Pigeon 3 Matlab Code Documentation

Version 1.0, updated 19/07/10

Contents

List of Figures

1 Introduction

1.1 The Platform Structure

This document comes to describe the Pigeon Matlab interface. The Pigeon software pack includes several entities (as can be seen in Fig.1) that negotiate between the user defined experiment protocol and the FPGA platform which eventually runs the experiment. For these experiments the FPGA card can be considered as the experiment CPU that controls in real time over the involved instruments. However, this FPGA CPU does not control over the lasers locking circuits, it is done by another FPGA card.

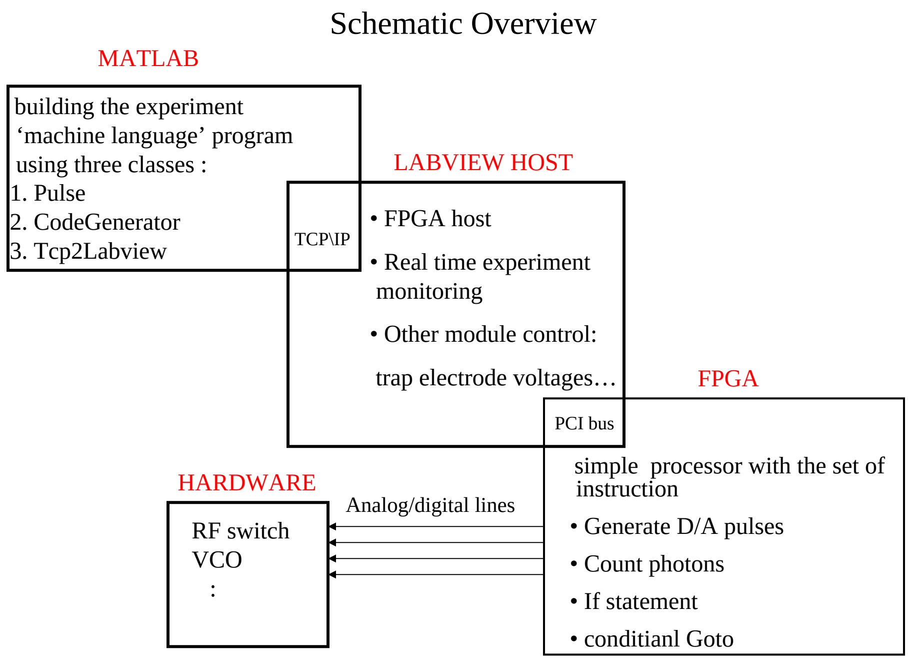


Figure 1: Schematic chart of the Pigeon code structure and entities.

Each of the entities that are involved in the programming process can be imagined as a layer. The top layer is the Matlab interface which gives the user a set of methods, gathered in Matlab classes (a manifestation of object oriented programming in Matlab) that span the envelope of the CPU capabilities.

1.2 The Matlab Interface

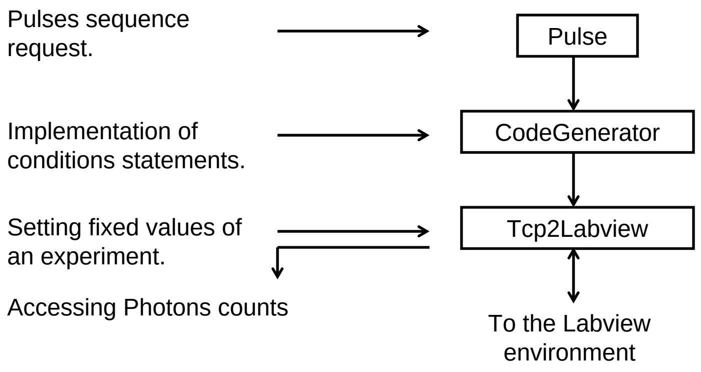

Pulses sequence request.
Implementation of conditions statements.
Setting fixed values of an experiment.
Accessing Photons counts
Pulse
CodeGenerator
Tcp2Labview
To the Labview environment

Figure 2: Experiment implementation using the matlab interface. On the left appears a list an experiment building blocks. On the right appear the interface classes that handle the matched functionality.

The matlab interface code purpose is to translate user defined commands (equivalent to programming language commands) to an instructions code (equivalent to a machine code) which can be executed by the CPU. As can be seen in Fig.2 different commands are addressed to different classes of the interface.

2 Classes

This section comes to give a comprehensive description of the classes from which the interface to the experiment executing FPGA is made. Each class includes:

1. Constants that usually defines the possible settings of the class.
2. Variables that host the user defined parameters of the class.
3. Methods that translate the user definition to standard executable code.

This section does not includes description of the classes: Stack, Sequence, NumControl. That is because these classes come to assist the other classes and their structure is not relevant for the programming interface (like the full code implementation of the interface methods).

2.1 Pulse

The Pulse class is the building blocks of the experiment sequence as each pulse object is translated to some hardware operation (as RF switch, AOM, PMT, etc)which start and stop at specified time. The various pulses properties are define in the function PulseChannelInfo as a persistent variable (starting from this version the data base is not define in the Pulse class). Each pulse is a structure in a cell array called 'info'. Each stracture/Pulse MUST have the fields : 'ChannelName' and 'ChannelType'. In addition any number of fields can be added. These properties are used by the class CodeGenerator to implement in hardware the required pulse.

2.1.1 Variables

1. Channel
2. Tstart
3. Tend
4. Freq=-1;
5. Phase=-1;

These variable are initialize by the constructor. 'channel' is the index in the cell array in PulseChannelInfo which correspond to this pulse. The Pulse timing is store in 'Tstart' and 'Tend' in units of clock cycles. 'Freq' and 'Phase' can be use for channels that their freq and phase can be set.

2.1.2 Methods

1. Pulse

   Syntex obj = Pulse(ch,ts,width,varargin)

Input **ch** - [String] Channel name.
ts - [Float] Pulse starting time [μsec] in the sequence frame.
width - [Float] Pulse duration [μsec]. Zero (0) input keeps the beam on without shutting it down. -1 sets the beem off.
varargin - Optional extension for the definition of the pulse frequency ('freq',X) and/or phase shift ('phase',X).

Output textbfobj - Pulse object.

Description Constructor of the Pulse object.

2. Shift

Syntex P=Shift(obj,t)

Input **obj** - Pulse object.
t - [Float] Time shifting amount [μsec].

Output **P** - Pulse object.

Description Shifts a given pulse in time and returns a Pulse object which is identical to the input just shifted in time.

2.1.3 Static Methods

1. Sequence2TimeLine

Syntex timearray = Sequence2TimeLine(pulses)

Input **pulses** - Pulse objects vector.

Output **timearray** - Array of number of pulses by 4, each line follows [time channel operation parameter]. The "operation" number corresponds the settings: start (1), stop (2), setfreq (3) ,setphase(4), do nothing(5).

Description Returns an array which contain the sorted operations list which is derived from the input pulses list.

2. PlotTimeLine

Syntex PlotTimeLine(t)

Input **t** - ?.

Output None

Description Plots a graph which is somehow related to the pulses of the sequence.

2.2 CodeGenerator (handle class)

This class "compiles" the experiment sequence to a list of instructions to be executed by the FPGA. This class is a set of various macros for implementing all various operations as digital outputs, photon counting, logic statement(If-Else) and loops. The product of this class is an executable program which is arranged in an array ('Code' class variable) as a list of four columns of integers (16 bits). This array is sent to the FPGA host by the Tcp2Labview class.

2.2.1 Constants

1. CommandList - Cell array that lists the FPGA instructions.
   Do nothing
   Analog out
   Digital out
   Photon count
   Register
   If
   Goto T/F
   Push to FIFO
   End program
   DDS Prog.

2. SubcommandList see the matlab file.

   This data structure is ONLY used by the 'DisplayCode' method to translate the 'machine language' to assembler like code. The content here has no effect on the FPGA and is only a mirror image of it (caution - may not be updated !!).

3. NumOfDDS - Integer greater than 0 that specify the number of DDS that are operated by the CPU.

4. DDSInternalClockFreq - The DDS's internal oscillator frequency (28.147497 MHz).

5. DDSExternalFrequency - The frequency of the external clock which is fed into the DDS [MHz].

6. DDSPLLRatio - Integer, the DDS internal PLL value [$4 \div 20$].

7. DDSFrequencyLimit - Two values vector that limits the frequencies that are allowed to be set by the user.

2.2.2 Variables

1. code - An array that contains the 'machine language' code having the following form: [command(byte) subcommand(byte) par1(int16) par2(int16)]

2. currentline - Indicates the updated relevant line for programming.

3. stack - Assists in following the IF-ELSE statements indexing.

