of the amplitude shape is

Algorithm 2 The API program generated from the sequence shown in Algorithm(1)

```
begin_sequence()
begin_finite(loop)
initialize_frequency(f_carrier)

# doppler cooling
set_ttl(["866 sw","397 dopp","397 sw",1)
wait(1000)
set_ttl(["866 sw","397 dopp","397 sw",0)

# single qubit rotation
switch_frequency(f_carrier,0)
rising_slope(shape_time,amplitude)
wait(t_pulse)
falling_slope(shape_time,amplitude)
switch_off(f_carrier)

# detection
set_ttl(["866 sw","397 det","397 sw",1)
wait(2000)
set_ttl(["866 sw","397 det","397 sw",0)

end_finite_loop(count=100)
end_sequence()
```

not altered. If the parameters of the amplitude shape has to be altered this leads to long compilation times and makes system calibration tedious.

Another bug of the programmable pulse generator is that when a new sequence is loaded, sometimes the first command of the sequence may not be executed correctly. This is due to a known bug in the FPGA firmware. A workaround is simply sequence recompilation.

2 Programmable pulse generator manual

This section is written for the first version of the python server. It is not strictly valid for the second generation python server, but the general ideas are the same. For an accurate reference manual of the sequencer2 server please refer to the sequencer2 website.[18]

2.1 The python server

In this section the commands for programming the programmable pulse generator (PPG) are explained and general instructions for using the PPG are given. A more recent version of this documentation may be found on the pulse sequencer homepage[19].

Typographic conventions Python code is written as :

```
python_function(arg1,arg2=10)
python_var=True
```

Filenames and command line commands are written as: `path_to/filename.txt` .

2.1.1 Installing the python server

As a prerequisite the python programming language in version 2.4 or 2.5 is required. It may be downloaded from

`http://www.python.org` .

The python server source code for generating and executing pulse sequences for the PPG are available for download at the PPG project homepage.

`http://pulse-sequencer.sf.net`

The package for QFP2.0 is called `sequencer-python/python-qfp-2.0`
The package for QFP_LIN is called `sequencer-python/python-qfp-2.0`

2.1.2 starting up the python server

The server may be started directly by the command line if the python executable is within the search path of the environment variable.

```
python start_box_server.py
```

The command line options for this command are:

`--debug debug_level` Where the debug levels are supposed to be used as follows:

- 1 : The most important server messages are displayed
- 2 : Many server messages are displayed
- 3 : Many compiler messages are displayed. It's not recommended to use this with the server. This debug level is intended for low level debugging in the machine code generation code

[18] http://sequencer.brainity.com
[19] http://pulse-sequencer.sf.net

`--nonet` The python server is started without any network connection. This switch may be used for testing the server on a computer where no PPG is connected.

2.1.3 Troubleshooting

Error messages from the server are generally displayed on the command line terminal executing the server. If the command line terminates without displaying an error messages, open a new command window and run the server in this window. This command window does not close when the python server crashes. If the box is connected to the voltage supply the LEDs beneath the DIP switch for selecting the MAC address should blink a few times while booting up. If this is not the case the clock settings and the presence of a clock signal should be checked. Below a few error messages and possible solutions are described.

No pulse transfer protocol reply: If the server complains that it got no pulse transfer protocol reply there may be a hardware or network problem. Following steps are recommended for troubleshooting:

- If it worked before it may help to flush the ARP[20] cache of windows. Open a command shell and type

`netsh interface ip delete arpcache` . If this does not solve the problem, waiting for 10 minutes before retrying to start the server may be the solution.

- The network connectivity can be checked with "link" LEDs at the FPGA board. If this LED is dark, it's most probably a hardware problem. It should be checked if the network cables are plugged in correctly and the right cable type (No crossed connections) is used.
- The DIP switch for setting the MAC address of the FPGA board might not be set to match the settings given in the python compiler configuration. See the section about configuring the server for details.
- There is no DHCP server running on the network where the box can get an IP address from or the box may be misconfigured so it doesn't send an DHCP request. The DHCP server is most likely implemented in an Ethernet router.

For further troubleshooting it is advisable to use a network analyzing software like the freely available wireshark[21].

When investigating the network traffic with wireshark , restart the FPGA and look for a MAC address of 00:01:ca:22:22:xx performing a DHCP request. (xx is the value of the DIP switch for setting the IP address) If this request is accepted, the FPGA is obtaining a IP address successfully. It is likely that the error source is then be a misconfigured python server.

[20] address resolution protocol
[21] www.wireshark.org

KeyError , AttributeError If the server returns some strange errors like KeyError or AttributeError there might have been an error with transferring the variables from LabView to the server. Another possibility is that the current sequence is using a TTL channel which is not defined in the hardware configuration file. Check if all TTL channels used in the current sequence are available in the QFP2.0 hardware configuration file.

Pulses still overlap If you got this error and the sequential mode is used there might be something wrong with the delay times in `Innsbruck/__init__.py` . If this error occurs while running in a parallel environment t some overlapping pulses may have been defined. This might not be a problem but the timing of your script may be incorrect.

