USER MANUAL FOR RZ6 PROGRAM

The biox\_rz6 framework

© Radboud University

By Ruurd Lof

Biox version 3.25

Last changes 15/12/2020

version 3.25

* buttonbox echo added: [Method rz6.write\_buttonbox\_echo()](#_Method_rz6.write_buttonbox_echo())

version 3.24

* bugfix with respect to trigger events and hold input byte
* soft trigger 3 removed in [Waitfortrigger](#_A2__‘Waitfortrigger’)
* ‘loop’ option added to sound type ‘WAV’
* ‘reset’ and ‘continue’ will not loop the WAV

version 3.23:

* read\_rcx\_version and read\_biox\_version added
* ‘SoundAB’ functionality added [Task types ‘SoundA’,‘SoundB’ and ‘SoundAB’](#_Task_types_‘SoundA’)
* ‘SoundA’ and ‘SoundB’ are not synchronized when used seperately
* ‘AttA’ and ‘AttB’ combined in [Task type ‘Att’](#_Task_type_‘Att’)
* ‘ITD’ added in [Task type ‘ITD’](#_Task_type_‘ITD’)
* Note to ITD with NOISE in [Task type ‘ITD’](#_Task_type_‘ITD’)
* Added notes on ITD or ILD measurements [Appendix E](#_Appendix_E:_Notes)

version 3.22:

* Edited description [MIX task](#_Task_type_‘Mix’)
* Added parameter boundaries for tasks in [Appendix A](#_Appendix_A:_Debugging)
* Added MIX task in [Appendix A](#_Appendix_A:_Debugging)
* RCX change: audible tic removed at sound stop
* Common source option in Noise sound as text 2.4.4.4
* Ripple RCX file debugged

version 3.21:

* WAV data in rows or collums 2.3.2
* Common source and ITD option in Noise sound 2.4.4.4
* Delay of 6 cycles for start of sound A in case of WAV 2.4.4.6
* Bugfixs of task ‘SoundB’, ‘B=A’
* function ReadVersion() Added 2.3.19

Table of Contents

[1 The biox\_rz6 framework 5](#_Toc48126829)

[1.1 Introduction 5](#_Toc48126830)

[1.2 What can biox\_rz6 do? 5](#_Toc48126831)

[1.3 Matlab interface versions 5](#_Toc48126832)

[1.4 RCX Versions 6](#_Toc48126833)

[1.5 Programming a Trial 6](#_Toc48126834)

[2 The MATLAB interface 7](#_Toc48126835)

[2.1 Introduction 7](#_Toc48126836)

[2.2 The Zbus client 7](#_Toc48126837)

[2.2.1 Method zbus.trigger() 7](#_Toc48126838)

[2.3 The RZ6 client object 7](#_Toc48126839)

[2.3.1 Method rz6.write\_tasklist() 8](#_Toc48126840)

[2.3.2 Method rz6.write\_wavdata() 8](#_Toc48126841)

[2.3.3 Method rz6.read\_acqdata() 9](#_Toc48126842)

[2.3.4 Method rz6.read\_acqready() 9](#_Toc48126843)

[2.3.5 Method rz6.read\_acqsize() 9](#_Toc48126844)

[2.3.6 Method rz6.read\_trialready() 10](#_Toc48126845)

[2.3.7 Method rz6.read\_samplerate() 10](#_Toc48126846)

[2.3.8 Method rz6.write\_signalbyte() 10](#_Toc48126847)

[2.3.9 Method rz6.read\_inputbyte() 10](#_Toc48126848)

[2.3.10 Method rz6.read\_inputholdbyte() 10](#_Toc48126849)

[2.3.11 Method rz6.read\_responsetime() 10](#_Toc48126850)

[2.3.12 Method rz6.trigger() 11](#_Toc48126851)

[2.3.13 Method rz6.resetlist() 11](#_Toc48126852)

[2.3.14 Method rz6.mov\_spm\_dac () 11](#_Toc48126853)

[2.3.15 Method rz6.mov\_sp\_list () 11](#_Toc48126854)

[2.3.16 Method rz6.read\_tasklist(tasklist) 12](#_Toc48126855)

[2.3.17 Method rz6.read\_taskindex() 12](#_Toc48126856)

[2.3.18 Method rz6.read\_timer() 12](#_Toc48126857)

[2.3.19 Method rz6.read\_rcx\_version() 12](#_Toc48126858)

[2.3.20 Method rz6.read\_biox\_version() 12](#_Toc48126859)

[2.4 The task list object 12](#_Toc48126860)

[2.4.1 Method tl.addtask() 13](#_Toc48126861)

[2.4.2 Method tl.debug 14](#_Toc48126862)

[2.4.3 Task type ‘WaitForTrigger’ 14](#_Toc48126863)

[2.4.4 Task types ‘SoundA’,‘SoundB’ and ‘SoundAB’ 15](#_Toc48126864)

[2.4.5 Task type ‘MUX’ (multiplexer control) 19](#_Toc48126865)

[2.4.6 Task type ‘HoldInput’ (determine subject response) 20](#_Toc48126866)

[2.4.7 Task type ‘SoundMov’ (moving sounds) 21](#_Toc48126867)

[2.4.8 Task type ‘Daq’ (data acquisition) 22](#_Toc48126868)

[2.4.9 Task type ‘SetDIO’ (digital output) 23](#_Toc48126869)

[2.4.10 Task type ‘TrigOut’ (output triggers) 23](#_Toc48126870)

[2.4.11 Task type ‘MultiConfigA’ or ‘MultiConfigB’ 24](#_Toc48126871)

[2.4.12 Task type ‘Att’ (attenuation control) 25](#_Toc48126872)

[2.4.13 Task type ‘ITD’ 26](#_Toc48126873)

[2.4.14 Task type ‘Mix’ (mix sounds) 26](#_Toc48126874)

[2.4.15 Task type ‘Reset’ (reset task list) 27](#_Toc48126875)

[2.4.16 Task type ‘Ready’ (set ready flag) 27](#_Toc48126876)

[2.5 Matlab files 27](#_Toc48126877)

[3 The functionality of the RCX file 27](#_Toc48126878)

[3.1 Introduction 27](#_Toc48126879)

[3.2 Tasks 28](#_Toc48126880)

[3.3 Timing of tasks 28](#_Toc48126881)

[3.3.1 Use event recorder for timestamping the tasks 29](#_Toc48126882)

[3.3.2 Use data-acquisition for checking the task timing 30](#_Toc48126883)

[3.4 DAC and ADC channels. 30](#_Toc48126884)

[3.4.1 Introduction 30](#_Toc48126885)

[3.4.2 DAC channels: playing sounds 30](#_Toc48126886)

[3.4.3 ADC channels: recording 30](#_Toc48126887)

[3.5 Digital I/O 31](#_Toc48126888)

[3.5.1 Introduction 31](#_Toc48126889)

[3.5.2 Byte A, input signals, triggers 31](#_Toc48126890)

[3.5.3 Byte B, Output signals, triggers 31](#_Toc48126891)

[3.5.4 Byte C, Multiplexer interface 32](#_Toc48126892)

[3.6 DAC offset compensation 32](#_Toc48126893)

[3.7 ITD experiments 32](#_Toc48126894)

[3.7.1 ITD with ‘noise’ sound 32](#_Toc48126895)

[3.7.2 ITD with WAV 33](#_Toc48126896)

[Appendix A: Debugging info 34](#_Toc48126897)

[A1 Introduction 34](#_Toc48126898)

[A2 ‘Waitfortrigger’ 34](#_Toc48126899)

[A3 ‘SoundA’, ‘SoundB’ or ‘SoundAB’ 34](#_Toc48126900)

[A4 ‘MUX’ 35](#_Toc48126901)

[A5 ‘HoldInput’ 35](#_Toc48126902)

[A6 ‘Daq’ 35](#_Toc48126903)

[A7 ‘DIOout’ 36](#_Toc48126904)

[A8 ‘TRGout’ 36](#_Toc48126905)

[A9 ‘ConfigureA’ or ‘ConfigureB’ 36](#_Toc48126906)

[A10 ‘Reset’ 36](#_Toc48126907)