4. numofreadout - should indicate the number of times a push to FIFO command appears in the code. !!! Caution - cannot be trusted!!

5. DDSCurrentState - An array (length NumOfDDS) of records, each with the following fields: **IntRegMap** - an array of 40*8 binary entries which reflects the current registers state of the DDS (parallel to the AD9854 registers map on the manual). **IOPortBuffMap** - an array parallel to IntRegMap which encapsulate the values of the IO port registers before they are transferred to the inrenal ones. **LegsValues** - a 1*4 binary array which reflects the value of each DDS relevant legs (1 - Reset, 2 - FSK, 3 - WRB, 4 - Ioud). The hole structure is initiated to zeros on the object construction.

6. DDSBusCurrentAddress - Six bits vector that encapsulate the current binary values of the address bus.

7. DDSBusCurrentData - Eight bits vector that encapsulate the current binary values of the data bus.

2.2.3 Methods

1. CodeGenerator

   Syntex obj = CodeGenerator

   Input None.

   Output textbfobj - CodeGenerator object.

   Description Constructor of the CodeGenerator object. Zeros the DDSCurrentState structure.

2. GenSeq

   Syntex GenSeq(obj,arrayofpulses,varargin)

   Input **arrayofpulses** - Vector of Pulse objects.
   varargin- [float] The inital time for the sequence of pulses.

   Output None.

   Description generate the code for implementing the list of pulses. All the pulses in vector refer to the same time base with the same starting point.

3. GenPhotonTimes

   Syntex GenPhotonTimes(obj,duration)

   Input **duration** - [float] Photon gathering time [μsec].

   Output None.

   Description Preforms monitoring of arriving photons by recording the time stamp of arrival. The result of this instruction is a series of time stamps, each stands for photon count (on both PMTs), separated by sequences of zeros. The zeros buffers between different requests for photon scattering monitoring along the program. For example, if along the program two calls for photon gathering were done than on the output record (given by Tcp2Labview.ReadOut) there will be two sequences of time counts separated by a single sequence of zeros. The time stamp which is given to each count is measured from the beginning of the detection sequence.

4. GenPhotonPhase

   Syntex GenPhotonPhase(obj,duration)

Input **ch** - [String] Channel name.

Output None.

Description The same as GenPhotonTime only that by this method the time stamps are measured form the recent zero crossing of the trap oscillating wave. Hence, a relative phase between the photon scattering rate sine and the trap feed can be extracted.

5. GenWait

   Syntex GenWait(obj,c)

   Input **c** - [Integer] Clock cycles.

   Output None.

   Description Embeds waiting period in the code. The resulting waiting time (under the FPGA clock frequency of 40MHz) is: $t = 100c + 25$ [nsec].

6. GenPause

   Syntex GenPause(obj,c)

   Input **c** - [float] Time in [μsec].

   Output None.

   Description replace the GenWait. Embeds waiting period in the code in a single line(using FPGA hardware) has a time resolution of a single clock cycle(25 nS).

7. GenWaitExtTrigger

   Syntex GenWaitExtTrigger(obj)

   Input None.

   Output None.

   Description Holds the program until external trigger appear.

8. GenRepeatSeq

   Syntex GenRepeatSeq(obj,arrayofpulses,c)

   Input **arrayofpulses** - Vector of Pulse objects.
   c - [Integer] Number of loops.

   Output None.

   Description Similar to GenSeq only allows repetition over the pulses sequence several times.

9. GenIfDo

   Syntex GenIfDo(obj,expr,c)

   Input **expr** - [String] Condition expression, currently just 'PhtonCount¡='.
   c - [Integer] Number of photons count.

   Output None.

Description Implements condition statement in the code. In term of the machine code this methods compares between registers and skips of executes a code segment, conditioned upon the comparison result. Although the method needs the line number of the code for which it has to skip in case the condition was not fulfilled it cannot be declared when using the method. This is because the length of the code, which is executed if the condition is fulfilled is unknown, yet. Therefore, this method must be followed by the GenElseDo and GenElseEnd commands that fill, retroactively, the missing addresses at the skipping instructions.

10. GenElseDo

    Syntex GenElseDo(obj)

    Input None.

    Output None.

    Description Cuts the code segment which is executed in case the condition, which was defined in the GenIfDo statement, is fulfilled.

11. GenElseEnd

    Syntex GenElseEnd(obj)

    Input None.

    Output None.

    Description Seals the If - Else statement.

12. GenFinish

    Syntex GenFinish(obj)

    Input None.

    Output None.

    Description Terminates the program.

13. GenDDSBusState

    Syntex GenDDSBusState(obj,address,data)

    Input **address -** [String] Two letters of the desired register address (in Hex basis).
    data - [String] Two letters of the desired register data (in Hex basis).

    Output None.

    Description Sets values for the address and data but which is in common for all the DDSs. This command modifies non of the DDSs registers. Registers values are set only by the following WRB and Ioud latches (that are aimed for a specific DDS).

14. GenDDSResetPulse

    Syntex GenDDSResetPulse(obj)

    Input None.

    Output None.

Description Resets all the DDSs by raising the voltage over the reset leg of the chips.

15. GenDDSFSKState

    Syntex GenDDSFSKState(obj,DDSNum,value)

    Input **DDSNum -** [Integer] The number of the destination DDS.
    value - [Integer] The desired value of the FSK (0 or 1).

    Output None.

    Description Sets the value of the FSK leg on a given DDS. This flag allows rapid switching between on DDS operation state (frequency / phase) to a different one. The flag is irrelevant for single tone mode of operation.

16. GenDDSWRBPulse

    Syntex GenDDSWRBPulse(obj,DDSNum)

    Input **DDSNum -** [Integer] The number of the destination DDS.

    Output None.

    Description Transfers information from the collective bus to buffers of the defined DDS.

17. GenDDSIOUDPulse

    Syntex GenDDSIOUDPulse(obj,DDSNum)

    Input **DDSNum -** [Integer] The number of the destination DDS.

    Output None.

    Description Transfers information from the buffer of a DDS to its' internal registers (and by that orders the DDS).

18. GenDDSInitialization

    Syntex GenDDSIOUDPulse(obj,DDSNum,modeNum)

    Input **DDSNum -** [Integer] The number of the destination DDS.
    modeNum - [Integer] The desired operation mode number for the relevant DDS.

    Output None.

    Description Sets the mode of operation for a specific DDS. The modes index: 0 - Single tone, 1 - Two frequencies hopping (FSK), 2 - Continuous scan between two frequencies (Ramped FSK), 3 - Monotonic frequency increase (Chirp), 4 - Two phases hopping (BPSK).

19. GenDDSFrequencyWord

    Syntex GenDDSFrequencyWord(obj,DDSNum,wordNum,freqValue)

    Input **DDSNum -** [Integer] The number of the destination DDS.
    wordNum - [Integer] The number of the destination frequency word (1/2).
    freqValue - [double] The frequency word content [MHz].

    Output None.

Description Fills the six chosen DDS registers with a value which correspond to the given freqValue. Each DDS contain two frequency words. The modes Single tone, Chirp, BPSK take just the first frequency word while the FSK, Ramped FSK modes require two frequencies words definition.

20. GenDDSPhaseWord

    Syntex GenDDSPhaseWord(obj,DDSNum,wordNum,phaseValue)

    Input **DDSNum -** [Integer] The number of the destination DDS.
    wordNum - [Integer] The number of the destination phase word (1/2).
    phaseValue - [double] The phase word content [radians].

    Output None.

    Description Fills the four chosen DDS registers with a value which correspond to the given phaseValue. Each DDS contain two phase words. Only the BPSK mode of operation requires the definition of the two frequency word, between which the DDS jumps.

21. GenDDSSweepParameters

    Syntex GenDDSSweepParameters (obj,DDSNum,DFW,dwellPeriod)

    Input **DDSNum -** [Integer] The number of the destination DDS.
    DFW - [double] The frequency word content [MHz].
    dwellPeriod - [double] dwell time word content [μsec].

    Output None.

    Description Fills the Difference Frequency Word (DFW) and the dwell period word of the chosen DDS. This command is relevant for the Ramped FSK and the Chirp modes of operation. Both these modes employ frequency scan (whether between two predefined frequencies or from a single frequency and on). The DFW and the dwell period defines the resolution of each frequency step and its time duration correspondingly. This command, together with the GenDDSToggleACC2 command, is essential for the execution of these two modes.

22. GenDDSIPower

    Syntex GenDDSIPower(obj,DDSNum,pwr)

    Input **DDSNum -** [Integer] The number of the destination DDS.
    pwr - Precentage of the DDS maximal output power [$0 \div 100$].

    Output None.

    Description Scales the DDS output power between 0 (DDS is shut down) to 100 (full output power).