Other errors One error that occurs frequently in new installed systems are incorrect decimal separators sent from LabView to the python server. Check that the decimal separator is a point "." and not a comma "," .

2.1.4 Pulse programming reference

There are two fundamental modes of programming a pulse sequence:

- sequential programming (default, used for QFP2.0)
- parallel environments (used for QFP Lin)

2.1.5 Sequential programming

This is the default programming mode where the pulses are usually executed sub sequentially. Delays between pulses may be inserted manually. The structure of a program is as follows:

```
pulse1
pulse2
wait(10)
pulse3
```

This is intended to be used for sequences when the 729 beam is used to make rotations on the qubits. The pulses are characterized by the ion, transition , the duration given in angle, and the phase:

`R729(ion,theta,phi,[transition],[start_time],[is_last])`

If the transition is omitted a default transition given by the Labview control program is used.

The command for inserting a waiting time is.

`seq_wait(time)`

The pseudo XML pulse program structure A sequence file is read by the LabView program as well as by the python server. Therefore more information than just the pulse sequence is required to be stored in one file. This is realized by creating groups of information

which are separated by XML[22] tags. The tags which are interpreted by the python server are <VARIABLES>, <TRANSITION> and <SEQUENCE>. In the <VARIABLES> group the variables which are controlled by and transmitted from LabView are defined. In the <TRANSITION> group the transition objects which are defined in LabView may be edited before the phase accumulators in the FPGA are initialized. The <SEQUENCE> group contains the actual sequence.

```
<VARIABLES>
Duration=self.set_variable("float","Duration",20,1,2e7)
</VARIABLES>
<some_tag_for_labview>
some content for labview
</some_tag_for_labview>
<SEQUENCE>
TTL("PM trigger",50)
seq_wait(Duration*1000.0)
TTL("PM trigger",50)
</SEQUENCE>
```

Variable definition The variable definition in the XML file has the following syntax:

```
VAR_NAME=self.set_variable("TYPE","LV_NAME",[MIN],[MAX])
```

VAR_NAME:
: The name of the python variable which is used in the script

`TYPE:`
: The type of the variable. One of FLOAT, INT, BOOL, STRING

`LV_NAME:`
: The name of the variable in the LabView program.

`MIN ; MAX`
: The bounds of possible values of the variable. This value is only used by LabView and has no influence on the python server.

Pulse code The pulse code is in the `<SEQUENCE>` group. Variables defined as described above may be used as a simple python variable. Below an overview over the different commands is given.

TTL Outputs The names of the TTL outputs are taken from the "Hardware Settings.txt" file of the QFP. If the device is !PB the output is inverted that means that setting the output to 0 will result in a voltage of 3.3V at the according output. The location of the Hardware settings file is determined in `innsbruck/__init__.py` .

```
866 sw.ch=0
866 sw.Device=!PB
```

[22] Extended Markup Language

Generates the device “866 main” . To generate a pulse on this channel the following code is used:

```
ttl_pulse("866 sw",10)
```

This creates a TTL **pulse** of given duration. To switch the state of the TTL outputs on a given position in the sequential environment the `set_TTL` command is used:

```
set_TTL("866 sw",1)
...
set_TTL("866 sw",0)
```

RF pulses The command for generating RF pulses is:

`R729(ion,theta,phi,[transition],[start_time],[is_last])`

The technical details of the rotation are given by the transition object. See the paragraph above how to define these transitions.

Depending on the version of the python compiler used the output for the second DDS does **not** support phase coherent switching. The actual implementation of this command may differ from one particular experiment to another. It is advisable to customize the pulses for the particular experiment with the help of include files as described below.

Defining Transitions Transition objects in the python server are needed to do phase coherent switching between pulses. A transition object has information about the frequency, the shape form, the amplitude and the Rabi times of the radio frequency pulse which is needed to excite the given atomic transition. Transitions may be defined in two different ways:

- Direct in the sequence file
- From LabView via the TRANSITION keyword

An example for the direct definition

```
Carrier=transition(transition_name="Carrier",t_rabi=t_carr,
frequency=freq1,amplitude=1,slope_type="blackman",
slope_duration=0.2,amplitude2=-1,frequency2=0)
```

This transition will be called by `R729(ion,theta,phi,Carrier)`
A transition named carrier which is transmitted from LabView is called by: `R729(ion,theta,phi,'carrier')`

Note that the name of the transition is now a string, whereas by direct definition above the transition object is handled as a variable. The information of the transition is submitted from LabView to the server.

seq_wait The syntax for the sequential wait function is:

```
seq_wait(time)
```

Where time is the waiting time in us. The specialty of this function is that it is possible to define Some pre and post delays depending on the previous and following commands. For an example if a seq_wait() instruction is in between to R729() commands, it will count the delay between the two points where the amplitude of the slopes are at the half maximum. An error message will be returned if the wait duration is smaller than the slope duration plus the frequency switching delays.