[A11 ‘Ready’ 36](#_Toc48126908)

[A12 ‘Att’ 36](#_Toc48126909)

[A13 ‘ITD’ 37](#_Toc48126910)

[A14 ‘MIX’ 37](#_Toc48126911)

[A15 ‘SoundMov’ 37](#_Toc48126912)

[A16 ‘Stop’ 37](#_Toc48126913)

[A17 ‘Tone’ 37](#_Toc48126914)

[A18 ‘Sweep’ 38](#_Toc48126915)

[A19 ‘Noise’ 38](#_Toc48126916)

[A12 ‘Ripple’ 38](#_Toc48126917)

[A21 ‘WAV’ 38](#_Toc48126918)

[A22 ‘MultiTone’ 38](#_Toc48126919)

[A23 ‘B=A’ 38](#_Toc48126920)

[Appendix B: Matlab classes 39](#_Toc48126921)

[B1 biox\_abstract\_client 39](#_Toc48126922)

[B2 biox\_rz6\_client 39](#_Toc48126923)

[B3 biox\_rz6\_<x>c (and others) 39](#_Toc48126924)

[B4 biox\_PAT\_LAB (and others) 39](#_Toc48126925)

[Appendix C: Hardware accessories 40](#_Toc48126926)

[C1 Patch panel (PP\_RZ6\_Digital-I/O) 40](#_Toc48126927)

[C2 Response box 40](#_Toc48126928)

[C3 RA8GA Adjustable Gain Preamp 41](#_Toc48126929)

[C4 Response button 41](#_Toc48126930)

[Appendix D: Examples 42](#_Toc48126931)

[D1 Simple Trial 42](#_Toc48126932)

[D2 Moving sound and reaction time 43](#_Toc48126933)

[D3 Recording head movement and playing ‘WAV’-file 44](#_Toc48126934)

[D4 Recording of the task timing 45](#_Toc48126935)

[Appendix E: Notes on ITD or ILD measurements 47](#_Toc48126936)

[E1 Notes on ITD measurements 47](#_Toc48126937)

[E2 Notes on ILD measurements 47](#_Toc48126938)

# The biox\_rz6 framework

## Introduction

A framework, called “biox\_rz6”, is developed for performing a variety of auditory experiments with the RZ6 multi I/O processor from Tucker Davis. This framework consists of different versions of an RCX program for single, triple or quadruple core RZ6 and a set of MATLAB objects and functions. An RCX file is the software that is uploaded into the RZ6 device in order to control the experiment. The MATLAB code for an experiment uses MATLAB objects in order to communicate with the RCX program on the RZ6.

The goal of this framework is to minimise the effort of the experimenter to realize a variety of experiment that uses the RZ6 as a tool. The experimenter should be able to use the MATLAB interface without knowing about the intricacies and complications of the RCX program running on the RZ6. The name “biox” is derived from BIOphysics eXperiments.

## What can biox\_rz6 do?

* Produce different types of [sounds](#_Task_types_‘SoundA’) on two channels independently
  + [Pure and modulated tones](#_Sound_type_‘Tone’)
  + [Stacked tones](#_Sound_type_‘MultiTone’) (e.g. harmonics)
  + [Noise](#_Sound_type_‘Noise’)
  + [Sweeps](#_Sound_type_‘Sweep’) (whoops)
  + [Ripple sounds](#_Sound_type_‘Ripple’)
  + [Preloaded sounds](#_Sound_type_)
* [Bounce sounds between any pair of speakers](#_Sound_type_‘B=A’)
* [Move sounds through an array of speakers](#_Task_type_‘SoundMov’)
* [Mix any pair of sound types](#_Task_type_‘Mix’)
* [Register button presses and measure response times](#_Task_type_‘HoldInput’)
* [Give LED feedback](#_Task_type_‘SetDIO’)
* [Receive](#_Task_type_‘WaitForTrigger’) and [send](#_Task_type_‘TrigOut’) triggers
* [Control multiplexers](#_Task_type_‘MUX’)
* [Perform data acquisition](#_Task_type_‘Daq’)
* [Accurate time registration of internal and external events](#_Timing_of_tasks)
* ITD experiments
* ILD experiments

## Matlab interface versions

For the communication between matlab and the RZ6 hardware an RZ6\_client class is used. There are specific versions for every lab.

The following versions of the matlab code files for the specific RZ6 clients will be available:

* biox\_PAT\_LAB.m
* biox\_EEG\_NIRS.m
* biox\_VST\_CHR.m
* biox\_SPK\_ARM.m
* biox\_SPH\_LAB.m (in future when extra 4 DSP RZ6 is available)

Chapter 2 covers the Matlab interface with the RZ6.

## RCX Versions

A standard 3DSP RCX file is developed for the 3DSP (and 4DSP) version of the RZ6. Chapter 3 and 4 covers the functionality of this version. For testing purposes a 1DSP version is available with limited functionality. The 3DSP and 4DSP versions run at a clock rate of 50 kHz, the 1DSP at a lower clock rate of 25 kHz. For special purposes two extra 4DSP versions are develloped. One for moving sounds along a speaker array and another for playing ripple sounds. These versions will be covered in chapter 6.

The names of the different RCX file versions is as follows:

* biox\_rz6\_1C\_25kHz.rcx single DSP version
* biox\_rz6\_3C\_50kHz.rcx triple DSP version (standard)
* biox\_rz6\_4C\_50kHz.rcx quad DSP version (standard)
* biox\_rz6\_4C\_50kHz\_MOV.rcx quad DSP version for moving sounds
* biox\_rz6\_4C\_50kHz\_RIP.rcx quad DSP version for ripple sounds

The 1C version misses some functionality with respect to the 3C version. The Ripple sound version is identical to the 1C version with only the ripple sound functionality added.

## Programming a Trial

A trial is specified as a task list in MATLAB. The task list is transformed into a matrix that is uploaded to the RZ6 program. This enables the RZ6 to perform a trial with a large number of different tasks without any interference of the MATLAB program.

[Chapter 2](#_The_MATLAB_interface) will be concerned with how to create the task list and how to communicate with the RZ6.

When a trial consists of an interactive session, where the response of the user determines the follow up tasks, the Matlab code should contain the following steps:

1. Create a new task list
2. Upload the task list to the rz6
3. Let Matlab code wait for the task list to return ready (with rz6.read\_trialready())
4. Read the user response (e.g. with rz6.read\_inputholdbyte())

Repeat from step 1.

The last task should be of task type ‘Ready’.

%step 1: create task list

tl.add\_task(0.000,'WaitForTrigger','External', bin2dec('0000 1111');

tl.add\_task(0.000,'SoundA','noise', 2000, 200);

tl.add\_task(1.000,'SoundA', 'stop');

tl.add\_task(1.000,'Ready');

%step 2: upload list

rz6.write\_tasklist(tl);

%step 3: wait for trial to be ready

while ~rz6.read\_trialready()

%do nothing

end

%step 4: read user response

input byte = rz6.read\_inputholdbyte();

# The MATLAB interface

## Introduction

The matlab interface for the RZ6 consists of three objects. One is a RZ6 client object that does the communication with the RCX program that runs on the RZ6. The second is a TaskList object, that will be used to create a list of instructions that makes up a trial. The third object is a Zbus client for communication with the Zbus (e.g. for generating Zbus triggers). In the following paragraphs the Zbus object, the RZ6 object and the task list will be discussed and each paragraph will have a matlab code example in an outline box.

All parameters are to be given in standard SI units.

## The Zbus client

The Zbus client object <biox\_zbus\_client> is used for creating Zbus triggers.

|  |
| --- |
| zbus = biox\_zbus\_client(); %creates zbus client object |

biox\_zbus has only one Method: <trigger>

### Method zbus.trigger()

The function <trigger> creates a general bus trigger in the Zbus. The input parameters is a string. Valid strings are ‘A’ and ‘B’.

|  |
| --- |
| zbus.trigger('A'); %generates a general ZbusA trigger |

When it is necessary that Zbus triggers are specific for a certain caddy, the ActiveX component for the zbus should be used instead of the zbus client.

## The RZ6 client object

For the communication with an RZ6 the parent class <biox\_rz6\_client> is developed. When an instance of this class is created the object will connect to the ZBus and the RZ6 and upload an RCX file.