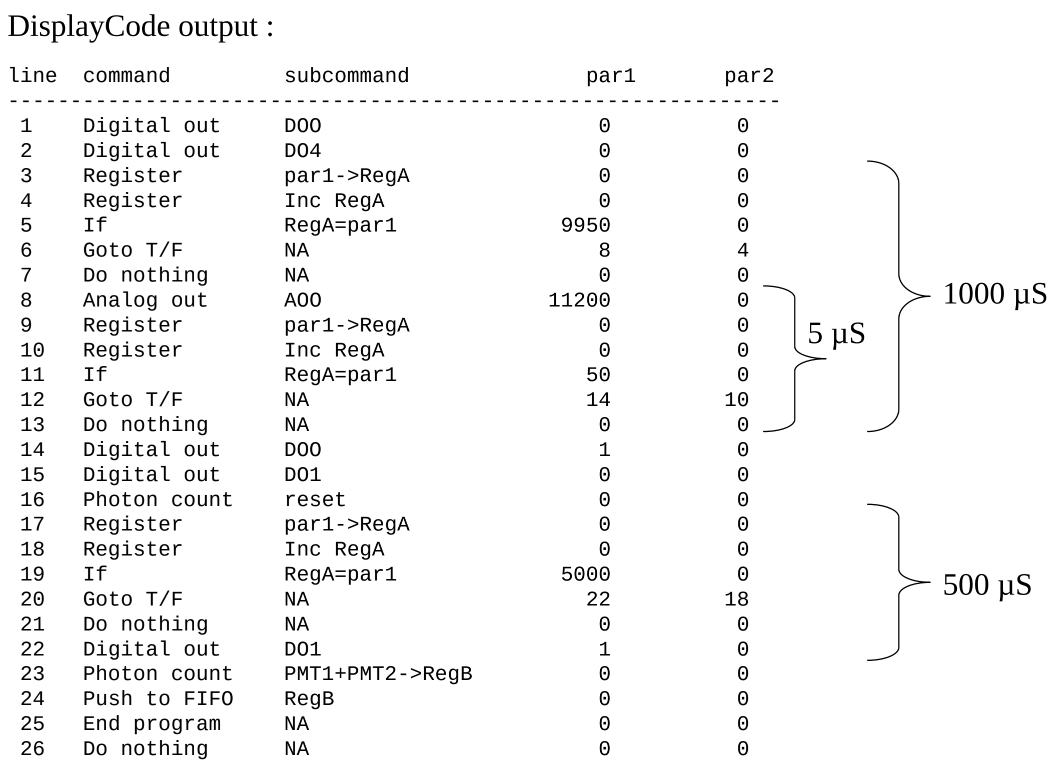
23. DisplayCode

    Syntex DisplayCode(obj)

    Input None.

    Output None.

    Description Displays textual plot of the code based on the command and subcommand lists. An example of such a plot can be seen in Fig.3.



DisplayCode output :

```
line  command         subcommand              par1       par2
-------------------------------------------------------------
 1    Digital out     DO0                        0          0
 2    Digital out     DO4                        0          0
 3    Register        par1->RegA                 0          0
 4    Register        Inc RegA                   0          0
 5    If              RegA=par1               9950          0
 6    Goto T/F        NA                         8          4
 7    Do nothing      NA                         0          0
 8    Analog out      AO0                    11200          0
 9    Register        par1->RegA                 0          0
 10   Register        Inc RegA                   0          0
 11   If              RegA=par1                 50          0
 12   Goto T/F        NA                        14         10
 13   Do nothing      NA                         0          0
 14   Digital out     DO0                        1          0
 15   Digital out     DO1                        0          0
 16   Photon count    reset                      0          0
 17   Register        par1->RegA                 0          0
 18   Register        Inc RegA                   0          0
 19   If              RegA=par1               5000          0
 20   Goto T/F        NA                        22         18
 21   Do nothing      NA                         0          0
 22   Digital out     DO1                        1          0
 23   Photon count    PMT1+PMT2->RegB            0          0
 24   Push to FIFO    RegB                       0          0
 25   End program     NA                         0          0
 26   Do nothing      NA                         0          0
```

Figure 3: Typical plot of CodeGenerator.DisplayCode method.

24. get.codenumoflines [?]

    Syntex value = get.codenumoflines(obj)

    Input None.

    Output None.

    Description Returns the code number of lines.

2.2.4 Static Methods

None

2.3 Tcp2Labview

This class functions as a communication socket between the matlab environment and the FPGA host.

1. It transmits the compiled machine code to the shared memory region of the host program and orders it to execute the FPGA target program.
2. It pulls data from the shared memory region on the host program.
3. It sets the values of parameters that are fixed along the experiment.

2.3.1 Constants

None.

2.3.2 Variables

1. TcpID
2. UseOpenChannel

2.3.3 Methods

1. Tcp2Labview

   Syntex obj=Tcp2Labview(computerip,port)

   Input **computerip-**TCP address of the host computer.
   port- [Integer] Port number of the vi server.

   Output **obj -** Tcp2Labview object.

   Description Constructor of the Tcp2Labview object.

2. Delete

   Syntex Delete(obj)

   Input None.

   Output None.

   Description Terminates Tcp2Labview object.

3. UploadCode

   Syntex intsend=UploadCode(obj,codeobj)

   Input **codeobj -** The code matrix of a CodeGenerator object.

   Output None.

   Description Copies the program to the shared memory on the host vi.

4. UpdateFpga

   Syntex UpdateFpga(obj)

Input None.

Output None.

Description Orders the host to copy the shared memory to the targret vi.

5. Execute

   Syntex Execute(obj,repeatnum)

   Input **repeatnum -** [Integer]

   Output None.

   Description Orders the target vi to begin with the execution of the program. The number of following executions is set by the repeatnum value.

6. ReadOut

   Syntex output=ReadOut(obj,num)

   Input **num -** [Integer]

   Output None.

   Description Reads num cells of the shared memory segment which is dedicated to the target-host FIFO date. The target pushes photon counting data to the FIFO and the host places this data at the accessed segment of the shared memory. For num ¡=0 all the FIFO data is read. [?] over flow of num, the 1000th cell doesn't tell how much data there is?

7. UpdateTrapElectrode

   Syntex UpdateTrapElectrode(obj,DCL,DCR,Commensation,EndcapS,EndcapN)

   Input

   Output None.

   Description A method to set the constant voltage over the trap electrodes. [?] units.

8. UpdateRAP

   Syntex UpdateRAP(obj,duration,amp)

   Input None.

   Output None.

   Description Not in use.

9. ReadNoiseEaterDetector

   Syntex out=ReadNoiseEaterDetector(obj)

   Input None

   Output Analog Input1 .

   Description return the value of AI1 that should indicate the 674 intensity on the noise eater detector.

10. UpdateBfield

Syntex UpdateBfield(obj,v)

Input **v** - [array of float].

Output None.

Description Set the values of the feedforward.

11. SetAO7

    Syntex SetAO7(obj,c)

    Input **c** - [float]

    Output None.

    Description A method to set the setpoint of the 674 noise eater. By that the power level of the regulated beam can be determinded. [?] units.

12. GetLasersStatus

    Syntex status=GetLasersStatus(obj)

    Input None.

    Output None.

    Description Not in use.

13. WaitForHostIdle

    Syntex WaitForHostIdle(obj)

    Input None.

    Output **r** - [Boolean]

    Description Indicates whether or not the stack is empty. [?]

2.3.4 Static Methods

None.