The is_last variable There is a possibility to generate overlapping pulses by using the `is_last` variable. For a sequence which generates the following pulses

```
TTL "866 sw" from 0 to 100
TTL "397 sw" from 90 to 110
```

This would look in the sequential mode like:

```
ttl_pulse("866_sw",100,start_time=0,is_last=False)
ttl_pulse("397 sw",20,start_time=90)
```

The `is_last` variable is by default `True`.

Defining new commands In the `Innsbruck/__init__.py` file the include directories are defined:

```
includes_dir="e:/My Documents/qfp_2.0/PulseSequences/Includes/"
```

Writing include files A sample include file may look like:

```
description="long optical Pumping for ca43"
function_name="ca43.OpticalPumping2"
arguments="optional: length=50"

class OpticalPumping2(PulseCommand):    #for Ca43
  def __init__(self,length=50):
    configuration=self.get_config()
    ttl_pulse(length,"7",is_last=False)
    ttl_pulse(length+1,"8",start_time=0)

ca43.OpticalPumping2=OpticalPumping2
```

This defines the function `ca43.OpticalPumping2` which may be used in a sequence file. The variables description , Function_name and arguments are necessary for the built in documentation system.

The prefixes for Include functions are:

- `ca40` for the linear trap
- `ca43` for the ca43 experiment

- `cqed` for the CQED experiment
- `segtrap` for the segmented trap

To avoid confusion the prefix for a particularly experiment should be used.

Documenting Include files The variables `description`, `function_name` and `arguments` are used to automatically generate a documentation of the commands defined in the include files. To display this list the python `script include_doc.py` which resides in the same directory as the `start_server.py` script is used.

2.1.6 Parallel environment

In this programming mode the absolute start times in relation to the start of the sequence and the durations of the pulses are given. The pulses may overlap but they **cannot have the same start or stop time**. An example parallel program:

```
start_parallel_env()
ttl_add_to_parallel_env("866 sw",0,10)
ttl_add_to_parallel_env("397 sw",2,5)
shape_add_to_parallel_env(start_time=12,duration=5.0,frequency=freq1,\
phase=0,type="blackman",slope_duration=1.1,amplitude=1.0)
end_parallel_env(trigger="Line",repeat=30)
```

This switches on the "866 sw" output at 0us for 10 us and the "397 sw" output at 2μs for 5μs. To some extend timing conflicts which arise from overlapping pulses may be resolved. The server sends a warning response if the conflict is not resolved successfully. The last line repeats the sequence 30 times and waits for a line trigger every time it is running a single repetition. A parallel environment is started with the start_parallel_env() function. Prior to this function the coherent frequency initialization may be performed. The end of a parallel environment is set with the end_parallel_env(trigger,repeat) function. The trigger variable is either "None" or "Line". When it is set to "Line" the PPG waits for a rising slope on the Trigger input 0.

Coherent frequency initialization Before using phase coherent frequency switching the frequencies have to be initialized at the beginning of your program :

```
freq1=coherent_create_freq(frequency,0)
first_dds_init_frequency(freq1)
```

TTL pulses A TTL pulse is generated with the command

```
ttl_add_to_parallel_env(ttl_channel,start_time,stop_time)
```

RF pulses RF pulses are generated with the command

```
get_shaped_pulse(duration,frequency,type,[slope_duration=0],\
  [amplitude=1],[phase=0],[frequency2=0],[amplitude2=0])
```

If `amplitude2` is bigger than 0 than the second DDS channel is switched on with frequency `frequency2` . If `slope_duration` is bigger than 0 than the output is a pulse with a Blackman shape where the slope time is given in microseconds.

It is not advisable to use arbitrary numbers for the slope_duration and amplitude. The program pre compiles the shapes because they take a long time to compile. It has also to recompile if the amplitudes of the pulses change.

2.2 Configuring the software

This section is divided in two parts. In the first part the steps necessary for setting up a "standard" PPG for use with QFP is described whereas the second part covers all configuration possibilities. Any change in the configuration requires a restart of the server.

2.2.1 Basic configuration

This part relies on a working QFP2.0 environment. The configuration files for the python servers reside in the `innsbruck` and the `test_config` directories. The main files are the `__init__.py` files in the respective directory.

In the innsbruck directory the `configuration` class contains the configuration information. To alter the configuration the methods of this class may be changed.

The hardware configuration file The location of the hardware configuration file is defined by the `hardware_config` method of the `configuration` class. This file is generated by the QFP2.0 `Settings Editor`.

The sequences directory The sequence directory is defined by the `sequences_dir` method of the configuration. An example is `sequences_dir=path_to_qfp/PulseSequences/`.

The includes directory The includes directory is defined by the `includes_dir` method of the configuration. An example is `includes_dir=path_to_qfp/PulseSequences/Includes/` .