For every RZ6 in the lab there is a child class <biox\_rz6\_xxxxxx>, where xxxxxx specifies the lab. The constructor of this class will initialize several parameters concerning the specific hardware configuration of the setup.

The following rz6\_client child classes will be available (see also [Appendix B: Matlab classes](#_Appendix_B:_Matlab)):

* biox\_PAT\_LAB
* biox\_EEG\_NIRS
* biox\_VST\_CHR
* biox\_SPK\_ARM
* biox\_SPH\_LAB (in future)

A matlab example for creating an RZ6 client object for the patientlab:

|  |
| --- |
| rz6 = biox\_PAT\_LAB; %creates an rz6 client object |

The RZ6 client class has the following methods (member functions):

* write\_tasklist() writes task list to RZ6
* write\_wavdata() writes wav data to RZ6
* read\_acqdata() reads acquired data from RZ6
* read\_acqready() reads acquisition is ready flags from RZ6
* read\_acqsize() reads the size of the acquired data from the RZ6
* read\_trialready() reads trial is ready flag from RZ6
* read\_samplerate() reads sample rate from RZ6
* write\_signalbyte() writes signal byte to RZ6
* read\_inputbyte() reads input byte from RZ6
* read\_inputholdbyte() reads input hold byte from RZ6
* read\_responsetime() reads input response time from RZ6
* trigger() generates a Soft trigger to the RZ6
* startlist() (re) starts the task list from the beginning

Only availlable on subclass for moving sounds:

* mov\_spm\_dac() specifies coupling between output channels and muxes
* mov\_sp\_list() specifies the speaker list for moving sounds

Parameters boundaries

### Method rz6.write\_tasklist()

The function <write\_tasklist> writes a list of tasks to the RZ6. The function takes a task list (an instance of the class <biox\_rz6\_tasklist>) as input parameter.

|  |
| --- |
| tl=biox\_rz6\_tasklist; %create task list object  ..... %add tasks to list  ..... %add tasks to list  rz6.write\_tasklist(tl); %write task list to RZ6 |
|  |

### Method rz6.write\_wavdata()

The function <write\_wavdata> writes sound wave data to the RZ6. The function takes two parameters as input: the first parameters is an array of floating points (-10<=value<=10) that specifies the sound wave(s), the second is an array of integers that specifies the channels for which the sound data is meant. OUT-A is channel 1 and OUT-B is channel 2.

N.B. The data for can be in the form of one or two collumns, or one or two rows.

|  |
| --- |
| rz6.write\_wavdata(data, chanlist); %write wav data to RZ6 |

When two different sounds are uploaded for channel A and B you should use:

|  |
| --- |
| rz6.write\_wavdata([data1;data2], [1 2]); %write wav data to RZ6 |

or

|  |
| --- |
| rz6.write\_wavdata(data1, [1]); %write wav data to RZ6  rz6.write\_wavdata(data2, [2]); %write wav data to RZ6 |

### Method rz6.read\_acqdata()

The function <read\_acqdata> reads acquired data from the RZ6. The function takes one parameters as input: an array of integers that specifies the channels for which the acquisition data will be read.

* IN-A is channel 1 and IN-B is channel 2.
* OUT-A is channel 3 and OUT-B is channel 4.
* External ADC channels 1…6 (e.g. from RA8GA) correspond to channels 5…10.
* In case there are magnetic fields for head tracking the channels 5…7 normally correspond to the X, Y and Z output of the head tracker system.

The function returns a list of arrays. Each array is a row vector of floating points. Because the returned data can have different number of samples for different channels the data returned in the form of a cell array.

|  |
| --- |
| chanlist = 1 2; %specify the channels to read from  acqdata = rz6.read\_acqdata(chanlist); %read acq data from RZ6  data1 = acqdata{1}; %acqdata is a cell array  data2 = acqdata{2}; %acqdata{2} returns data for chan 2 |

### Method rz6.read\_acqready()

The function <read\_acqready> reads a list of Boolean flags from the RZ6 that indicates which channel is ready with its acquisition task. The function takes one parameters as input: an array of integers that specifies the channels for which the flags are read out.

|  |
| --- |
| chanlist = 1 2; %specify the channels to read  acqready = rz6.read\_acqready(chanlist); %reads acq ready flags from RZ6 |

### Method rz6.read\_acqsize()

The function <read\_acqready> reads a list of Boolean flags from the RZ6 that indicates which channel is ready with its acquisition task. The function takes one parameters as input: an array of integers that specifies the channels for which the flags are read out.

|  |
| --- |
| chanlist = 1 2; %specify the channels to read  acqsize = rz6.read\_acqsize(chanlist); %reads acq sizes from RZ6 |

### Method rz6.read\_trialready()

The function <read\_trialready> reads a Boolean flag from the RZ6 that indicates that the trial is ready. The function takes no parameters as input. It returns a Boolean value.

|  |
| --- |
| trialready = rz6.read\_trialready(); %reads trial ready flag from RZ6 |

### Method rz6.read\_samplerate()

The function <read\_samplerate> reads a value from the RZ6 that indicates that the sample rate the RZ6 is running at. The function takes no parameters as input. It returns an integer value.

|  |
| --- |
| samplerate = rz6.read\_samplerate(); %reads sample rate from RZ6 |

### Method rz6.write\_signalbyte()

The function <write\_signalbyte> writes the byte to the RZ6 that determines the digital output channel(s) that is (are) used for sending signals to the event recorder. The function takes one parameters as input: a byte. The bits correspond to outputs B0…B7.

|  |
| --- |
| signalbyte = 128; %128 corresponds to output bit B7  rz6.write\_signalbyte(signalbyte); %write signal byte to RZ6 |

### Method rz6.read\_inputbyte()

The function <read\_inputbyte> reads the byte from the RZ6 that specifies the current status of the digital input channels. The function returns a byte. The bits correspond to inputs A0…A7.

|  |
| --- |
| input byte = rz6.read\_inputbyte(); %read input byte from RZ6 |

### Method rz6.read\_inputholdbyte()

The function <read\_inputholdbyte> reads the byte from the RZ6 that specifies the digital input channels that have been high since the last reset of the input byte. The function returns a byte. The bits correspond to inputs A0…A7. Multiple inputs can be recorded as high at the same time.

|  |
| --- |
| input byte = rz6.read\_inputholdbyte(); %read input byte from RZ6 |

### Method rz6.read\_responsetime()

The function <read\_response> reads the response time (in seconds) of the input response, after the task ‘HoldInput’ is performed. The function returns an floating point. The accuracy of the measurement is about 0.1 ms when the RZ6 is running at 48.8 kHz.

|  |
| --- |
| input byte = rz6.read\_inputholdbyte(); %read input byte from RZ6 |

### Method rz6.trigger()

The function <trigger> sends a soft trigger to the RZ6. The input parameters is a string. Valid strings are ‘soft1’ and ‘soft2’.

|  |
| --- |
| rz6.trigger('soft1'); %generates a soft1 trigger in the RZ6 |

### Method rz6.resetlist()

The function <resetlist> resets and starts the task list. This is the same as calling soft trigger(4)

|  |
| --- |
| rz6.resetlist(); %starts task list from beginning |

### Method rz6.mov\_spm\_dac ()

The first speaker ‘sp0’ is connected to ‘OUT-A’ or ‘OUT-B’. The default value is ‘OUT-A’.

|  |
| --- |
| rz6.mov\_spm\_dac('OUT-A'); %couple middle speaker to a DAC output. |

### Method rz6.mov\_sp\_list ()

The method rz6.mov\_sp\_array adds a list of speakers for making moving sounds over multiple speakers. The speakers should be listed from left to right. The total number of speakers should be odd. With the method rz6.mov\_sp0\_ch() the audio output channel ‘A’ is connected to either all even or all odd speakers. The speaker list should contain in the first collumn the mux id’s and in the second collumn the mux channels. The list should contain an odd number of rows and a maximum of 21 rows.

