**ALICE Universal hardware interface developer’s guide**

**Background:**

The main Universal ALICE GUI is written in Python and depends only on the more or less generic standardly available Python plugin libraries (such as Tcl/Tk, numpy, pyplot etc.). The hardware specific interface level would generally also be implemented in Python plus any plugin libraries (such as pyserial, pyaudio etc.) with any other extensions depending on the target hardware.

The universal version of ALICE itself is meant to be hardware agnostic and comes basically in two parts. The hardware agnostic GUI and a much smaller hardware specific part that just handles communicating with target hardware. Developers of hardware specific interface should generally not need to change anything within the main ALICE GUI file.

Three main functions are needed in any interface. The first function is used to connect to the device generally that is to establish the USB serial communications link. The Second function is used to capture arrays of input data samples from the device (ADC channels). The third is used to send output data arrays to the device (DAC channels). Right now this data exchange is not streaming in either direction so sample buffers on the hardware are used.

Various miscellaneous functions are often also needed depending on the functionality of the target hardware. One example might be a function that sets or changes the ADC sample rate based on the oscilloscope horizontal time base.

**Pointing to the hardware file:**

There are a number of variables that the user can use to customize the user interface. A file named alice\_init.ini is used to configure these variables. It should be placed in the same directory with the Alice-universal-alpha.pyw source file or the directory where the program is started. To point the ALICE GUI at the required hardware file the following line needs to appear in the alice\_init.ini file:

HardwareFile = "<Your-Interface-file-name-goes-here>.py" # Name of hardware specific file to load

The alice\_init.ini file is read, if found, when ALICE starts and before any of the windows are created. If no init file is found the internal default settings are used and the program will prompt the user for the name of the interface file to load.

**Interface Code Sections:**

**Check to see if the right libraries are installed:**

It is generally a good idea to check to make sure any required Python support libraries are installed. In this example a check is done while importing the pyserial library which is needed in the case of this hardware example.

#

try:

import serial

import serial.tools.list\_ports

except:

root.update()

showwarning("WARNING","Serial Library not installed?!")

root.destroy()

exit()

#

**Defining variables:**

The following system wide variables need to be set based on the capabilities of the hardware.

# adjust for your specific hardware by changing these values

CHANNELS = 3 # Number of supported Analog input channels

AWGChannels = 2 # Number of supported Analog output channels

PWMChannels = 1 # Number of supported PWM output channels

DigChannels = 8 # Number of supported Dig channels

LogicChannels = 6 # Number of supported Logic Analyzer channels

EnablePGAGain = 0 # Does the hardware support input channel programmable gain?

EnableAWGNoise = 0 # Does the hardware support built in noise generation?

AllowFlashFirmware = 1 # Allow the user to possibly flash new firmware to device?

Tdiv.set(10) # Set the number of horizontal time divisions on the Scope grid

AWGPeakToPeak = 3.29 # This variable is used to set or calibrate the peak AWG (DAC) output voltage

DevID = "Pico MCP 3" # Hardware specific device ID name text goes here

SerComPort = 'Auto' # Tells the ConnectDevice() function to try and automatically find target serial port

TimeSpan = 0.01 # Define and set the starting Horizontal time span variable

ADC\_Cal = 3.29 # This variable is used to set or calibrate the peak analog (ADC) input voltage

AWGRes = 4095 # Define the AWG DAC resolution, 255 For 8 bits, 4095 for 12 bits, 1023 for 10 bits

InterpRate = 4 # Set the interpolation rate of the input samples

EnableInterpFilter.set(1) # variable used to turn on and off the interpolation

MaxSampleRate = SAMPLErate = 12500\*InterpRate # define and set place holder for ADC sample rate

AWGSampleRate = 50000 # define and set place holder for DAC sample rate

# variables used to define ADC LSB size based on ADC resolution

LSBsizeA = LSBsizeB = LSBsizeC = LSBsizeD = LSBsize = ADC\_Cal/4096.0

PhaseOffset = 12.5 # dummy variable to correct any phase offset between scope channels

MinSamples = 1024 # Set number of input samples per capture

AWGBuffLen = 2048 # Set number of AWG output samples

SMPfft = MinSamples\*InterpRate # Set FFT size based on above fixed acquisition record length

# Pre define these variables

Cycles = 1

VBuffA = numpy.ones(MinSamples\*InterpRate) # holds Scope channel A input samples

VBuffB = numpy.ones(MinSamples\*InterpRate) # holds Scope channel B input samples