Setting the MAC address of the FPGA board The address of the FPGA board is set in the `__init__.py` file in the `test_config` directory. The address is set via the DIP switches on the FPGA board. The function of the DIP switches are defined as follows:

Reset	IP1	IP2	IP3	IP4	DHCP

Where the address of the FPGA board is calculated as $IP1 + 2\,IP2 + 4\,IP3 + 8\,IP4$. The address is set in the `mac_byte` variables of the sequencer generator statements in the `__init__.py` file. All occurrences of the sequencer generator have to be altered.

2.2.2 In depth configuration

TCP ports The communication with LabView is handled over the TCP protocol. The server is listening by default on port 8880 . The port is set in the `default_port` method of the `configuration` class.

Activating the parallel mode For using with the old qfp the parallel mode has to be activated. This is done by setting the `parallel_mode` method of the `configuration` class to `True`.

Setting the TCP server mode The TCP server is able to switch between different TCP connection protocols. All variables discussed here are methods of the `configuration` class. If the `answer_tcp` method is set to `True`, the server sends a response to the LabView program after finishing compiling and transferring the sequence. If the method `send_pre_return` is `True`, an additional answer is sent to LabView after error checking but before compiling the sequence. This is helpful if the compilation time is long and therefore it is possible to determine whether the server is not responding or it is busy with compiling a sequence. For the standard settings of QFP2.0 these methods have to be set to `True`.

If the `disconnect_tcp` method is set to `True`, the TCP connection is terminated after each LabView command. This may be helpful if problems with a long lasting TCP connection may arise. For a standard QFP2.0 setup this method should be set to `True`.

Resetting the TTL outputs If it is desired to reset the TTL outputs of the PPG with the beginning of each sequence the `reset_ttl` method of the `configuration` class may be set to `True`.

Configuration of pulse shapes Pulse shapes have to be configured in two files in the `innsbruck` directory. First an entry has to be made in the `pulse_dictionary` method of the `configuration` class. In this dictionary a function has to be assigned with a string. The functions for the pulse shapes are defined in the `pulses.py` file.

Calibration of the VGA The calibration function for the DAC and the VGA is defined in the `calibration` function at the end of the `__init__.py` file. For the QFP2.0 environment the input value is the desired output power in dB where 0 is the maximum and the return value is the DAC value which is an integer between 0 and 16383. For the old QFP the input value is a linear power value where 1 corresponds to the old Marconi setting of +13dBm.

Syntax of the hardware configuration file Normally the hardware configuration file is generated by QFP2.0. For testing purposes it might be necessary to generate a different hardware configuration file. The hardware definition syntax is as follows:

```
854 sw.Device=PB
854 sw.ch=15
866 sw.Device=!PB
866 sw.ch=17
```

The `!PB` indicates a inverting channel. It doesn't matter if the device or the channel is the first argument.

2.2.3 Configuring the devices

The compiler has to be reconfigured if you make changes to your hardware settings; e.g. adding an additional DDS , ...

Configuring the devices For configuring the devices it may be necessary to edit the `innsbruck/__init__,py` as well as the `test_config/__init__.py` files

The timing and the RF frequency generation depend on the clock of the DDS boards and the FPGA. It is assumed that the FPGA clock is derived from the DDS clock and that it operates at 1/8 of the DDS clock frequency.

```
ref_freq=800
cycle_time=(1e3/ref_freq)
```

The cycle time is given in nanoseconds. Every other time used within the compiler should be given in microseconds.

```
ttl_device={}
```

The ttl_device is a dictionary where the corresponding channels to the hardware file are stored. It also stores whether the channel is inverting.

```
first_dac_device=dac_factory.create_device(chain_address = 0x01)
```

This defines the first DAC which is accessible via innsbruck.first_dac_device. If another DAC should be added just another line which is similar to this and a command in the `api.py` file is necessary .

```
dds_factory_create_devices(chain_address={1; 0x1, 2; 0x2},ref_freq=ref_freq)
```

This defines the first DDS device. Note that it's not possible to mix up DDS with different reference frequency because the data transfer relies on fixed reference frequency dividers between the FPGA clock and the DDS clock.

2.3 Configuring the Hardware

Details on the firmware and on programming the FPGA are given in section(3). In this section the configuration of the FPGA board, the chain boards and the evaluation boards is covered

The FPGA board The jumpers CLK0 and CLK2 select the different clock options. Where internally CLK0 is routed to the PCP core clock and CLK2 is routed to the PTP and the Ethernet clock.

The DIP switch J3 determines the network address and the DHCP mode the switch is configured as:

Reset	IP1	IP2	IP3	IP4	DHCP

Where the address of the FPGA board is calculated as $ADDR = IP1 + 2\,IP2 + 4\,IP3 + 8\,IP4$. The other switches are not used in the current configuration. The ip address is 192.168.0.220 + ADDR and the MAC address is 00:01:ca:22:22:ADDR .

The chain boards The chain boards for the DAC and the DDS are configured by a DIP switch. The pin configuration is as follows:

Addr0	Addr1	Addr2	Addr3

The address is calculated as $Addr0 + 2\,Addr1 + 4\,Addr2 + 8\,Addr3$.