N.B. The even speakers and odd speakers should be physically connected to different DAC outputs of the RZ6.

|  |
| --- |
| % create list with mux\_id’s and mux\_channels of the speakers  sp\_list = [2 0  3 0  2 1  3 1  2 2  3 2  2 3  3 3  2 4  3 4  2 5  3 5  2 6  3 6  2 7  3 7  2 8  3 8  2 9  3 9  2 10]  rz6.mov\_spm\_dac('OUT-A'); % connect even speakers to the DAC 'OUT-A'  rz6.mov\_sp\_list(sp\_list); % upload speaker list |

### Method rz6.read\_tasklist(tasklist)

Reads the content of the tasklist matrix back from the RZ6.

### Method rz6.read\_taskindex()

Reads the index of the last performed task from the RZ6. The first task has index 1.

### Method rz6.read\_timer()

Reads the time from the rz6 that has been passed since the beginning of the execution of the tasklist or the since the ending of the last ‘Wait for trigger’-task by a trigger. The time is given in seconds.

### Method rz6.read\_rcx\_version()

Reads the version of BIOX from the RZ6 code. The function returns a string.

|  |
| --- |
| >> version = rz6.read\_rcx\_version()  version =  ‘BIOX RCX V3.23’ |

### Method rz6.read\_biox\_version()

Reads the version of BIOX from the Matlab code. The function returns a string.

|  |
| --- |
| >> version = rz6.read\_biox\_version()  version = ‘BIOX MATLAB V3.23’ |

### Method rz6.write\_buttonbox\_echo()

Sets the buttonbox echo ‘on’ or ‘off’. The buttonbox echo is ‘On’ by default. Optional is a second variable that sets the minimual duration of the buttonbox echo. The minimal duration is 0.5 seconds by default. When the buttonbox echo is on, the LEDs on the ButtonBox will display the status of the corresponding input on the ButtonBox. When a button is pressed for less than the minimal duration, the LED will be on for the set minimal duration. When the button is pressed longer than the minimal duration the LED will be on for as long as the button is pressed. The minimal duration cannot be longer than 10 seconds.

|  |
| --- |
| minimal\_duration = 0.5; %duration in seconds  rz6.write\_buttonbox\_echo('On', minimal\_duration);  rz6.write\_buttonbox\_echo('Off'); |

## The task list object

The task list object is used for specifying the tasks that the RCX program has to perform within a single trial. The specifics of the tasks will be covered in the following paragraphs.

Before you can specify the task list, you have to call the function that creates the task list object:

|  |
| --- |
| tl=biox\_rz6\_tasklist; %create a task list |

The task list object has the following members:

* add\_task() Add a task to the list
* debug() Set debug mode on/off

### Method tl.addtask()

The function <addtask> will add a new task to the task list. The function can have up to seven parameters:

* delay mandatory - specifies a delay in ms with respect to the last trigger
* task type mandatory - specifies the task type
* sound type optional - specifies the sound type in case of a sound task
* par1 optional – context dependent
* par2 optional - context dependent
* par3 optional - context dependent
* par4 optional - context dependent

|  |
| --- |
| delay = <some value>;  tasktype '<some task type>';  soundtype = '<some sound type>';  par1 = <some value>;  par2 = <some value>;  par3 = <some value>;  par4 = <some value>;  tl.add\_task(delay, tasktype, soundtype, par1, par2, par3, par4); |

The delay is specified by an integer and the unit is in milliseconds. The range is 0 to  2,147,483,647. For the use of the delay parameter see $3.3 Timing of tasks.

The task types are specified by a name identifier of type string and the names and their function are listed below:

|  |  |
| --- | --- |
| **Name** | **Description** |
| ‘WaitForTrigger’ | RZ6 waits for trigger |
| ‘SoundA’ or ‘SoundB’ | Start or stop sound at output ch.A or ch.B |
| ‘MUX’ | Open channel on a mux or reset a mux |
| ‘HoldInp’ | Remember any input byte set high |
| ‘SoundMov’ | Start or stop moving sound |
| ‘Daq’ | Start or stop data acquisition |
| ‘SetDIO’ | Set digital out byte |
| ‘TrigOut’ | Sent a single or double trigger |
| ‘Reset’ | Reset the state machine to the first task |
| ‘Ready’ | Set the TSK\_Ready flag high |
| ‘MultiConfigA’ or ‘MultiConfigB’ | Set configuration for multi tone sound ch.A or ch.B |
| ‘AttA’ or ‘AttB’ | Set attenuation for ch.A or ch.B |
| ‘Mix’ | Mixes sounds |

N.B. ‘SoundMov’ can only be used with the ‘biox\_rz6\_4C\_50kHz\_M.rcx’ file.

### Method tl.debug

The function debug() has one Boolean input parameter <onoff>.When set high the task list object is set in debug mode. When the debug mode is used the add\_task function will output an array of seven integers representing the task.

|  |
| --- |
| onoff = 1; %1=on, 0=off  tl.debug(onoff); %set debug mode |

### Task type ‘WaitForTrigger’

The task ‘waitfortrigger’ is used when the program needs to wait for an event. There are three types off events that can trigger the program to continue: 1) a Zbus trigger is detected, 2) an input bit goes high or 3) a software trigger ‘soft’ is detected. The parameter <triggertype> determines which of the three event types can be used as a trigger for the program to continue:

* ‘ZbusA’ Zbus trigger A
* ‘ZbusB’ Zbus trigger B
* ‘External’ Input byte A, with mask specified by PAR2
* ‘Soft1’ Software trigger 1
* ‘Soft2’ Software trigger 2

The program does only react to triggers after the delay has expired.

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'WaitForTrigger'; %task type  triggertype = 'ZbusB'; %trigger type  tl.add\_task(delay, tasktype, triggertype); %add task to list |

When the option ‘external’ is used an extra parameter <inputmask> determines which input bits of input byte A can be used as a trigger:

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'WaitForTrigger'; %task type  triggertype = 'External'; %trigger type  inputmask = bin2dec('0100 0000'); %trigger input is bit 6  tl.add\_task(delay, tasktype, triggertype, inputmask); |

N.B. When a WaitForTrigger task has a delay of 0,

### Task types ‘SoundA’,‘SoundB’ and ‘SoundAB’

The task ‘SoundA’, ‘SoundB’ and ‘SoundAB’ can start or stop a sound at DAC output OUT-A, OUT-B or both respectively. The duration of the sound is determined by the time scheduled between the start command and the stop command. The sounds at the channels A and B can be programmed completely independent of each other, but ‘SoundAB’ will program both sounds in one task.

It is also possible to couple the sound at channel B to the sound defined for channel A. This coupling can only be activated by the sound task for channel B.

Note that when between the start command and the stop command a ‘wait for trigger’ task is used the sound will just continue during the wait and the sound duration is therefore not pre-determined.

When ‘SoundAB’ is used, the sounds for both channel are most of the time equal except for the ‘WAV’ sounds (both channels can play their own WAV) and for ‘Noise’ sounds when the different settings for the filters are used.

N.B. All sounds ramp up and ramp down with a squared cosine function with a rise and fall time of 10 ms in order to avoid transient ‘clicks’.

|  |
| --- |
| tasktype = 'SoundA';  tl.add\_task(delay, task type , soundtype, par1, par2, ....); |

The following sound types can be used for the task type ‘SoundA’ and ‘SoundB’:

|  |  |
| --- | --- |
| **Name** | **Description** |
| ‘Stop’ | Stop sound |
| ‘Tone’ | Pure or modulated tone |
| ‘Sweep’ | Frequency sweep (whoop) |
| ‘Noise’ | Gaussian noise |
| ‘Ripple’ | Ripple sound |
| ‘WAV’ | Array of floating point data @ 48.8 kHz\* |
| ‘MultiTone’ | Sum of up to four pure tones |

\* 24.4 kHz in single Core RZ6

N.B. Sound type ‘B=A’ wich couples channel B to channel A can only be used for the task type ‘SoundB’:

|  |  |
| --- | --- |
| ‘B=A’ | A and B have the same sound |

#### Sound type ‘Stop’

The sound type ‘stop’ is used to stop a sound.