3 Code Examples

3.1 Resonance Fluorescence

```
% defining the pulse
init=Pulse(Repump1092',0,0);
cooling=Pulse('OffRes422',0,1000);

detetction= Pulse('OnRes422',1000,500,'freq',220);
photoncount= Pulse('PhotonCount',1000,500);

Seq=[init cooling detection photoncount];
% compile the program
prog=CodeGenerator;         % initialize code
prog.GenSeq(seq);           % create the code for the pulse seq
prog.GenFinish ;            % end code
prog.DisplayCode ;
% FPGA/Host control
com=Tcp2Labview('localhost',6340);% open TCP communication channel
pause(1);
com.UploadCode(prog);      % transmit the code to the host memory
com.UpdateFpga;            % order the host to update the fpga code
com.WaitForHostIdle;       % wait until host finished it last task
com.Execute(100);          %  execute the program 100 time
com.WaitForHostIdle;       % wait until host finished it last task
counts=com.ReadOut(100);   %  read 100 values
com.Delete;           % close the tcp communication channel

disp(mean(counts));
```

3.2 Condition Implementation

4 Appendix A. Pigeon Instructions Set

command	subcommand	operatoin
0	NA	Idle
1	0 (default)	param1 → AO0
	1	param1 → AO 1
	2	param1 → AO2
	3	param1 → AO3
2	0 (default)	param1 (biary)→ Cpnnector0 DIO0 [OffRes422]
	1	param1 (biary)→ Cpnnector0/DIO1 [OnRes422]
	2	param1 (biary)→ Cpnnector0/DIO2 [OnResCooling]
	3	param1 (biary)→ Cpnnector0/DIO3 [OpticalPumping]
	4	param1 (biary)→ Cpnnector0/DIO4 [Repump1092]
	5	param1 (biary)→ Cpnnector0/DIO5 [Repump1033]
	6	param1 (biary)→ Cpnnector0/DIO6 [674Switch1]
	7	param1 (biary)→ Cpnnector0/DIO7 [674Switch2]
	8	param1 (biary)→ Cpnnector0/DIO8 [RAP408]
	9	param1 (biary)→ Cpnnector2/DIO1 [DO0 Keithley Burst]
	10	param1 (biary)→ Cpnnector2/DIO2 [DO10 674 Switch3]
	11	param1 (biary)→ Cpnnector2/DIO3 [DO11 674 Switch4]
	12	param1 (biary)→ Cpnnector2/DIO4 [DO12 PI Int. Hold]
	13	param1 (biary)→ IntTrigger
3	0 (default)	Push photons counts from both PMTs together to register B.
	1	Push photons counts from PMT1 to register B.
	2	Push photons counts from PMT2 to register B.
	3	Reset photons count (inverting flag 1).
4	0 (default)	Merge paramenters 1(low) and 2(hi) into register A.
	1	Push parameter 1 into register B.
	2	Push parameter 1 into register C.
	3	Increase register C by 1.
	4	Increase register A by 1.
	5	Equal register C to register B.
5	0 (default)	If the mergent of parameters 1(low) and 2(hi) equals register A - raise flag 0 otherwise lower it.
	1	If parameter 1 is greater than register B raise flag 0 otherwise lower it.
	2	If parameter 1 equals register C raise flag 0 otherwise lower it.
	3	If register B is greater than register C raise flag 0 otherwise lower it.
	4	Push external trigger state to flag 3, on rising of the external trigger raise flag 0, otherwise lower it.
6	NA	If flag 0 is high skip the instruction indicator to the line of parameter 1, otherwise skip it to the line of parameter 2.
7	0 (default)	Push register A to FIFO.
	1	Push register B to FIFO.
	2	Push register C to FIFO.
	3	Push Photon Phase to FIFO.
8	NA	Raise all the three flags and stop the program executation.

command	subcommand	operatoin
9	0 (defualt)	Transfer the six low bits of param1 to the address bus bits. Transfer the eight low bits of param2 to the data bus bits.
	1	Transfer the four low bits of param1 to the first DDS legs by the order: 0 (LSB) - Reset, 1 - FSK, 2 - WRB, 3 - Ioud.
	2	Transfer the four low bits of param1 to the second DDS legs by the order: 0 (LSB) - Reset, 1 - FSK, 2 - WRB, 3 - Ioud.

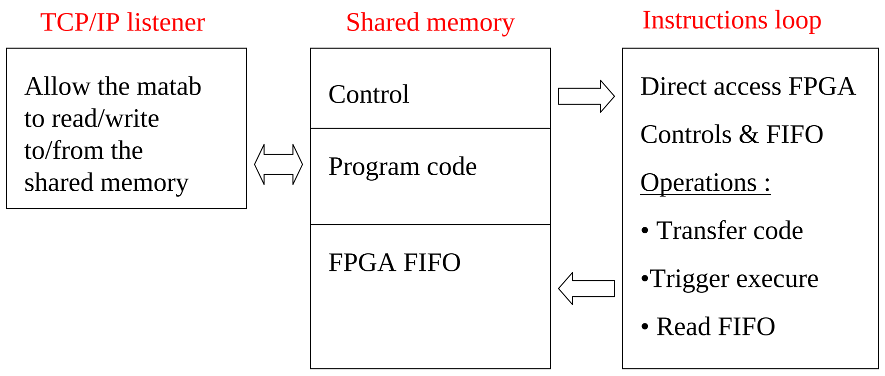


Figure 4: Schematic chart of the the Labview program hierarchy as implemented of the host computer and the FPGA card.

FPGA - Processor structure

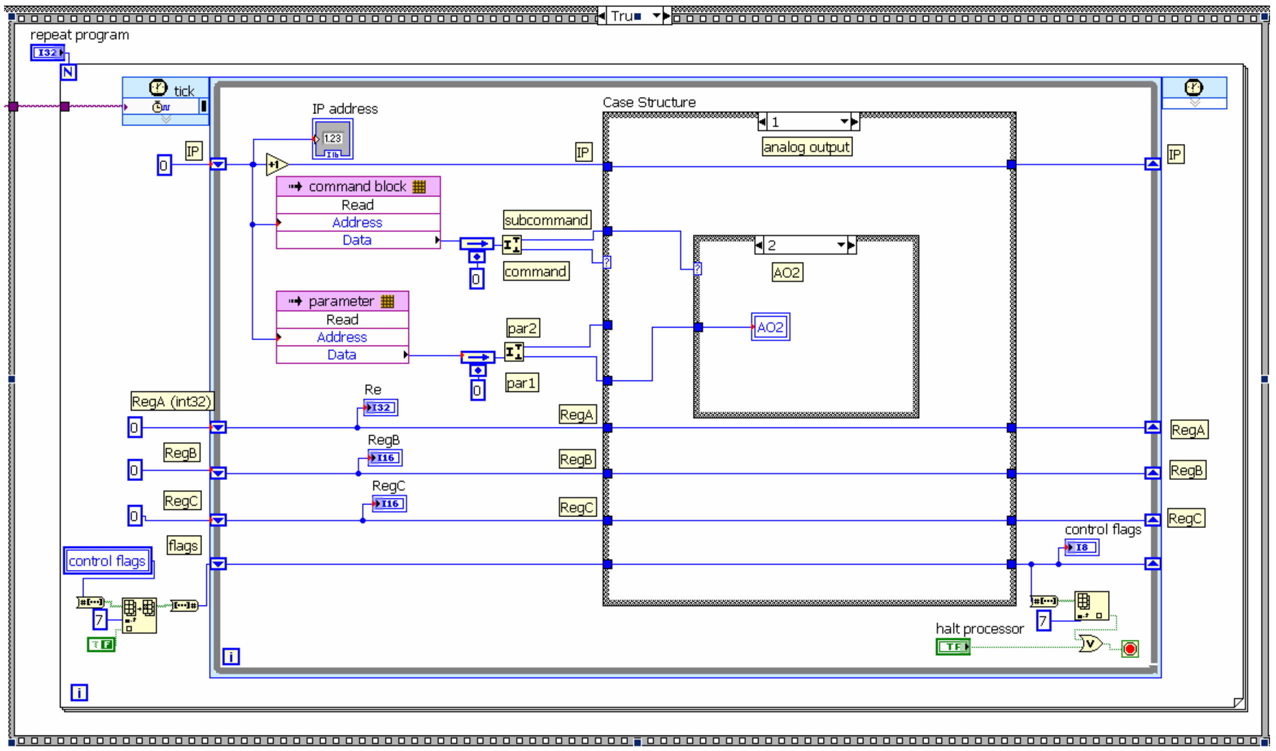


Figure 5: An Image of the Pigeon core as implemented by Labview FPGA. Each iteration of the times loop executes a line of the program code. The two case statements inside route the code by the command the subcommand values of each instruction. One can spot the three registers and the flags array of the core.

5 Appendix B. DDS Operation

5.1 Introduction

A Direct Digital Synthesiser (DDS) is a chip which generates arbitrary frequency and phase signals. It's frequency accuracy and stability relays over the external clock which drives it. We use Analog Devices AD9854 DDS which is embedded inside a NovaTech dds8m evaluation board through which the DDS is interfaced.

The DDS evaluation board allows sweeping the frequency of the DDS from 0 to 100 MHz (limited by a low pass filter at the output). It has its own local oscillator but can also operated under external reference clock. Commands are transferred to the DDS by two ways:

1. By serial (RS-232) communication protocol, which is too slow for our purposes (frequency word updating time of ~ 1 msec).

2. By parallel communication port which employees a bus of 18 lines.

The DDS control is done by addressing and configuring its internal registers. This DDS has 40 internal registers that determine the DDS signal frequency, phase, amplitude, mode of operation and other features.

A DDS operation sequence can be split to the following steps:

1. **Initialization and Configuration.** This part is done before the execution of the experiment. The DDS is initialized by a reset pulse and configured for operation is the desired mode. The DDS offers several modes of operations that allow quick modification of the DDS output frequency or phase.