VBuffC = numpy.ones(MinSamples\*InterpRate) # holds Scope channel C input samples

VBuffD = numpy.ones(MinSamples\*InterpRate) # holds Scope channel D input samples

VBuffG = numpy.ones(MinSamples\*InterpRate) # holds Digital input channel samples

MBuff = numpy.ones(MinSamples\*InterpRate) # holds Math samples

MBuffX = numpy.ones(MinSamples\*InterpRate) # holds Math X samples

MBuffY = numpy.ones(MinSamples\*InterpRate) # holds Math Y samples

VmemoryA = numpy.ones(MinSamples\*InterpRate) # The memory for Ch A averaging

VmemoryB = numpy.ones(MinSamples\*InterpRate) # The memory for Ch B averaging

VmemoryC = numpy.ones(MinSamples\*InterpRate) # The memory for Ch C averaging

VmemoryD = numpy.ones(MinSamples\*InterpRate) # The memory for Ch D averaging

#

**Function to close / exit ALICE**

This function is used to close down and or disconnect the hardware connection and exit the main GUI program. It is included in the interface file because it may contain hardware specific commands.

Example:

# Hardware specific function to close and exit ALICE

def Bcloseexit():

global RUNstatus, Closed, ser

RUNstatus.set(0)

Closed = 1

#

try:

# Turn off AWG here

# Turn off PWM here

# Try to write last config file, Don't crash if running in Write protected space

BSaveConfig("alice-last-config.cfg")

# May need to be changed for specific hardware port

ser.close()

# exit

except:

donothing()

root.destroy()

exit()

#

**Function to change or set sample rate:**

Send whatever commands to the hardware are needed to set the input sample rate perhaps based on horizontal time base.

def SetSampleRate():

global TimeSpan, SHOWsamples, InterpRate, Tdiv

global MaxSampleRate, SAMPLErate, TimeDiv, ser

try:

TimeDiv = UnitConvert(TMsb.get())

except:

pass

# various if statements here to set sample rate

if TimeDiv < 0.000099:

…..

#

**Function to get input data samples:**

This function requests however many input sample buffers from the hardware and places the data as floating point voltages into one or the other VBuffA, VBuffB, VBuffC, VbuffD, or VBuffG numpy arrays. Best for developers to familiarize themselves with an existing example hardware interface before attempting to write this section.

#

# Main function to request and receive a set of ADC samples

#

def Get\_Data():

global VBuffA, VBuffB, VBuffC, VBuffD, VBuffG

global ShowC1\_V, ShowC2\_V, ShowC3\_V, ShowC4\_V

# These variables determine which scope channels are being displayed

global LSBsizeA, LSBsizeB, LSBsizeC, LSBsizeD

global MaxSampleRate, SAMPLErate, EnableInterpFilter

global ser, SHOWsamples, TRIGGERsample, TgInput, TimeSpan

global TrigSource, TriggerEdge, TriggerInt, Is\_Triggered

global vct\_btn, vdt\_btn, HoldOff, MinSamples, Interp4Filter

global D0\_is\_on, D1\_is\_on, D2\_is\_on, D3\_is\_on

global D4\_is\_on, D5\_is\_on, D6\_is\_on, D7\_is\_on

global DBuff0, DBuff1, DBuff2, DBuff3, DBuff4, DBuff5, DBuff6, DBuff7

global D0line, D1line, D2line, D3line, D4line, D5line, D6line, D7line

# Return with filled sample buffers

**Function to connect to hardware device:**

This function is required to establish the communications connection to the device. In this example the serial ports are polled to find the one that matches the proper vendor and product USB IDs.

#

## Try to connect to board

#

def ConnectDevice():

global SerComPort, DevID, MaxSamples, SAMPLErate, MinSamples, AWGSampleRate

global bcon, FWRevOne, HWRevOne, MaxSampleRate, ser, SHOWsamples

global CH1Probe, CH2Probe, CH1VRange, CH2VRange, TimeDiv

global CHAsb, CHBsb, TMsb, LSBsizeA, LSBsizeB, ADC\_Cal, LSBsize

global d0btn, d1btn, d2btn, d3btn, d4btn, d5btn, d6btn, d7btn

# print("SerComPort: ", SerComPort)

if DevID == "No Device" or DevID == "Pico MCP 3": # This needs to match value set above

#

if SerComPort == 'Auto':

ports = serial.tools.list\_ports.comports()

for port in ports: # ports:

# looking for this ID: USB\VID\_2E8A&PID\_000A

if "VID:PID=2E8A:000A" in port[2]:

print("Found: ", port[0])

SerComPort = port[0]

# Setup instrument connection

print("Trying to open ", SerComPort)

ser = serial.Serial(SerComPort) # open serial port

if ser is None:

print('Device not found!')