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'SoundA'; %channel A  soundtype = 'Stop'; %stop sound  tl.add\_task(delay, tasktype, soundtype); |

#### Sound type ‘Tone’

The sound type ‘tone’ is used to start a pure tone or a frequency modulated tone. The parameter <tonefreq> determines the centre frequency of the tone. The parameter <modfreq> determines the frequency of the modulation. The parameter <modBW> specifies the modulation band width. If <modBW> is set to zero the sound is a pure sinewave.

Example for pure tone:

delay = 0.000; %delay in seconds

tasktype = 'SoundA'; %start a sound for channel A

soundtype = 'Tone'; %start a tone sound

tonefreq = 500; %centre frequency in Hz

tl.add\_task(delay, tasktype, soundtype, tonefreq);

Example for modulated tone

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'SoundA'; %start a sound for channel A  soundtype = 'Tone'; %start a tone sound  tonefreq = 500; %centre frequency in Hz  modfreq = 5; %modulation frequency in Hz  modBW = 20; %modulation band width in Hz  tl.add\_task(delay, tasktype, soundtype, tonefreq, modfreq, modBW); |

#### Sound type ‘Sweep’

The sound type ‘Sweep’ produces exponential sweeps of pure tones starting at a base frequency and going up a number of octaves at a specified pace. Exponential in this context means that the frequency is doubling in a fixed time. After the highest frequency is reached, the sound starts again at the start frequency.

The parameter <startfreq> specifies the start frequency of the sweep. The parameter <nroctaves> specifies the number of octaves that are covered by the sweep. The parameter <period> specifies the time (in milliseconds) it takes for a single sweep.

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'SoundA'; %start a sound for channel A  soundtype = 'Sweep'; %start a sweep sound  startfreq = 250; %start frequency in Hz  nroctaves = 5; %number of octaves  period = 0.2; %period in seconds  tl.add\_task(delay, tasktype, soundtype, startfreq, nroctaves, period); |

#### Sound type ‘Noise’

The sound type ‘Noise’ produces Gaussian Noise with a pass band determined by a low pass frequency <lpfreq> and a high pass frequency <hpfreq>. Outside of this frequency band the sound is filtered by 4th order Biquad filters.

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'SoundA'; %start a sound for channel A  soundtype = 'Noise'; %start a noise sound  hpfreq = 250; %high pass filter frequency in Hz  lpfreq = 5000; %low pass filter frequency in Hz    tl.add\_task(delay, tasktype, soundtype, hpfreq, lpfreq); |

* When a different order Biquad filter has to be used, the RPvdsEx file has to be altered. Ask the technical staff for that.
* The lpfreq should always be greater than the hpfreq.
* When the task ‘SoundAB’ is used both channels will share the same Noise source. Separate parameters can be specified for the channels A and B. When only two filter parameters are specified both channels will have the same filtering.

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'SoundAB'; %start a sound for channels A and B  soundtype = 'Noise'; %start a noise sound  hp-A = 250; %freq high pass channel A [Hz]  lp-A = 5000; %freq low pass channel A [Hz]  hp-B = 200; %freq high pass channel B [Hz]  lp-B = 2500; %freq low pass channel B [Hz]    tl.add\_task(delay, tasktype, soundtype, hp-A, lp-A, hp-B, lp-B ); |

#### Sound type ‘Ripple’

A ripple sound is a stack of tones that are modulated in frequency and time. When plotted the power levels of the sound in a spectrogram (log{frequency} versus time) it creates equidistant ripples. The angle of the ripples is determined by the modulation frequencies in the time <modintime> and frequency domain <modinfreq>. The <modinfreq> is specified as a phase (in degrees) per octave. The number of tones is 25 per octave and the number of octaves is three.

**N.B. The ‘Ripple’ task is only availlable in the Ripple version of the RCX file.**

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'SoundA'; %start a sound for channel A  soundtype = 'Ripple'; %start a ripple sound  startfreq = 200; %lowest frequency in the ripple sound  modintime = 5; %modulation in time (in Hz)  modinfreq = 360; %modulation in frequency (phase per octave)  moddepth = 0.5; %modulation depth (0..1)  tl.add\_task(delay, tasktype, soundtype, startfreq, modintime, modinfreq, moddepth); |

#### Sound type ‘WAV’

A sound can be represented as an array of floating point values. It is possible to upload an array (maximum of 1e+6 points for each channel) and play this as a stream. When the sound task is executed by default the buffer is reset so it will start playing from the beginning. The parameter ‘reset’ can be also given explicitly. If the value ‘Continue’ is used, the Sound will start playing without resetting the buffer.

The command SoundAB will start the playing WAV-data written in channel 1 to output A and the WAV-data written in channel 2 to output B.

In case of 2-channel mono or stereo sound, the task SoundAB must be used to ensure that the channels will be perfectly synchonized.

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'SoundA'; %start a sound for channel A  soundtype = 'WAV'; %start a wav sound  reset = 'Reset'; %start the sound from the beginning  tl.add\_task(delay, tasktype, soundtype, reset); |

A WAV can be interrupted and started again, playing from where it stopped:

tl.add\_task(0.000, 'SoundA', 'WAV', 'Reset');

tl.add\_task(0.100, 'SoundA', 'Stop');

tl.add\_task(0.200, 'SoundA', 'WAV', 'Continue');

tl.add\_task(0.250, 'SoundA', 'Stop');

tl.add\_task(0.400, 'SoundA', 'WAV', 'Continue');

tl.add\_task(0.450, 'SoundA', 'Stop');

#### Sound type ‘MultiTone’

The sound type ‘MultiTone’ plays the sound that consists of up to four pure tones that are pre-configured by the task ‘MultiConfigA’ or ‘MultiConfigB’.

**N.B. The multitone task is not available in the Ripple version of the RCX file.**

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'SoundA'; %sound in channel A  soundtype = 'MultiTone'; %multitone (pre configured)  tl.add\_task(delay, tasktype, soundtype); |

It is possible to change the sound while playing, by progamming the task ‘MultiConfigA’ or ‘MultiConfigB’.

|  |
| --- |
| tl.add\_task(0.000,'MultiConfigA',1,300,0,100); %define tone 1  tl.add\_task(0.000,'MultiConfigA',2,400,0,100); %define tone 2  tl.add\_task(0.000,'SoundA','MultiTone'); %play multitone (1+2)  tl.add\_task(0.500,'MultiConfigA',2,500,0,100); %change tone 2 during play  tl.add\_task(1.000, 'SoundA', 'Stop'); |

If more than four tones are necessary you can use the ‘Mix’ task in order to mix multitone A and multitone B together to channel A or channel B (or both). This way up to eight tones can be stacked together.

|  |
| --- |
| tl.add\_task(0.000, 'SoundA', 'MultiTone');  tl.add\_task(0.000, 'SoundB', 'MultiTone');  tl.add\_task(0.000, 'Mix', 'AtoB'); %play Sound A+B in channel B |

#### Sound type ‘B=A’

The sound type ‘B=A’ couples the sound from channel B to the sound of channel A. The waveforms are equal and in phase with each other, but the amplitudes can vary. This creates a virtual source that can be moved on a line between the two speakers. The parameter <movtype> defines the movement type. The options are ‘Fixed’, ‘Sine’ and ‘Linear’. The first option gives a fixed virtual source between the speakers. The position of the virtual source is determined by the attenuation of the channels. The ‘sine’ and ‘linear’ option define a sinewave or a linear back and forth movement of the virtual source between the speaker. The parameter <period> defines the period of the movement in milliseconds. The parameter <phase> defines the phase of the movement in degrees (-180 to 180).

**N.B. The ‘B=A’ task is not availlable in the 1 Core and Ripple versions of the RCX file.**

There are three options: 1) ‘Fixed’ the virtual source is fixed in place; 2) ‘Sine’ the virtual source moves back and forth between the speakers with a velocity following a sine function; 3) ‘Linear’ the virtual source moves back and forth between the speakers with a linear velocity.