2. **Switching and Updating.** Changing of the DDS output along the experiment either by switching between two different working points (frequency/phase) or by updating internal registers values. The exact intervention depends upon the mode for which the DDS was configured to work in.

The parallel communication bus is made of address and data buses (that are common for all the DDS) and controlling bits (latches) that are exclusive for each DDS. This bus is driven by the FPGA connection box (SCB-68), which is dedicated to the DDS operation. Though this connection box allows manipulation of 40 lines of a bus, not all these connections are used, currently. Among the lines that are active:

1. 6 lines are dedicated for the address bus, common for all the DDSs.

2. 8 lines are dedicated for the data bus, common for all the DDSs.

3. 1 line is dedicated for the Reset bit, common for all the DDSs.

4. 4 additional lines per DDDS are required to to direct the information flow from the common bus to a specific DDS and switch it. These four lines drive the FSK, WRB and Ioud latches of the DDS. Additional line commands the RF switch which is at the output of each DDS.

A visualization of the lines allocation can be seen in fig.6.

5.2 The DDS API

The Matlab Application Program Interface (API), which supports the DDS operation, translates the user commands to Pigeon instructions (under command 9) that program the DDS. The API commands are manifested by methods of the CodeGenerator class (under "GenDDS..." prefix), they are divided to two sets of commands:

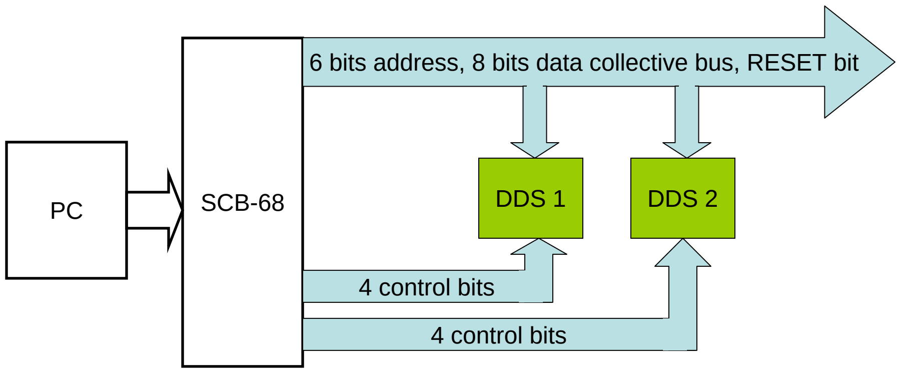


Figure 6: The lines allocation of the FPGA connection box.

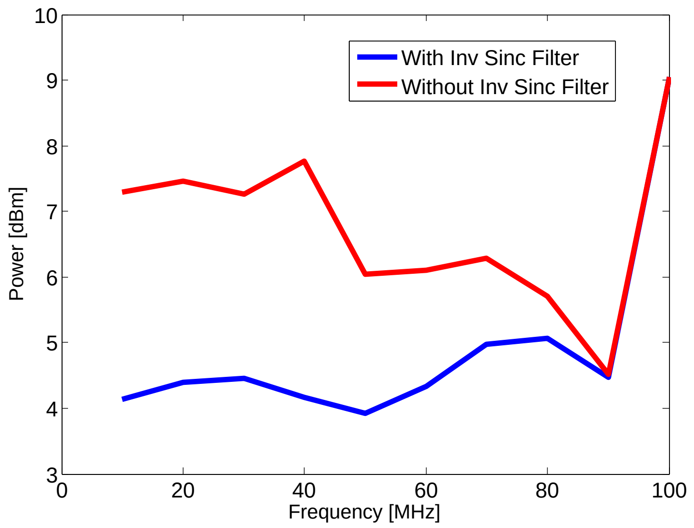


Figure 7: The output power of a DDS with and without inverse sinc filter.

1. **Low Commands.** Small group of commands that preform the basic transmission between the Pigeon FPGA and the DDS. These commands include setting the values of the address and the data buses, signaling the DDS by its latches.

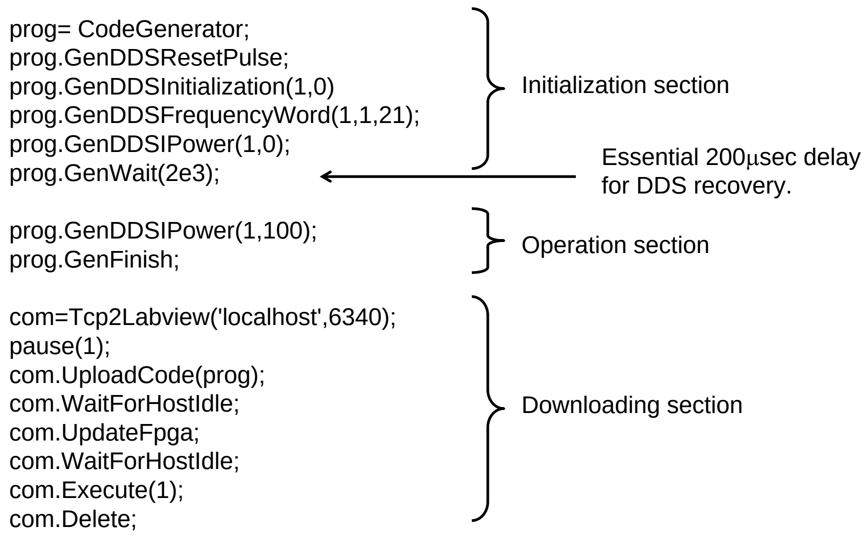
2. **High Commands.** A set of commands that encapsulate the low level commands for a specific purpose like initialization, frequency setting, etc.

Generally speaking, the interface to the DDS should be done by the high commands only.

Code that run the DDS has to follow the basic model which is:

1. **Initialization and definitions.** This stage reset the DDSs (GenDDSResetPulse), defines each DDS mode of operation (GenDDSInitialization) and the relevant frequencies / phases / amplitudse(GenDDSFrequencyWord, GenDDSPhaseWord, GenDDSIPower respectively). When working in the ramped FSK or chirp modes consideration has to be given to the frequency shift parameters (GenDDSSweepParameters).
2. **Operation.** This stage is buffered from the initialization by 200μsec of dwelling to allow the DDS to configure itself. In this stage the DDS works and accept signals while it works. These signals can be change of the FSK pin logical value (GenDDSFSKState), and changing of the DDS output frequency / phase / amplitude.

Figure 8 demonstrates the structure of a DDS running code.



```
prog= CodeGenerator;
prog.GenDDSResetPulse;
prog.GenDDSInitialization(1,0)
prog.GenDDSFrequencyWord(1,1,21);
prog.GenDDSIPower(1,0);
prog.GenWait(2e3);

prog.GenDDSIPower(1,100);
prog.GenFinish;

com=Tcp2Labview('localhost',6340);
pause(1);
com.UploadCode(prog);
com.WaitForHostIdle;
com.UpdateFpga;
com.WaitForHostIdle;
com.Execute(1);
com.Delete;
```

Figure 8: The initialization and operation sections of a DDS code.

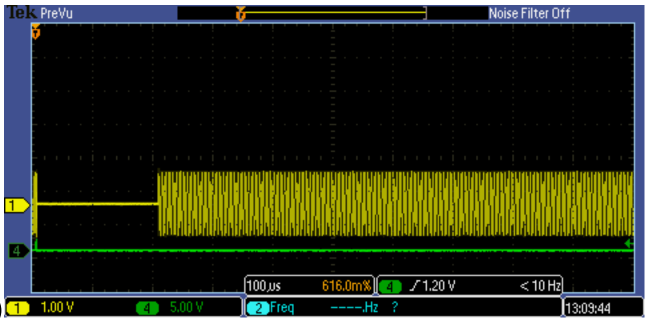
5.3 Modes of operation and code examples

The following sections demonstrate the DDS operation modes.

5.3.1 Single Tone (0)

In single tone mode of operation the DDS output is a single sine wave with constant frequency. The following code implement single tone mode of operation.

```
instrreset

prog= CodeGenerator; prog.GenDDSResetPulse;
prog.GenDDSInitialization(1,0);
prog.GenDDSFrequencyWord(1,1,21); prog.GenDDSIPower(1,0);
prog.GenWait(2e3); prog.GenDDSIPower(1,100);
prog.GenFinish;

com=Tcp2Labview('localhost',6340); pause(1);
com.UploadCode(prog);
com.WaitForHostIdle; % wait until host finished it last task
com.UpdateFpga;
com.WaitForHostIdle; % wait until host finished it last task
com.Execute(1); com.Delete;
```

a) 

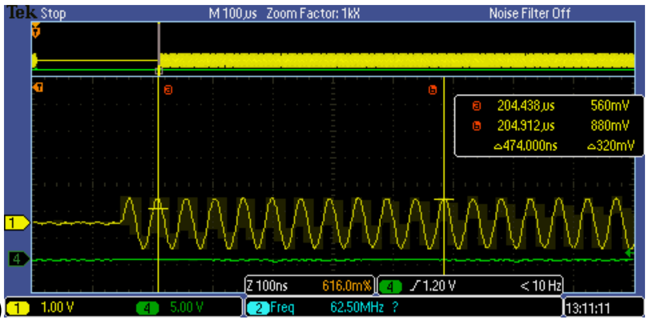
b) 

Figure 9: a) General plot of the DDS output (yellow), triggered by the reset pin voltage (green). b) Close-up on the sine wave shows an output frequency of 21 MHz.