The DDS evaluation board On the dds evaluation board the parallel board connector U4 has to be removed. The jumper W2 has to be set to external mode.

The DAC evaluation board The output transformer T1 has to be removed and the solder bridge JP8 has to be closed. The DAC clock has to be controlled externally, therefore the jumper JP2 has to be set and the solder bridge JP3 has to be closed.

2.4 The LabView interface

In this section the interface between the python server discussed above and the experiment control software is described. As for the programming of the sequence there exist two different modes:

- Parallel environment: Build to be compatible with the old qfp program and using the Matlab sequence files
- Sequential environment: Works with the new qfp program and uses python sequence files

The communication is based on a plain text transmission via a TCP socket, where the listening program (server) is the python environment and the sending program is the LabView experiment control program. The standard TCP port is 8880.

2.4.1 Parallel environment

In the parallel environment LabView just sends the whole pulse sequence as a string to the server. All TTL and RF pulses are calculated in the LabView program. There are no additional global variables.

2.4.2 Sequential environment

In sequential mode the transmission from LabView to python consists of a command string with information about Rabi times, RF amplitudes and the sequence file Name. The sequence file is then read out and compiled by the python server. The sequence file format is described above.

General format The server accepts strings in the following format:

`command1,option1_1,option1_2;command2,option1,option1`

a simple example:

`NAME,test_ttl.py;TRIGGER,NONE;FLOAT,duration,3.4;`

The available variables are:

Variable name	Description	Usage
NAME	The name of the sequence file	NAME,file_name
INT	Sends an integer value to the script	INT,var_name,value
FLOAT	Sends a float value to the script	FLOAT,var_name,value
STRING:	Sends a string value to the script	STRING,var_name,value
BOOL	Sends a Boolean value to the script	BOOL,var_name, value
TRIGGER	Gives the type of the trigger.	TRIGGER,trig_string

Possible Trigger strings are: NONE , LINE . The commands are defined in the file `innsbruck/handle_commands.py` .

The transition object The transition object is defined as follows:

TRANSITION, transition_name ;RABI, Rabi_times ;
SLOPE_TYPE, slope_type ;SLOPE_DUR, slope_duration ;
AMPL, slope_ampl ;FREQ, frequency ;IONS, ion_map ;
AMPL2, second_amplitude ;FREQ2, second_frequency ;

Where Rabi frequencies for multiple ions are defined as:
1:19.34 , 2:21.12 , 3:20.34 , 4:22.67
And the ion_map:
1:101 , 2:102 , 3:103 , 4:104

The default transition There is the possibility to define a default transition which is used if no transition is given in the R729 command:

DEFAULT_TRANSITION,transition_name;

2.4.3 Creating pulse commands

The functions intended for use in user level are defined in `user_function.py` or in `api.py` . There exists a framework for handling global variables in a sequence and transferring this variable back to LabView.

An example user mode function is the PMT Detection function:

```
class PMDetection(PulseCommand):
  def __init__(self,detect_wait,CameraOn=False):
    configuration=self.get_config()
    detection397=configuration.detection397
    PMTrigger=configuration.ion_trig_device
    PMGate=configuration.PMGate
    trigger_length=configuration.PMT_trigger_length
    detection_count=self.get_variable("detection_count")
    self.set_variable("detection_count",detection_count+2)
    self.add_to_return_list("PM Count","detection_count")
    seq=TTL_sequence()
    ttl_pulse(detection397,detect_wait,is_last=False)
    ttl_pulse(PMGate,detect_wait,is_last=False)
    seq.add_pulse(PMTrigger,trigger_length,is_last=False)
    if CameraOn:
      ttl_pulse(trigger_length,configuration.Detection,is_last=False)
      ttl_pulse(trigger_length,configuration.CameraTrigger,is_last=False)
      ttl_pulse(PMTrigger,trigger_length,
          start_time=detect_wait-trigger_length,is_last=False)
```

This function can be divided in following parts:

- class definition
- variable definition from the configuration in in `__init__.py`
- handling of the return variables
- the main ttl sequence

This function generates the TTL pulses required for a detection sequence. It generates different pulses if a CCD camera is present. Additionally it uses the variable "detection_count" to send information about the number of detection pulses to LabView.

class definition for user level functions The class definition has to refer to the parent class (PulseCommand) and the method (function) that is executed when the class is called is __init__.

```
class PMDetection(PulseCommand):
  def __init__(self,detect_wait,CameraOn=False):
```

It is possible other methods if local variables are required. A sample code that increases a variable and returns it to LabView may look like:

```
class counter1(PulseCommand):
  def __init__(self,value=0):
    self.value=value
  def incme(self):
    self.value+=self.increase
    def return_to_labview(self):
    self.set_variable("counter",self.value)
    self.add_to_return_list("COUNTER 1","counter")
```

it is used in python like:

```
inc1=counter1()
inc1.incme()
...
...
if CameraOn
  inc1.incme()
...
...
inc1.return_to_labview()
```

2.4.4 Returning values to LabView and using global variables

To return a value to LabView you have to create a global variable:

```
detection_count=self.get_variable("detection_count")
self.set_variable("detection_count",detection_count+1)
self.add_to_return_list("PM Count","detection_count")
```

First the current value of the variable is obtained. Then the variable is increased and returned back to the global variable. Then the global variable is added to the return list. If the variable is a list the values a returned separated by commas.