For the second and third option, a phase can be set anywhere between -180 and 180. The following situation apply to the phase:

|  |  |
| --- | --- |
| Phase | Action |
| -180 | Starts in the middle, moving left |
| -90 | Starts left, moving right |
| 0 | Starts in the middle, moving right |
| 90 | Starts right, moving left |
| 180 | Same as -180 |

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'SoundB'; %start a sound for channel B  soundtype = 'B=A'; %channel B coupled to channel A  movtype = 'Sine'; %virtual sound source moves as sinewave function  period = 5.0; %period of sinewave movement in seconds  phase = 90; %start sound at channel B  tl.add\_task(delay, tasktype, soundtype, movtype, period, phase, itd); |

With respect to timing the following two instructions are the same, but in case of ‘SoundAB’ both channels must be provided with a ‘WAV’.

rz6.write\_wavdata(wavdata, [1 2]); %write wav data to RZ6

tl.add\_task(0.000, 'SoundAB', 'WAV', 'Reset');

is equivalent to

rz6.write\_wavdata(wavdata, [1]); %write wav data to RZ6

tl.add\_task(0.000, 'SoundB', 'B=A', 'Fixed'); %B must be set first

tl.add\_task(0.000, 'SoundA', 'WAV', 'Reset');

### Task type ‘MUX’ (multiplexer control)

The RZ6 can be connected to up to four PM2R multiplexers. The multiplexers each have their own device number (0...3). The patch panel PP RZ6 Digital-I/O should be used for the connection between the PM2R’s to the RZ6 in order to make this task work. The task has input parameters for the device and channel that is to be activated. The use of the <reset> command results in all channels closed.

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'MUX'; %open or close a mux channel  channel = 7; %channel id (0..15)  device = 2; %device id (0..3)  action = 'Set'; %opens channel  tl.add\_task(delay, tasktype, device, action, channel); |

An example with ‘Reset’:

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'MUX'; %open or close a mux channel  device = 2; %device id (0..3)  action = 'Reset'; %closes all channels on device 2  tl.add\_task(delay, tasktype, device, action); |

N.B. On a certain multiplexer only one channel can be activated (open) at the same time. When using the task with the ‘Set’ action repeatedly without using ‘Reset’, each command will close the previous open channel and open the new one at the same moment.

### Task type ‘HoldInput’ (determine subject response)

The byte A of the digital I/O of the RZ6 is used for input signals e.g. from buttons. The ‘HoldInput’ task is used to hold an input value until it is read out by MATLAB and reset. The input channels that are held, are specified by a bit mask <inputmask>. This is useful when the user gives a response with the button box and you want to know which button has been pressed. The hold value can be read by calling:

|  |
| --- |
| Value = rz6.read\_inputbyte(); |

N.B. Each time the ‘HoldInput’ task is executed, the value of INP\_Byte is reset to zero.

When multiple inputs are detected as having been high between the holdinput task and the readout of the INP\_Byte, it will result in multiple bits of the INP\_Byte are flagged as high. The function read\_inputholdbyte() then returns a value that is the sum of the input id’s, if the byte is read as an integer value.

When the patch panel PP\_RZ6\_Digital-I/O is used the input bits are connected to the following hardware:

* Bit 0 = A0 (RBOXRZ6 Button)
* Bit 1 = A1 (RBOXRZ6 Button)
* Bit 2 = A2 (RBOXRZ6 Button)
* Bit 3 = A3 (RBOXRZ6 Button)
* Bit 4 = A4 (PP RZ6 Digital-I/O BNC)
* Bit 5 = A5 (PP RZ6 Digital-I/O BNC)
* Bit 6 = A6 (PP RZ6 Digital-I/O BNC)
* Bit 7 = A7 (PP RZ6 Digital-I/O BNC)

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'HoldInput'; %hold high inputs for readout  inputmask = bin2dec('0000 1111'); %set input mask to 00001111  tl.add\_task(delay, 'HoldInput', inputmask); |

### Task type ‘SoundMov’ (moving sounds)

The task ‘moving sound’ will move a sound smoothly over an array of speakers. Only the sound that is configured with task type ‘SoundA’ is used for the moving sound. ‘SoundB’ is ignored.

**N.B. The ‘SoundMov’ task is only availlable in the MOV version of the RCX file.**

Before you can use the task ‘SoundMov’ you have to specify the speakers that will be used by calling the method <rz6.mov\_sp\_list()>. See paragraph 2.3.13.

The preconfigured speaker array can contain a total of 21 speakers. The parameter <NumbSpeakers> identifies the number of speakers that are actually used of the array. This has to be an odd number that is smaller or equal to the total of the speakers in the array. The sound will always travel in a sinusoidal movement around the centre speaker of the array.

The parameter <Period> specifies the time (in seconds) that it will cost to move a complete cycle along the speakers (the sound is back where it started and moving in the same direction).

Parameter <Phase> specifies the starting phase (-180...180), which determines where the sound starts and in what direction:

|  |  |  |
| --- | --- | --- |
| -180 | Start in middle | Move left |
| -90 | Start left | Move right |
| 0 | Start in middle | Move right |
| 90 | Start right | Move left |
| 180 | Start in middle | Move left |

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'SoundMOV'; %Start or stop moving sound  startstop = 'Start'; %Start moving sound  NumSpeakers = 21; %Number of speakers 3...21 uneven  Period = 5.0; %Period in seconds  Phase = 90.0; %phase in degrees (-180...180)  tl.add\_task(delay, tasktype, startstop, NumSpeakers, Period, Phase ); |

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'SoundMOV'; %Start or stop moving sound  startstop = 'Stop'; %Stop moving sound  tl.add\_task(delay, tasktype, startstop); |

N.B. Some variables needed for the ‘SoundMov’ option to work are set in the constructor of the RZ6 class. See paragraphs 2.3.12 and 2.3.13.

### Task type ‘Daq’ (data acquisition)

The task type ‘Daq’ is used for recording of analog and digital signals. There are channels for recording signals from ADC’s or from digital inputs, but also channels that monitor signals that are generated at DAC’s or at digital outputs. The analog channels give arrays of The digital Daq-channels give an array integer output (0..255) representing a Byte.

|  |  |  |  |
| --- | --- | --- | --- |
| **Ch. Nr.** | **Function** | **Channel type** | **Signal** |
| 1 | IN-A | ADC | -10...10V |
| 2 | IN-B | ADC | -10...10V |
| 3\* | internal monitor of OUT-A (DAC) | DAC | -10...10V |
| 4\* | internal monitor of OUT-B (DAC) | DAC | -10...10V |
| 5 | External channel 1 (ADC) | ADC | -10...10V |
| 6 | External channel 2 (ADC) | ADC | -10...10V |
| 7\*\* | External channel 3 (ADC) | ADC | -10...10V |
| 8\*\* | External channel 4 (ADC) | ADC | -10...10V |
| 9\*\* | External channel 5 (ADC) | ADC | -10...10V |
| 10\*\* | External channel 6 (ADC) | ADC | -10...10V |
| 11 | internal monitor of task execution | internal triggers | bit |
| 12\*\*\* | internal monitor of Byte A | digital input | byte |
| 13\*\*\* | monitor of Byte B | digital output | byte |
| 14\*\*\* | monitor of Byte C (=multiplexer control byte) | ditital output | byte |

\*The measured value is the output before attenuation

\*\*Only availlable when external ADC’s (RA8GA) are availlable

\*\*\*Only availlable in 4C RZ6

The ‘Daq’ task can start and stop individual channels or groups of channels. The channels are selected by means of the parameter <channelselection>. When acquisition at a channel is stopped a flag ‘ACQ<x>\_Ready’ is set high. The value can be read by calling:

|  |
| --- |
| RZ6.GetTagVal('ACQ<x>\_Ready'); |

For every channels a sample rate divisor can be set by means of the parameter <divisor>. The divisor divides the base sample rate (e.g. 48.8 kHz) of the RZ6. The consequence is that less data points are stored in the buffer and since the buffer has a fixed size, the acquisition time before the buffer rewrites itself is therefore larger. Starting an stopping a channel is done by assigning ‘Start’ or ‘Stop’ to the parameter <startstop>. The parameter <divisor> is optional and the default value is 1.

|  |
| --- |
| tasktype = 'Daq'; %start data-acquisition  chanlist = [3 4 10]; %select channels 3&4&10  divisor = 100; %store one sample per 100 cycles  startstop = 'Start'; %start acquisition  tl.add\_task(0.000, tasktype, startstop, chanlist, divisor); |

It is possible to start acquisition with different divisors for different channels,

tl.add\_task(0.000,'Daq','Start',[1 2],10 );%start channels 1&2 @4.88 kHz

tl.add\_task(0.000,'Daq','Start',[3 4],488);%start channels 3&4 @100 Hz

and stop them in a different grouping at different times.

tl.add\_task(0.100, 'Daq', 'Stop', [1 4]); %stop channels 1&4

tl.add\_task(0.200, 'Daq', 'Stop', [2 3]); %stop channels 2&3

* When a channel is started for a second time the corresponding buffer is reset and previous acquired data for that channel is overwritten.
* The buffer size per channel is 1e6 data points.
* The maximum acquisition time is determined by the buffer size, the base sample rate and the divisor (divisor\*buffersize/basesamplerate). With a divisor of 1 and 48.8 kHz sample rate this is 20.5 seconds.
* When a channel is acquiring longer then the maximum time the buffer will start to overwrite the previous acquired data from the beginning.
* When a longer acquisition time is needed at a certain sampling rate multiple channels can be used after each other.
* If you run the internal monitor of task execution (channel 11) with a low sample rate (divisor >> 1), it is possible that triggers that are close together will fuse to one signal in the output stream and the last trigger can be lost.