Figure 9 shows the output, in frequency of 21 MHz, which results by the code. Note that a 2000 clock cycles delay buffers between the declaration and the operation parts of the code. During this dwell the DDS output is shut down to avoid incoherent output.

The next code example demonstrates change of the DDS frequency while working in a single tone mode of operation. The frequency change is $\sim 100\mu sec$) after the sine wave stars.

```
instrreset

prog= CodeGenerator; prog.GenDDSResetPulse;
prog.GenDDSInitialization(1,0);
prog.GenDDSFrequencyWord(1,1,7); prog.GenDDSIPower(1,0);
prog.GenWait(2e3); prog.GenDDSIPower(1,100);
prog.GenWait(1e3); prog.GenDDSFrequencyWord(1,1,20);
prog.GenWait(1e3); prog.GenDDSFrequencyWord(1,1,30);
prog.GenFinish;


com=Tcp2Labview('localhost',6340); pause(1);
com.UploadCode(prog);
com.WaitForHostIdle; % wait until host finished it last task
com.UpdateFpga;
com.WaitForHostIdle; % wait until host finished it last task
com.Execute(1); com.Delete;
```

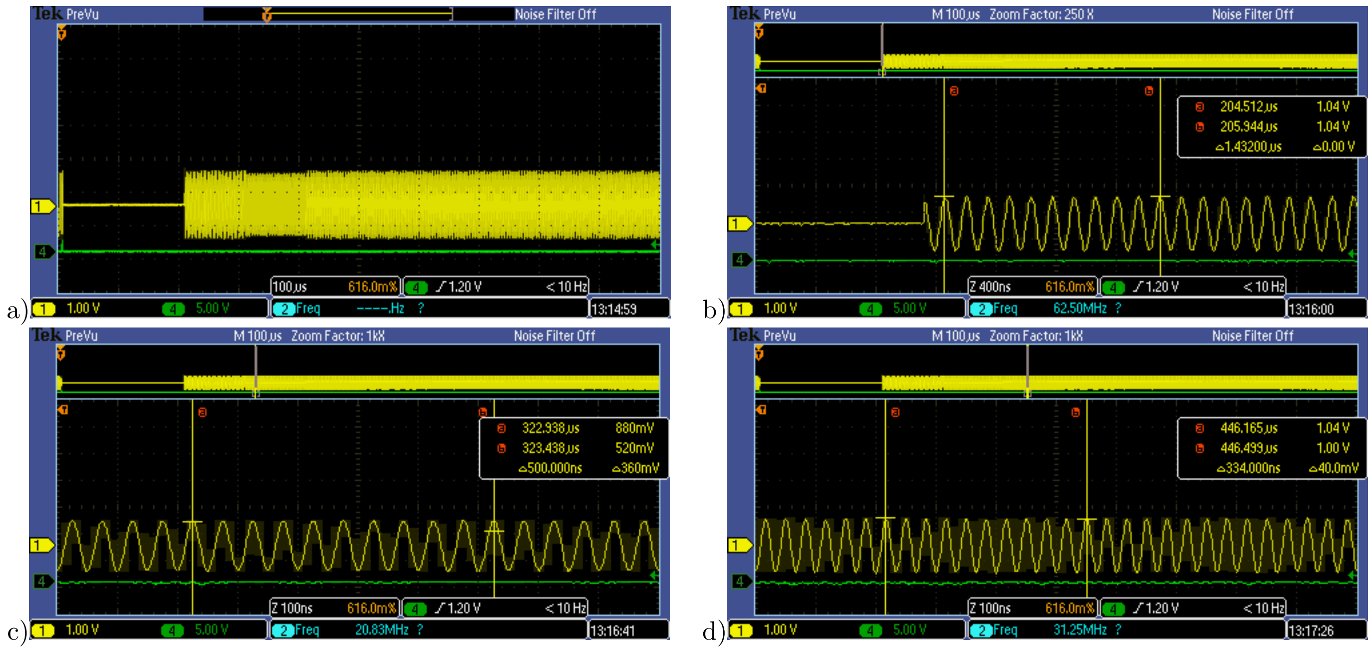


Figure 10: a) General plot of the DDS output (yellow), triggered by the reset pin voltage (green). b) The initial frequency is 7 MHz. c) Middle frequency of 20 MHz. d) Final frequency of 30 MHz.

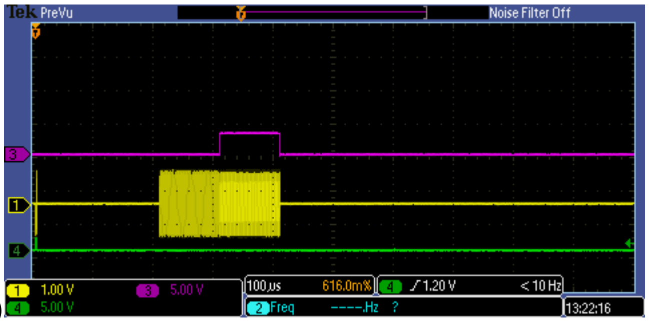
Figure 10 shows skipping of the DDS frequency between 7, 20, 30 MHz by updating the frequency word value. In this example, the minimal DDS dwell time in a single frequency takes ∼600 nsec (no faster frequency changes can be done).

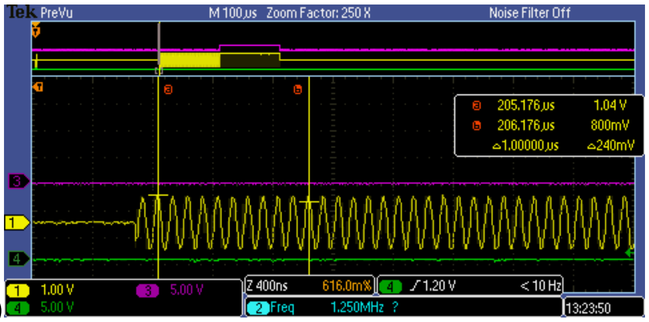
5.3.2 Frequency Shift Keying (FSK) (1)

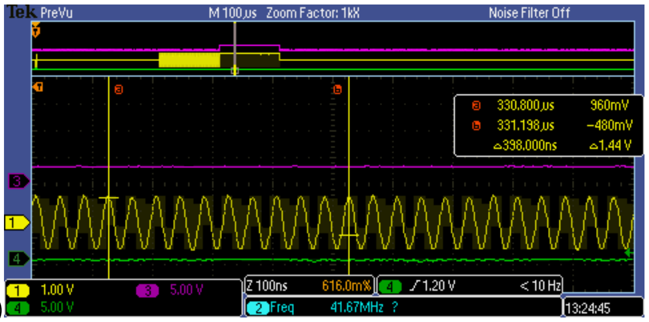
The FSK mode allows fast skipping of the DDS frequency between two predefined frequencies, just by toggling the FSK pin. The following code demonstrate frequency hopping between 10 to 25 MHz in FSK mode.

```
instrreset
clear prog

prog= CodeGenerator;

prog.GenDDSResetPulse; prog.GenDDSInitialization(1,1);
prog.GenDDSFrequencyWord(1,1,10);
prog.GenDDSFrequencyWord(1,2,25); prog.GenDDSIPower(1,0);
prog.GenWait(2e3);

prog.GenDDSIPower(1,100); prog.GenWait(1e3);
prog.GenDDSFSKState(1,1); prog.GenWait(1e3);
prog.GenDDSFSKState(1,0); prog.GenDDSIPower(1,0);
prog.GenFinish;


com=Tcp2Labview('localhost',6340); pause(1);
com.UploadCode(prog);
com.WaitForHostIdle; % wait until host finished it last task
com.UpdateFpga;
com.WaitForHostIdle; % wait until host finished it last task
com.Execute(1); com.Delete;
```

In Fig. 11 a skipping between two frequencies can be seen. Note that a delay of 350 nsec occurs between the raising of the FSK pin and the change of the DDS output frequency.

a)


b)


c)


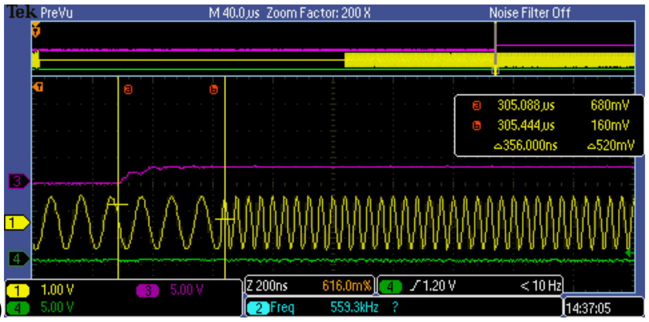
d)


Figure 11: a) General plot of the DDS output (yellow), triggered by the reset pin voltage (green). The FSK pin value can be seen in purple. b) Initial frequency of 10 MHz. c) Skipped frequency of 25 MHz. d) A delay of 350 nsec between the rising of the FSK pin voltage and the frequency updating.