Bcloseexit()

#exit()

#

ser.baudrate = 2000000 # Dummy number USB runs at max supported speed

# Now the rest of the commands needed to set up or configure the hardware as needed

#

return(True) # return a logical true if successful!

else:

return(False)

#

**AWG control functions:**

#

# AWG Stuff

#

# Function to download waveform samples to hardware

#

def AWGASendWave(AWG3):

global ser, AWGARecLength, AWGBuffLen, AWGRes

global AWGAAmplvalue, AWGAOffsetvalue, AWGPeakToPeak

# Expect array values normalized from -1 to 1

# Scale binary DAC values to send based on value of AWGRes

AWG3 = numpy.array(AWG3) \* 0.5 # scale by 1/2

# Get Low and High voltage levels

MinCode = int((AWGAAmplvalue / AWGPeakToPeak) \* AWGRes)

if MinCode < 0:

MinCode = 0

if MinCode > AWGRes:

MinCode = AWGRes

MaxCode = int((AWGAOffsetvalue / AWGPeakToPeak) \* AWGRes)

if MaxCode < 0:

MaxCode = 0

if MaxCode > AWGRes:

MaxCode = AWGRes

# print("MaxCode = ", MaxCode, "MinCode = ", MinCode)

# Scale to high and low voltage values

Gain = MaxCode - MinCode

Offset = int((MaxCode + MinCode)/2)

AWG3 = (AWG3 \* Gain) + Offset

n = 0

AWG1 = []

while n < len(AWG3):

AWG1.append(int(AWG3[n]))

n = n + 1

AWG1 = numpy.array(AWG1)

#

AWGARecLength = len(AWG1)

if AWGARecLength > AWGBuffLen:

AWGARecLength = AWGBuffLen

if len(AWG1) < AWGBuffLen:

# Send AWG Buffer Length command

SendStr = 'N' + str(len(AWG1)) + '\n'

#

SendByt = SendStr.encode('utf-8')

ser.write(SendByt)

else:

SendStr = 'N' + str(AWGBuffLen) + '\n'

#

SendByt = SendStr.encode('utf-8')

ser.write(SendByt)

#

index = 0

while index < AWGARecLength:

data = AWG1[index]

# Send buffer index and waveform sample data

SendStr = 'L' + str(index) + 'D' + str(data) + '\n'

# print(SendStr)

SendByt = SendStr.encode('utf-8')

ser.write(SendByt)

index = index + 1

#

Similar send function for AWG channel B as needed goes here.

# Function used to toggle on / off AWG A output

def SetAwgA\_Ampl(Ampl): #

global ser, AwgBOnOffBt, AwgaOnOffLb, AwgbOnOffLb

global AWGSampleRate

if Ampl == 0:

ser.write(b'Gx\n') # send off command here

else:

ser.write(b'Go\n') # send on command here

# Function used to toggle on / off AWG B output

def SetAwgB\_Ampl(Ampl): #

global ser, AwgBOnOffBt, AwgAOnOffBt, AwgaOnOffLb, AwgbOnOffLb

global AWGSampleRate

if Ampl == 0:

ser.write(b'gx\n') # send off command here

else:

ser.write(b'go\n') # send on command here

#

The following function is needed more or less as is based on the number and types of AWG waveforms supported. This example used the waveform generating functions built in the main ALICE GUI code to make the waveforms. The built in generators call the AWGASendWave(AWG3) and AWGBSendWave(AWG3) functions. Up to 16 waveform shapes are supported in the GUI.

For hardware with built in AWG waveforms, just need to send the appropriate command for each wave shape.