**N.B. The ‘Daq’ task is not availlable in the 1 Core and Ripple versions of the RCX file.**

### Task type ‘SetDIO’ (digital output)

The task ‘SetDIO’ sets the bits of byte B high or low by means PAR1. Any output bit that is set high will stay high until it is set low again. When a trigger pulse is generated at a channel that is set high, the bit will go low for one RZ6 sample cycle (=20 µsec).

When the patch panel PP\_RZ6\_Digital-I/O is used the output bits are connected to the following hardware:

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'SetDIO'; %set bits of output B  outputbyte = bin2dec('0000 0001'); %set bit 0 high and others low;  tl.add\_task(delay, tasktype, outputbyte); |

### Task type ‘TrigOut’ (output triggers)

The trigger task can produce single and double triggers at output B. The parameter <outputbyte> specifies the bits that determine at which output channels the triggers are given.

A single trigger is a pulse with the length of two RZ6 cycles (=41 µsec @ 48828 Hz). When the parameter <delaytime> is zero, the output is a single trigger.

A double trigger are two pulses of 20 µsec with a delay in between. The delay is specified by parameter <delaytime> and measures the time delay of the rising edges. The smallest delay between the pulses is two RZ6 samples cycles (=41 µsec @ 48828 Hz), and the delay should be given in multiples of the RZ6 sample cycle length (=20.48 µsec @ 48828 Hz) when the highest precision is needed.

When delays between pulses are longer than a few milliseconds it is also possible to use the trigger task twice with each producing a single trigger pulse. However, the double trigger task is always the more accurate one with respect to timing.

N.B. the delay time specified for a double trigger has no influence on the delay(s) of the next task(s).

When the patch panel PP\_RZ6\_Digital-I/O is used the output bits are connected to the following hardware:

* Bit 0 = B0 (RBOXRZ6 LED yellow)
* Bit 1 = B1 (RBOXRZ6 LED green)
* Bit 2 = B2 (RBOXRZ6 LED green)
* Bit 3 = B3 (RBOXRZ6 LED red)
* Bit 4 = B4 (PP RZ6 Digital-I/O BNC)
* Bit 5 = B5 (PP RZ6 Digital-I/O BNC)
* Bit 6 = B6 (PP RZ6 Digital-I/O BNC)
* Bit 7 = B7 (PP RZ6 Digital-I/O BNC)

Single trigger example:

delay = 0.000; %delay in seconds

tasktype = 'TrigOut'; %set trigger output channels

outputbyte = bin2dec('0000 0010'); %selects a bit for output

tl.add\_task(delay, tasktype, outputbyte);

Double trigger example:

|  |
| --- |
| delay = 0.0; %delay in seconds  tasktype = 'TrigOut'; %set trigger output channels  outputbyte = bin2dec('0000 0101'); %selects two bits for output  doubledelay = 0.0001; %timing second pulse in seconds  tl.add\_task(delay, tasktype, outputbyte, doubledelay); |

### Task type ‘MultiConfigA’ or ‘MultiConfigB’

A sound consisting of a stack of pure tones can be generated by the sound task when using the sound type ‘MultiTone’. However, the tones must be configured in advance by the ‘MultiConfigA’ or ‘MultiConfigB’ task. For the configuration of each pure tone you need to execute a configuration task separately. The configurations are set separately for channel A and B. The multitone consists of up to four tones (but only three in the single DSP version and two in the RIP version). The parameter <index> identifies the pure tone in the stack. A configuration is further defined by the parameters <frequency>, <phase> and <amplitude>. By default, all the amplitudes are set to zero. The parameter <phase> specifies the phase of a tone that is set at the start of the sound.

When the tones are configured, the ‘MultiTone’ sound can be used multiple times within the trial.

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'MultiConfigA'; %configure a multi sinewave sound at Ch.A  index = 1; %index of sinewave (1...4)  frequency = 500; %frequency of sinewave  phase = 90; %phase of sinewave at start (-180...180)  amplitude = 0.1; %amplitude of sinewave (0..1) in Volt  tl.add\_task(delay, tasktype, index, frequency, phase, amplitude); |

N.B. The sum of the amplitudes of the configured tones should not exceed 10. The standard amplitude for sounds is 1. This corresponds to 1V output.

### Task type ‘Att’ (attenuation control)

The ‘Att’ task will be used before a sound is created to set the attenuation for the channels A and B.

In the RZ6 hardware there are three fixed attenuation circuits that cover each 20 dB. Therefore there are four ranges: 0…19.99, 20…39.99, 40…59.99 and 60…79.99. Within each range the attenuation is performed digitally.

When the attenuation is changing from one range to another a short spike will be generated and create audible clicks. Therefore, in order to avoid these clicks, it is necessary to perform the AttAB task with all the MUX channels closed. The opening of a MUX channel can be timed at the same time as the attenuation task as long as the attenuation task is listed before the MUX task.

By default the scale of a simple sound (TONE, NOISE or Sweep) produced by the processor is around 1V in amplitude before attenuation. The corresponding bit-depth is between 17 and 21 bits (depending on the attenuation).

A scale factor can be used to pre-scale the sound that is fed to the attenuation circuits. A caveat is that the bit-depth of the sound will be deminished by one bit for every factor of 2 that the amplitude is deminished. Be aware that a scalefactor larger than one can result in clipping.

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'Att'; %set the attenuation for Ch.A  att1 = 22.5; %attenuation in dB (0..80)  sf1 = 0.1; %pre-attenuation scalefactor  att2 = 10; %attenuation in dB (0..80)  sf2 = 1; %pre-attenuation scalefactor  tl.add\_task(delay, tasktype, att1, att2, sf1, sf2); |

### Task type ‘ITD’

The ‘ITD’ task can be used for introducing small timing delays for the sounds played on both the A and B channel. The delay comes on top of the starting delay of the task. With this option you can introduce delays as small as on clock tick of the RZ6 (20 µsec) up to 10 ms.

|  |
| --- |
| delay = 0.000; %starting delay of task in seconds  tasktype = 'ITD'; %Task Interaural Time Difference  delay-A = 0.00005; %delay channel1 (0..0.010 sec)  delay-B = 0.0; %delay channel2 (0..0.010 sec)  tl.add\_task(delay, tasktype, delay-A, delay-B); |

N.B. When a Noise type is used the sound is filtered with a bandpass filter for each channel. The bandpass filters both give a group delay depending on the values for the LP cut-off and the HP cut-off frequencies. Since the delays are both depending on the sound frequencies and the cut-off frequencies, it is impossible to synchronize the sounds when different bandpass filter settings are used for channel A (Left) and channel B (right). The bandpassfilters give only synchronized sounds when the bandpass filter settings are the same for both channels and when task type ‘SoundAB’ is used.

The group delays have been experimentally dertermined (for sine wave @ 2000 kHz) and can be estimated by:

LP (for flp > 3 x f): delay = (-30 + 19.5 x f0 / flp)[µsec]

HP (for fhp < f / 3): delay = -0.05 x fhp [µsec]

where f0 is the clock rate of the RZ6.