5.3.3 Ramped FSK (2)

In ramped FSK mode of operation the DDS frequency is shifted between two frequencies gradually in predefined steps. The following code demonstrates shifting of the DDS frequency between 5 to 15 MHz in steps of 1 MHz, each steps lasts 1 μsec. When the FSK pin is in a logical high value the DDS seeks toward the frequency value of frequency word 2, otherwise the FSK seeks toward the frequency value of frequency word 1. All the DDS frequency shifts are subjected to the definition of the sweep parameters, which means that the frequency change rate remains constant no matter what. Therefore, if the FSK pin value won't remain fixed for time long enough, the DDS won't reach its destination frequency.

```
instrreset

prog = CodeGenerator; prog.GenDDSResetPulse;
prog.GenDDSInitialization(1,2);
prog.GenDDSFrequencyWord(1,1,5);
prog.GenDDSFrequencyWord(1,2,15);
prog.GenDDSSweepParameters (1,1,1); prog.GenDDSIPower(1,0);
prog.GenWait(2e3);

prog.GenDDSIPower(1,100); prog.GenDDSFSKState(1,1);
prog.GenWait(1e2); prog.GenDDSFSKState(1,0);
prog.GenWait(1e2); prog.GenDDSFSKState(1,1);
prog.GenWait(1e5); prog.GenDDSFSKState(1,0);
prog.GenFinish;

com=Tcp2Labview('localhost',6340); pause(1);
com.UploadCode(prog);
com.WaitForHostIdle; % wait until host finished it last task
com.UpdateFpga;
com.WaitForHostIdle; % wait until host finished it last task
com.Execute(1); com.Delete;
```

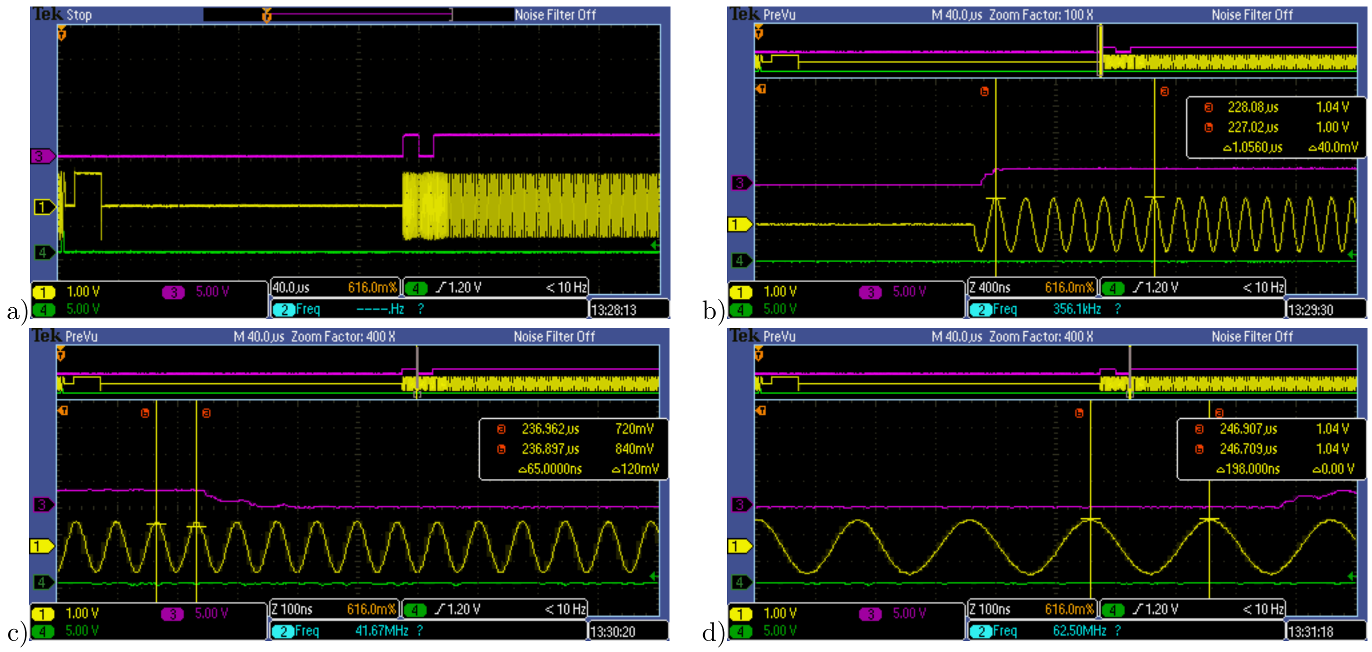


Figure 12: a) General plot of the DDS output (yellow), triggered by the reset pin voltage (green). The FSK pin value can be seen in purple. b) Initial frequency of 6 MHz. c) Middle frequency of 15 MHz. d) Final frequency of 5 MHz.

Note that, as can be seen in Fig.12, the output initial frequency is not 5 MHz as programmed but 6 MHz. That's because in ramped FSK mode the DDS adds one frequency step (1 MHz in our case) to the initial output frequency.

5.3.4 Chirp (3)

In chirp mode the DDS output frequency increase / decrease monotonously given an initial frequency and frequency swift parameters. Here the FSK pin is used to freeze the frequency scan (logical value 1) and to melt it (logical value 0). The following code demonstrates operation is a chirp mode. The DDS frequency shift parameters are set be the GenDDSSweepParameters as in ramped FSK mode.

```
clear prog com instrreset

prog = CodeGenerator; prog.GenDDSResetPulse;

prog.GenDDSInitialization(1,3); prog.GenDDSFSKState(1,1);
prog.GenDDSFrequencyWord(1,1,15);
prog.GenDDSSweepParameters (1,0.5,1);
prog.GenDDSIPower(1,0); prog.GenWait(2e3);

prog.GenDDSIPower(1,100); prog.GenWait(5e1);
prog.GenDDSFSKState(1,0); prog.GenWait(5e2);
prog.GenDDSFSKState(1,1); prog.GenWait(5e2);
prog.GenDDSFSKState(1,0);

prog.GenFinish;


com=Tcp2Labview('localhost',6340); pause(1);
com.UploadCode(prog);
com.WaitForHostIdle; % wait until host finished it last task
com.UpdateFpga;
com.WaitForHostIdle; % wait until host finished it last task
com.Execute(1); com.Delete;
```

In this example the DDS output frequency rises by 0.5 MHz every 1 μsec. Hence, after 50 μsec delay (in which the FSK logical value was 0) it increases by 25 MHz to 40.5 MHz. Again, the initial frequency includes one increase step.