#

## Make the current selected AWG waveform

# Name strings for each numbered wave shape

##AwgString1 = "Sine"

##AwgString2 = "Triangle"

##AwgString3 = "Ramp Up"

##AwgString4 = "Ramp Down"

##AwgString5 = "Stair Up"

##AwgString6 = "Stair Down"

##AwgString7 = "Stair Up-Down"

AwgString9 = "Cosine"

AwgString10 = "Full Wave Sine"

AwgString11 = "Half Wave Sine"

AwgString12 = "Fourier Series"

AwgString13 = "Schroeder Chirp"

AwgString14 = "Uniform Noise"

AwgString15 = "Gaussian Noise"

#

## Make or update the current selected AWG waveform

#

def MakeAWGwaves(): # re make awg waveforms in case something changed

global AWGAShape, AWGAShapeLabel, AWGBShape, AWGBShapeLabel

global AWGAAmplEntry, AWGAOffsetEntry, AWGAFreqEntry, AWGASymmetryEntry

global AWGADutyCycleEntry

global AWGAAmplvalue, AWGBOffsetvalue, AWGBAmplvalue, AWGBOffsetvalue, AWGAFreqvalue

global AWGBAmplEntry, AWGBOffsetEntry, AWGBFreqEntry, AWGBSymmetryEntry

global AWGBDutyCycleEntry

global FSweepMode, MaxSampleRate, BisCompA

global AwgString1, AwgString2, AwgString3, AwgString4, AwgString5, AwgString6

global AwgString7, AwgString8, AwgString9, AwgString10, AwgString11, AwgString12

global AwgString13, AwgString14, AwgString15, AwgString16

if FSweepMode.get() == 1: # If doing a frequency sweep only make new AWG A sine wave

if AWGAShape.get()==1:

AWGAMakeSine()

AWGAShapeLabel.config(text = AwgString1) # change displayed value

return

# Shape list

if AWGAShape.get()== 0:

AWGAMakeDC()

AWGAShapeLabel.config(text = "DC") # change displayed value

elif AWGAShape.get()==1:

AWGAMakeSine()

AWGAShapeLabel.config(text = AwgString1) # change displayed value

elif AWGAShape.get()==2:

AWGAMakeSquare()

AWGAShapeLabel.config(text = AwgString2) # change displayed value

elif AWGAShape.get()==3:

AWGAMakeTriangle()

AWGAShapeLabel.config(text = AwgString3) # change displayed value

elif AWGAShape.get()==4:

AWGAMakePulse()

AWGAShapeLabel.config(text = AwgString4) # change displayed value

elif AWGAShape.get()==5:

AWGAMakeRampDn()

AWGAShapeLabel.config(text = AwgString5) # change displayed value

elif AWGAShape.get()==6:

AWGAMakeRampUp()

AWGAShapeLabel.config(text = AwgString6) # change displayed value

elif AWGAShape.get()==7:

AWGAMakeStair()

AWGAShapeLabel.config(text = AwgString7) # change displayed value

elif AWGAShape.get()==8:

AWGAMakeSinc()

AWGAShapeLabel.config(text = AwgString8) # change displayed value

elif AWGAShape.get()==9:

AWGAMakeSine()

AWGAShapeLabel.config(text = AwgString9) # change displayed value

elif AWGAShape.get()==10:

AWGAMakeFullWaveSine()

AWGAShapeLabel.config(text = AwgString10) # change displayed value

elif AWGAShape.get()==11:

AWGAMakeHalfWaveSine()

AWGAShapeLabel.config(text = AwgString11) # change displayed value

elif AWGAShape.get()==12:

AWGAMakeFourier()

AWGAShapeLabel.config(text = AwgString12) # change displayed value

elif AWGAShape.get()==13:

SetAwgSampleRate()

AWGAAmplvalue = float(eval(AWGAAmplEntry.get()))

AWGAOffsetvalue = float(eval(AWGAOffsetEntry.get()))

AWGAFreqvalue = UnitConvert(AWGAFreqEntry.get())

NrTones = int(eval(AWGADutyCycleEntry.get()))

ampl = 3.0/NrTones

if ampl > 0.25:

ampl = 0.25

AWGASendWave(SchroederPhase(MaxSamples, NrTones, ampl))

AWGAShapeLabel.config(text = AwgString13) # change displayed value

elif AWGAShape.get()==14:

AWGAMakeUUNoise()

AWGAShapeLabel.config(text = AwgString14) # change displayed value

elif AWGAShape.get()==15:

AWGAMakeUGNoise()

AWGAShapeLabel.config(text = AwgString15) # change displayed value

else:

AWGAShapeLabel.config(text = "Other Shape") # change displayed value

#

if BisCompA.get() == 1:

SetBCompA()

#

if AWGBShape.get() == 0:

AWGBMakeDC()

AWGBShapeLabel.config(text = "DC") # change displayed value

elif AWGBShape.get() == 1:

AWGBMakeSine()

AWGBShapeLabel.config(text = AwgString1) # change displayed value

elif AWGBShape.get() == 2:

AWGBMakeSquare()

AWGBShapeLabel.config(text = AwgString2) # change displayed value

elif AWGBShape.get() == 3:

AWGBMakeTriangle()

AWGBShapeLabel.config(text = AwgString3) # change displayed value

elif AWGBShape.get() == 4:

AWGBMakePulse()

AWGBShapeLabel.config(text = AwgString4) # change displayed value

elif AWGBShape.get()==5:

AWGBMakeRampDn()

AWGBShapeLabel.config(text = AwgString5) # change displayed value

elif AWGBShape.get()==6:

AWGBMakeRampUp()

AWGBShapeLabel.config(text = AwgString6) # change displayed value

elif AWGBShape.get()==7:

AWGBMakeStair()

AWGBShapeLabel.config(text = AwgString7) # change displayed value

elif AWGBShape.get()==8:

AWGBMakeSinc()

AWGBShapeLabel.config(text = AwgString8) # change displayed value

elif AWGBShape.get()==9:

AWGBMakeSine()

AWGBShapeLabel.config(text = AwgString9) # change displayed value

elif AWGBShape.get()==10:

AWGBMakeFullWaveSine()

AWGBShapeLabel.config(text = AwgString10) # change displayed value

elif AWGBShape.get()==11:

AWGBMakeHalfWaveSine()

AWGBShapeLabel.config(text = AwgString11) # change displayed value

elif AWGBShape.get()==12:

AWGBMakeFourier()

AWGBShapeLabel.config(text = AwgString12) # change displayed value

elif AWGBShape.get()==13:

SetAwgSampleRate()

AWGBAmplvalue = float(eval(AWGBAmplEntry.get()))

AWGBOffsetvalue = float(eval(AWGBOffsetEntry.get()))

AWGBFreqvalue = UnitConvert(AWGBFreqEntry.get())

NrTones = int(eval(AWGBDutyCycleEntry.get()))

ampl = 3.0/NrTones

if ampl > 0.25:

ampl = 0.25

AWGBSendWave(SchroederPhase(MaxSamples, NrTones, ampl))

AWGBShapeLabel.config(text = AwgString13) # change displayed value

elif AWGBShape.get()==14:

AWGBMakeUUNoise()

AWGBShapeLabel.config(text = AwgString14) # change displayed value

elif AWGBShape.get()==15:

AWGBMakeUGNoise()

AWGBShapeLabel.config(text = AwgString15) # change displayed value

else:

AWGBShapeLabel.config(text = "Other Shape") # change displayed value

#

time.sleep(0.01)

#

**Triggering functions:**

Triggering can be done in two possible ways, either in the hardware (hardware triggering) or in the software after the input sample arrays have been captured.

Hardware triggering is of course handled in the hardware itself and the appropriate functions need to be defined here to send those commands to the hardware.

The main ALICE GUI code contains functions to implement software triggering and those functions are generally included as part of the Get\_Data() function. Again developers are recommended to familiarize themselves with how this is accomplished in an existing example file.

The main GUI uses the following function names to control triggering:

def BSetTriggerSource():

def BSetTrigEdge():

def BTriggerMode():

def SendTriggerLevel():

def BTrigIntExt():

**PWM Output Controls:**

If the device supports a digital PWM output then these two functions can be defined to send the appropriate commands to the hardware.

#

# Hardware Specific PWM control functions

#

def PWM\_On\_Off():

global PWM\_is\_on, ser

if PWM\_is\_on:

#print("Set pwm on")

ser.write(b'so\n') # send on command here

else:

#print("Set pwm off")

ser.write(b'sx\n') # send off command here

#

def UpdatePWM():

global PWMDivEntry, PWMWidthEntry, PWMLabel, ser

PWMLabel.config(text = "PWM Frequency") # text label in GUI can be changed for Freq or Peroid

FreqValue = int(UnitConvert(PWMDivEntry.get()))

#PeriodValue = int(( 133e6 / 256 ) / FreqValue)

#print("FreqValue = ", FreqValue, "PeriodValue = ",PeriodValue)

ByteStr = 'p' + str(FreqValue) + "\n"

SendByt = ByteStr.encode('utf-8')

ser.write(SendByt) # send frequency or period set command here

time.sleep(0.1)

DutyCycle = int(PWMWidthEntry.get())

#WidthFraction = float((DutyCycle/100.0))

#Width = int(PeriodValue \* WidthFraction)

ByteStr = 'm' + str(DutyCycle) + "\n"

SendByt = ByteStr.encode('utf-8')

ser.write(SendByt) # send width set command here

time.sleep(0.1)

#