2.4.5 User function framework

The supplied methods for user functions are

```
self.get_config()
```

Returns the global configuration class which is defined in innsbruck/__init__.py

```
self.set_variable(variable_name,value)
```

Sets a global variable. You can assign all types **except** dictionaries !!!

```
self.get_variable(variable_name,[default_value])
```

Returns the value set by set_variable().

If the variable is not set yet it returns default_value. If default_value is omitted the default_value is 0.

`self.add_to_return_list(Pre_String, variable_name)`

Adds the variable to the LabView return list. The variable is returned as:

`"Pre_String,value;"`

or if the variable is a **list** [var1,var2,var3,...]:

`"Pre_String,var1,var2,var3,...;"`

`self.get_error_handler()`

Returns the error handler function to return error messages to LabView.

`self.get_cycle_time()`

Returns the clock cycle time in microseconds.

`self.address_ion(ion,[start_time])`

Checks if the current ion is the parameter ion . If the ion has to be changed it invokes self.ion_event() . It handles the ion return list as well

`self.ion_event()`

A pointer to the TTL event class which switches the ions. Normally this is parallel_ttl () within TTL_sequence this is self.add_pulse

2.5 Internals of the Software

In this section some details of the python compiler are described. For a general overview see section(1.1).

File locations

The directory tree of the compiler is divided in the following directories

`sequencer`	Common parts of the compiler
`sequencer/pcp`	Definition and generation of the pcp commands
`sequencer/pcp/machines`	Implementation and generation of the instructions
`sequencer/pcp/events`	Abstract definition of the events called by the API
`sequencer/pcp/instructions`	Abstract definition of the pcp32 instructions
`sequencer/devices`	Definition of the hardware devices (DDS,DAC)
`sequencer/ptp/`	Definition and implementation of the PTP protocol
`test_config`	Definition of the API and configuration (for all experiments)
`innsbruck`	Definition of additional API and special configuration

The object structure

The compiler is based on an object oriented structure where the main objects are:

sequencer	The sequencer main object.
test_config	Includes the DDS and DAC and TTL objects.
innsbruck	Consists of additional API commands and the end user layer.

Overview

The sequencer object

The most important methods of the sequencer object are:

current_sequence	Methods and variables for managing the array of abstract events.
standard_params	Common constants and definitions.
main_program	Array for the end user layer events

The test_config object

first_sequencer	Object for the FPGA board with ptp address 1. Includes methods for compiling the sequence and sending it over PTP.
first .sequencer.machine	The abstract machine object. Includes methods for compiling the events to machine code.
first_dac_device	Abstract hardware object for the DAC with chain board address 1
second_dac_device	Abstract hardware object for the DAC with chain board address 2
dds_devices	Array of abstract hardware objects for the DDS. The index corresponds to the chain boards address

The innsbruck object

start_server()	Method to initialize the server and the compiler.
error_handler()	Method for returning error messages to Labview

3 Internals of the Programmable pulse generator

In this section not all aspects off the programmable pulse generators are covered. For a complete description of the concepts of the programmable pulse generator the reader is referred to reference [1].

3.1 The firmware

Compiling the firmware

The firmware is described in the hardware description language VHDL. The source code files available on the project homepage are macro files which itself generate the VHDL

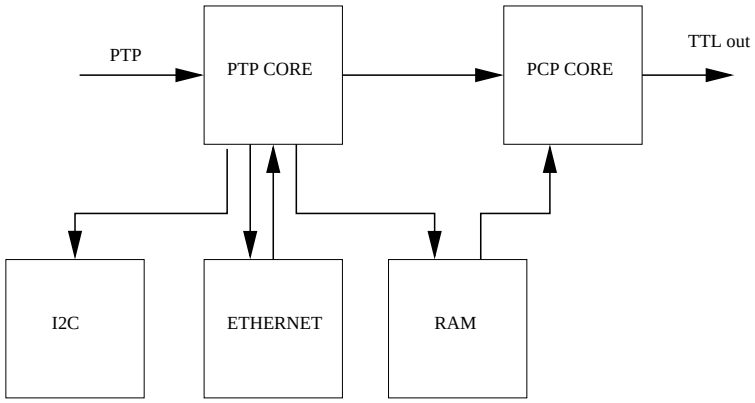

Figure 8: Overview over the firmware of the programmable pulse generator

source files. This additional step has the advantage that if changes on an object is made, this changes are made in all VHDL files containing this object. For compilation of the VHDL sources the Quartus II design suite available on the Altera homepage[23] is required. The source code may be obtained from the CVS repository of the pulse sequencer project homepage[24]. Additionally the GNU make tools and the M4 scripting interpreter are required. For the windows operating system these are provided by the cygwin[25] environment. To generate the Quartus II project file open a command window and type `make sequencer_top.vhd` . and `make sequencer_top.map.eqn`. This creates a file called sequencer_top.qpf which could be loaded in Quartus II and compiled by pressing ctr+L .