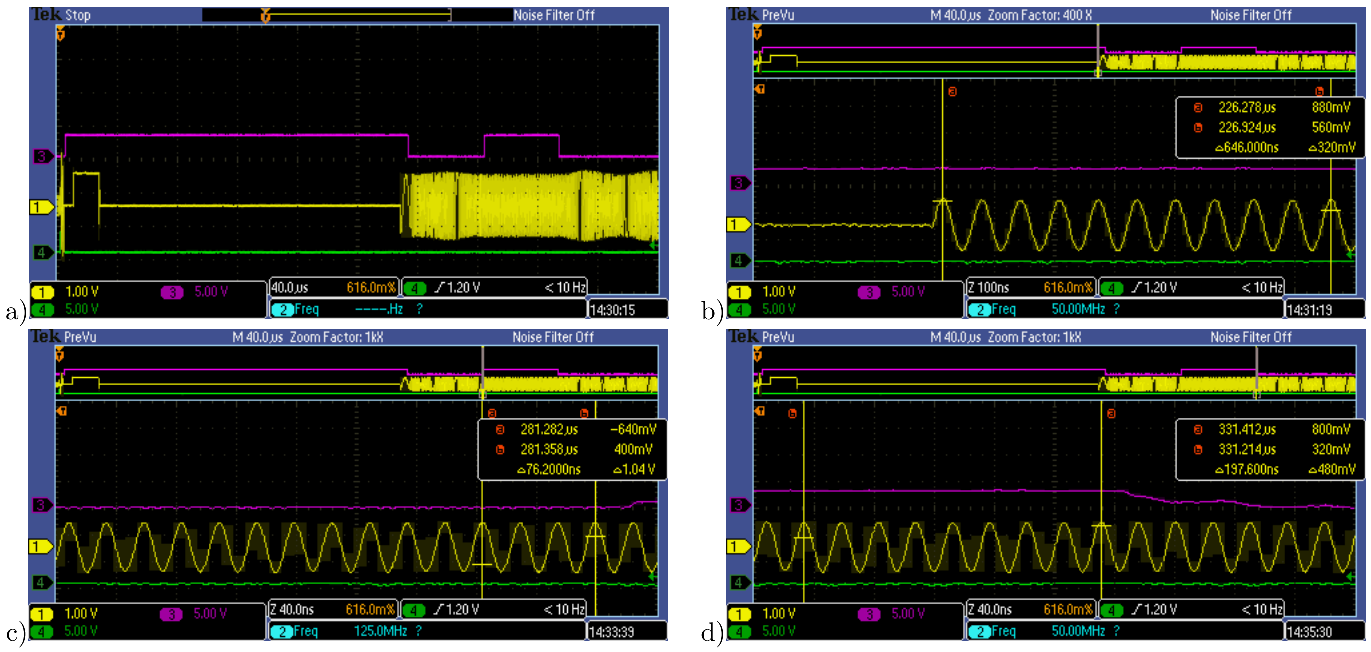


Figure 13: a) General plot of the DDS output (yellow), triggered by the reset pin voltage (green). The FSK pin value can be seen in purple. b) Initial frequency of 15.5 MHz. c) Middle frequency of 40.5 MHz. d) The frozen frequency remains 40.5 MHz.

5.3.5 Binary Phase Shift Keying (BPSK) (4)

In BSK mode the DDS output signal skips between two predefined phases, at the same frequency, switched by the FSK pin.

```
instrreset

prog= CodeGenerator; prog.GenDDSResetPulse;
prog.GenDDSInitialization(1,4);
prog.GenDDSFrequencyWord(1,1,1);
prog.GenDDSPhaseWord(1,1,0); prog.GenDDSPhaseWord(1,2,pi);
prog.GenDDSIPower(1,0); prog.GenWait(2e3);

prog.GenDDSIPower(1,100); prog.GenWait(50);
prog.GenDDSFSKState(1,1); prog.GenWait(50);
prog.GenDDSFSKState(1,0); prog.GenFinish;


com=Tcp2Labview('localhost',6340); pause(1);
com.UploadCode(prog);
com.WaitForHostIdle; % wait until host finished it last task
com.UpdateFpga;
com.WaitForHostIdle; % wait until host finished it last task
com.Execute(1); com.Delete;
```

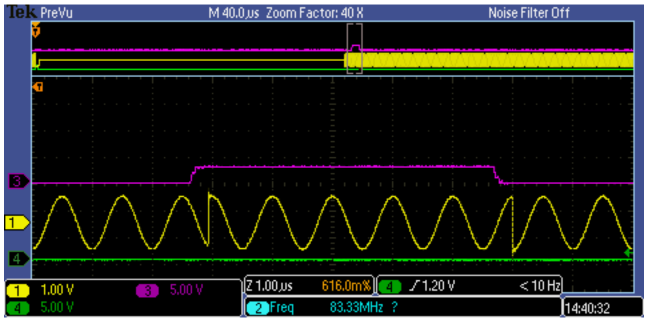


Figure 14: General plot of the DDS output (yellow), triggered by the reset pin voltage (green). The FSK pin value can be seen in purple. The output sine wave phase breaks by *pi* radians with the toggling of the FSK logical value.

Figure 14 shows a phase shift of a 1 MHz sine wave by a π. Note that the jump occurs ~250 nsec after the change in the FSK pin state.

5.3.6 Simultaneous two DDSs operation

Here we demonstrate running to DDSs simultaneously: one in a ramped FSK mode while the other is in single tone mode.

```
instrreset clear prog

prog = CodeGenerator;

prog.GenDDSResetPulse;

prog.GenDDSInitialization(1,2);
prog.GenDDSFrequencyWord(1,1,4);
prog.GenDDSFrequencyWord(1,2,50);
prog.GenDDSSweepParameters (1,0.5,1);
prog.GenDDSIPower(1,0); prog.GenDDSInitialization(2,0);
prog.GenDDSFrequencyWord(2,1,18); prog.GenDDSIPower(2,0);
prog.GenWait(2e3);

prog.GenDDSIPower(2,100); prog.GenDDSIPower(1,100);
prog.GenDDSFSKState(1,1); prog.GenWait(3e2);
prog.GenDDSFSKState(1,0); prog.GenDDSIPower(2,0);
prog.GenDDSFrequencyWord(2,1,25); prog.GenDDSIPower(2,100);
prog.GenWait(3e2); prog.GenDDSIPower(2,0);
prog.GenDDSIPower(1,0); prog.GenFinish;

com=Tcp2Labview('localhost',6340); pause(1);
com.UploadCode(prog);
com.WaitForHostIdle; % wait until host finished it last task
com.UpdateFpga;
com.WaitForHostIdle; % wait until host finished it last task
com.Execute(1); com.Delete;
```

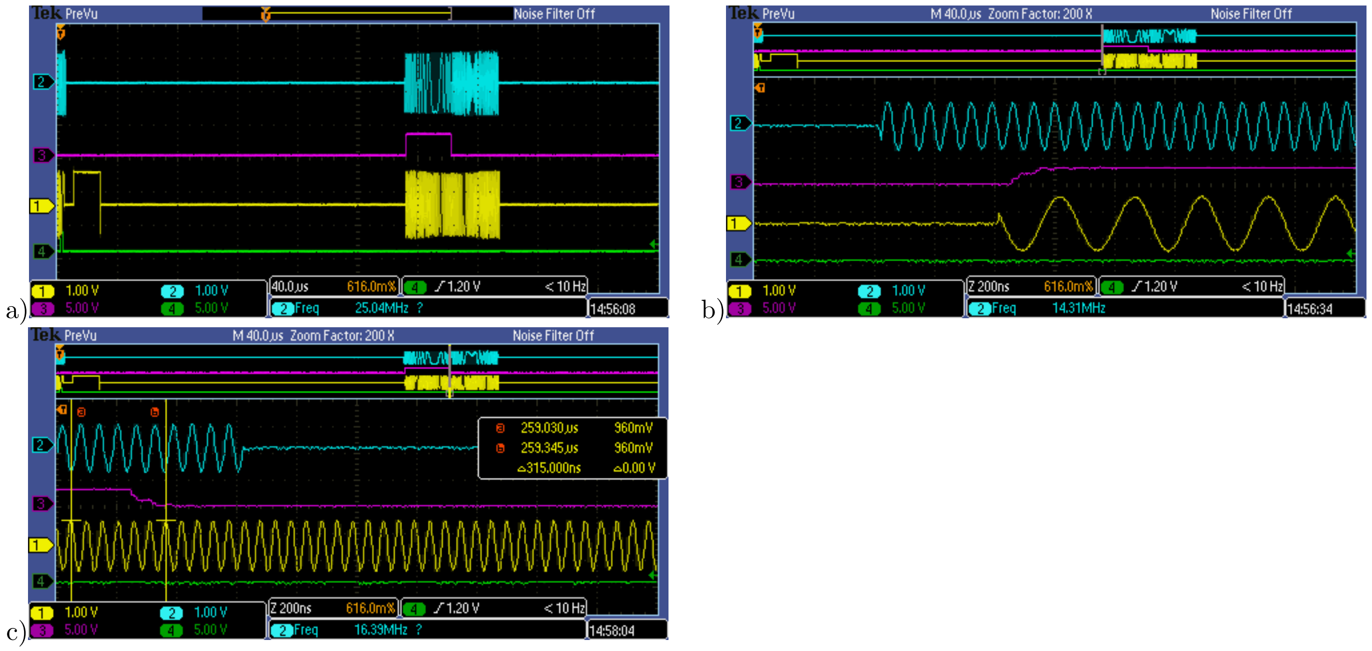


Figure 15: General plot of the two DDSs outputs. DDS1 (yellow), DDS2 (blue), DDS1 reset (green), DDS1 FSK (purple). b) DDS1 initial frequency of 4 MHZ, DDS2 frequency of 18 MHz. c) DDS1 frequency of 9 MHz, DDS2 frequency of 25 MHz.