Programming the FPGA The FPGA may be programmed with the programming software provided by the Quartus II design suite. The cable should be connected to the config port on the FPGA port. In the programming software choose the ByteBlaster II cable and the active serial programming mode. The file to program is `sequencer_top.qpf` .

Overview over the firmware

A general overview of the programmable pulse generator is given in section(1.1.3). The firmware consists of different blocks which are shown in Fig. 8. These blocks are interconnected with an system-on-a-chip bus. The "Wishbone" bus standard as defined by the Opencores[26] community is used. The bus system is described in detail in the Master's thesis of Paul Pham [1].

The network communication

The programmable pulse generator has two different possibilities for communication:

[23] www.Altera.com
[24] htttp://pulse-sequencer.sf.net
[25] http://www.cygwin.com
[26] http://www.opencores.org

OSI Layer	Function	Network Stack	Daisy-chain Stack
Physical	Connector interface	Ethernet PHY, RJ45	LVDS, RJ45
Datalink	Logical Link	Ethernet MAC	Daisy chain link
Network	Global Addressing	IP	PTP routing
Transport	Multiplexing	UDP, TCP	None
Application	API for high level functions	DHCP, PTP server	PTP server

Table 5: The OSI layers of the network communication methods.

- The Network Stack for communicating with the experimental control computer
- The Daisy-chain Stack for communicating with other programmable pulse generators.

The Daisy-chain stack is only required if two ore more programmable pulse generators are synchronized and programmed by the same experiment control computer. In our setup this feature is not used. Both stacks comply to the standard OSI layer model which are shown in Tab. 5. The layers of the Network Stack are the standard Internet layers as used by normal personal computers. The Daisy-chain Stack uses the same connectors and cabling. In difference to the Ethernet specification, the Daisy-chain physical layer is half-duplex to make clock recovery and synchronization easier. The Datalink layer is described by the pulse transfer protocol (PTP) frame definition. A PTP frame is shown in Tab. 6. It consists of fields containing the source and destination addresses, the firmware version, the PTP opcode, and the payload. The Network layer handles the routing of this PTP package depending on the values in the `Destination ID.` field. The highest layer is the PTP server which interprets the PTP Opcode and communicates with the other blocks in the programmable pulse generator. The possible Opcodes are shown in Tab. 7. The experiment control computer sends a PTP frame to the programmable pulse generator over the UDP connection. These packets are either routed to other programmable pulse generators in the PTP chain or interpreted by the PTP server. The PTP protocol is described in more detail in the Master's thesis of Paul Pham [1].

The PCP Core

The pulse control processor (PCP) core reads the instruction words from the RAM, interprets it and controls the outputs of the FPGA. In Paul Pham's Master's thesis the PCP0 instruction set is described while in the current setup the PCP32 instruction set is used. The main difference is the width of the instruction word. A PCP0 instruction word is 64

Field	Source ID	Dest. ID	Major Ver.	Minor Ver.	PTP Op-code	Zero	Total Length	Unused	Payload
Octets	1	1	1	1	1	1	2	2	variable
Address	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x8	0xA

Table 6: A pulse transfer protocol frame.

Opcode	Name	Function
0x00	Null	The PTP packet is ignored.
0x01	Status Request	The PPG sends a standard reply to the source of the package.
0x11	Status Reply	The answer to a Status Request.
0x02	Memory Request	Requests a RAM read or write function. The payload contains the data to be written.
0x12	Memory Reply	Answer to Memory Request. The payload contains either a write success word or the data to be read.
0x04	Start Request	Starts or Stops the various blocks of the PPG.
0x14	Start Reply	The answer to a Start Request.

Table 7: The pulse transfer protocol Opcodes.

bit wide whereas the PCP32 instruction set is 32 bit wide. The PCP0 instruction set has also no support for phase coherent switching. A block diagram of the PCP core is shown in Fig. 9. The Decoder interprets the instruction word and generates control signals for the other blocks. The Program Counter keeps track of the current RAM address and handles program control flow instructions, and the Phase Registers are needed for phase coherent switching. The opcodes for the PCP32 are shown in Tab. 8. The Opcodes for the PCP32 core are documented below. The 4 most significant bits of the 32 bit instruction word are reserved for the opcode. The remaining 28 bits are referred to as [27:0]. (If a function uses the lowest 8 bits it is denoted as [7:0])

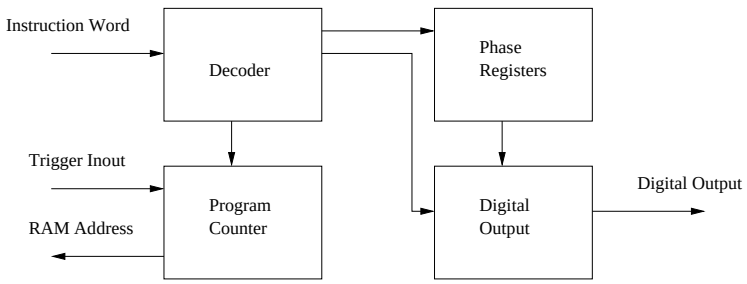

Figure 9: Block diagram of the PCP32 core.

Instruction	Opcode	Description
nop	0x0	No operation
btr	0x3	Branch on trigger
jump	0x4	Jump to address
call	0x5	Subroutine Call
return	0x6	Subroutine return
halt	0x8	Halt pcp
bdec	0xA	Branch decrement
ldc	0xB	Load constant
p	0xC	Pulse immediate
pp	0xD	Pulse phase
lp	0xE	Load phase

Table 8: Instruction set of the pcp32

nop

Opcode: 0x0

Name: No Operation

Function: Does nothing

Parameter: None

btr

Opcode: 0x3

Name: Branch on trigger

Functions: Jumps to the address given if the Trigger input matches the given data.

Parameter: Trigger word [27:19]

Parameter: Address [18:0]

jump

Opcode: 0x3

Name: Jump

Functions: Jumps to the given address.

Parameter: Address [18:0]

call

Opcode: 0x5

Name: Call Subroutine

Functions: Calls a subroutine at a given address.

Parameters: Address [18:0]

return

Opcode: 0x5

Name: Return Subroutine

Functions: Returns from a subroutine to the former address.

Parameters: None

halt

Opcode: 0x8

Name: Halt PCP

Functions: Stops the PCP core.

Parameters: None

bdec

Opcode: 0xA

Name: Branch decrement

Functions: Decrements the loop register and branches to the given address if the value in the register is bigger than zero.

Parameters: Address [18:0]

Parameters: Register Address [27:23]

The PCP32 has 64 registers for different finite loops. The Register Address word selects the register.

ldc

Opcode: 0xB

Name: Load constant

Functions: Loads a constant to the loop register.

Parameters: Constant [7:0]

Parameters: Register Address [27:23]

The PCP32 has 64 registers for different finite loops. The Register Address word selects the register.

p

Opcode: 0xC

Name: Pulse immediate

Functions: Sets the value of the given part of the output registers to the given value

Parameters: Value [15:0]

Parameters: Output select [17:16]

The output select selects the part of the 64 bits of the LVDS output which is controlled. If the value is 0 then the bits [0:15] are set if it is 1 then [16:31] are set, etc.

pp

Opcode: 0xD

Name: Pulse Phase

Functions: Adds the phase offset to the current value of the addressed phase accumulator and sets the corresponding output bits.

Parameter: Phase Offset [15:0]

Parameter: Byte select [16]

Parameter: Current [20]

Parameter: Phase Addend [21]

Parameter: Phase register address [27:23]

As the data bus for the DDS is only 8 bits wide, the 12 bit phase word has to be written in two different write cycles. The Byte select bit selects if the lower 8 or the upper 4 bits are written to the DDS.

lp

Opcode: 0xE

Name: Load Phase

Functions: Sets the value of the frequency tuning word of the given phase register.

Parameter: Phase Value [15:0]

Parameter: Byte select [16]

Parameter: Phase wren [22]

Parameter: Phase register address [27:23]

As the frequency tuning word is 32 bits wide it has to be splitted up in two 16 bit words.

3.2 Pin configuration of the LVDS Bus

In Tab. 9 the pin configuration of the LVDS bus in the actual setup is shown. The digital output form pin 0 is not used. Also the digital outputs ranging from pin 32 to pin 47 are not used in the actual setup due to firmware limitations.

Pin	Function	Device
0	Digital Out	
1	WRB	DAC
2-15	D0 - D13	DAC
16	PSEN	DDS
17	WRB	DDS
18-23	A0 - A5	DDS
24-31	D0 - D7	DDS
32 - 47	Digital Out	
48	IO Update	DDS
49,50	PS1, PS0	DDS
51-54	BA0 - BA3	DDS chainboard
55-59	Digital Out	
60-63	BA0 - BA3	DAC chainboard

Table 9: Output pin configuration of the FPGA

4 Bibliography

References

[1] P. Pham, *A general-purpose pulse sequencer for quantum computing*, Master's thesis, Massachusetts Institute of Technology (2005).

[2] H. Nyquist, *Certain topics in telegraph transmission theory*, Proceedings of the IEEE **90**, 280 (2002).

[3] *Analog Devices: A Technical Tutorial on Digital Signal Synthesis*, available at: http://www.analog.com/dds (1999).

[4] *Analog Devices: AD9858 Datasheet*, available at: http://www.analog.com/dds (2004).