### Task type ‘Mix’ (mix sounds)

The ‘Mix’ task can be used for mixing two different sounds to one output. The task has the options ‘Stop’, ‘AtoB’, ‘BtoA’ and ‘Mixed’. The ‘Stop’ options stop the mixing (but not the sounds). The ‘AtoB’ option mixes the A and B sounds and plays it at Out-B and no sound at Out-A. The ‘BtoA’ option the sounds and plays at Out-A and no sound at Out-B. The ‘Mixed’ option plays the AtoB mix at Out-B and the BtoA mix at Out-A. The parameter <par1> determines the amount of mixing that takes place.

If the task ‘AtoB’ is used with mixfactor = 0.5 then soundA is scaled by a factor of 0.5 and added to soundB. So the output of ChannelA is A+0.5B and the output of ChannelB = 0.

When ‘BtoA is used with mixfactor = 0.5 the output of ChannelA is 0 and the output of ChannelB = B+0.5A.

When ‘Mixed’ is used with mixfactor = 0.5 the output of ChannelA is A+0.5B and the output of ChannelB = B+0.5A.

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'Mix'; %mix soundA and soundB  mixtype = 'AtoB'; %mix x\*A+B to Out-B  mixfactor = 0.5; %mix factor x (0..1)  tl.add\_task(delay, tasktype, mixtype, mixfactor); |

N.B. The ‘Mix’ task also be used in conjuction with the ‘SoundMov’ task. In that case only the sound mixed to A will be played, so mixtype ‘BtoA’ need to be used in this case.

### Task type ‘Reset’ (reset task list)

The ‘Reset’ task will reset the trial (=task list) so that the first task in the list will be the next task that is executed. The tasks that are listed after the reset trial task will be ignored. The use of the reset trial task will cause the RCX program to loop indefinitely or until a new task list is uploaded.

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'Reset'; %reset the trial  tl.add\_task(delay, task type); |

N.B. A reset of the task list can also be forced by sending a software trigger in MATLAB the code (rz6.trigger(4)) or by pushing the ‘S4’ button in the RPvdsEx program).

### Task type ‘Ready’ (set ready flag)

The ‘Ready’ task will set a flag (‘STM\_Ready’) high. Task that are listed after the trial is ready task will be ignored. The flag can only be reset by using the zBusA trigger that resets the trial (=task list).

|  |
| --- |
| delay = 0.000; %delay in seconds  tasktype = 'Ready'; %flag the trial as ready  tl.add\_task(delay, task type); |

## Matlab files

The following matlab files are needed in your file path:

|  |  |
| --- | --- |
| **biox\_rz6\_xxxxxx.m\*** | class for creating rz6 client |
| **biox\_rz6\_tasklist.m** | class for creating task list |
| **biox\_zbus\_client.m** | class for creating zbus client |
| **biox\_rz6\_client.m** |  |
| **biox\_abstract\_client.m** |  |
| **biox\_inputParser.m** |  |
| **dispvarargin.m** |  |

\*xxxxxx specifies the setup

# The functionality of the RCX file

## Introduction

In order to perform an experiment, the user will have to design a program in MATLAB that manages the experiment. An experiment is divided into trials and each trial is separately uploaded from the experimental computer to the RZ6 and executed on the RZ6 mostly without further intervention of the MATLAB program. A trial is specified in the MATLAB code by a matrix that contains all the instructions for the trial. A trial can contain many (up to 1000) separate tasks.

## Tasks

There are basically three kinds of tasks: 1) a waiting task, 2) a do something task and 3) a configuration task.

1. A waiting tasks will wait for an indefinite amount of time until an event occurs. An event can be a user pushing a button or some other input signal, a trigger on the ZBusB or a Soft trigger from the computer.
2. A do something task is a single task that can be the setting of a flag or output, sending an output trigger, start or stop a sound or whatever task that can be performed in a single moment.
3. A configuration task is used to pre-configure a multi tone stack.

## Timing of tasks

Tasks are executed in the same order as they appear in the matrix, by column index. The timing (delay time) of a task is specified with respect to the end of the last waiting task or with respect to start-up or reset moment of the RCX file. You can for instance start with a task waiting until a ZBus trigger is detected, then start a sound 500 ms after the waiting task has ended (that is the moment the ZBus trigger was detected) and stop the sound 1000 ms after the waiting task has ended. So all the tasks following a certain waiting task form together a block. Within such a block all the tasks have the same timing reference (T=0). Note the following:

* The timing of a new waiting task is also specified with respect to end of the previous waiting task.
* If a waiting task is programmed with a certain delay time, the waiting task will only react on an event (trigger) that occurs after the delay time has expired.
* The timing reference of the first task(s) is 100 ms after start-up of the program.
* A waiting task directly (with no delay) programmed after another waiting task will be ignored.

As said before, tasks are always executed in the order of appearance in the matrix. So if a task is specified having a timing before the timing of a previous task, the task will be performed right after that previous task, thus being later than the timing specification intended. **This has to be avoided!**

A start and stop task pair do not have to be within one block. A start task can be specified after some waiting task (e.g. in block 1) while the corresponding stop task can be specified after another waiting task (e.g. in block 2).

Figure 1 shows the timing of a simple trial example. After loading the RCX program the program waits for a ZbusB-trigger to start a sound and then wait for a press on a user button to stop the sound.



Figure 1 An example of a trial consisting of three blocks. Note that each block has its own reference time (T1, T2 and T3). Colours: Blue=trigger event; Green=waiting task; Orange=Some other tasks.

### Use event recorder for timestamping the tasks

An output signal can be used for timestamping the execution of tasks with the event recorder. An 8 bit variable ‘SGN\_Byte’ is used to set the signal bit mask. This will determine at to which output channel(s) the signal is sent. The signal bit mask can be set by the command: RZ6.SetTagVal('SGN\_Byte', <value>). By default, the mask is zero and no output signal is produced.

The output signal can also be used for audio or visual feedback, e.g. by connecting the signalling output to a LED or to a monitor speaker.

When the ‘PP RZ6 Digital I/O’ patch panel is used the outputs that are available are listed below.

***Available Outputs (RZ6 byte B):***

* Bit 0 = B0 (RBOXRZ6 LED left)
* Bit 1 = B1 (RBOXRZ6 LED mid-left)
* Bit 2 = B2 (RBOXRZ6 LED mid-right)
* Bit 3 = B3 (RBOXRZ6 LED right)
* Bit 4 = B4 (PP RZ6 Digital-I/O BNC B4)
* Bit 5 = B5 (PP RZ6 Digital-I/O BNC B5)
* Bit 6 = B6 (PP RZ6 Digital-I/O BNC B6)
* Bit 7 = B7 (PP RZ6 Digital-I/O BNC B7)

### Use data-acquisition for checking the task timing

A data-acquisition task can be used for checking the timing of the tasks that are executed. Channel 11 of the data-acquisition channels is dedicated to internally monitoring the triggers for task execution. The data-acquisition should run at the full sample rate of the RZ6. The result is a bit stream of zero’s and a single one for every task that is executed. If the RZ6 is running at 48.8 kHz this means a timing resolution of about 20 mircoseconds.

N.B. Be aware that a lot of data is created with this method.

## DAC and ADC channels.

### Introduction

The main purpose of the RZ6 is to play analogue signals on the DACs and record analogue signals from the ADCs. The RZ6 has two DAC channels (Out-A and Out-B) and two ADC channels (In-A and In-B). More ADC channels can be added with an external device. The ADC channels can record independent of each other.

### DAC channels: playing sounds

The DAC channels are used to generate sounds. The DAC signals for channels A and B can be controlled completely independent of each other. There are several pre-defined sound that can be played: white noise, single pure tones or stacks of pure tones, sweeps (whoop) of a pure tone or frequency modulated tones. A special version of the RCX file can produce a stack of pure tones amplitude modulated in frequency space as well as temporal, called ripple sound.

The B channel can be coupled to the A channel with the waveform and phase being identical for both channels. However, the amplitude can be manipulated in such a way that a virtual sound source is created somewhere between the two speakers. When the channels are coupled this way, the task ‘SOUND\_1’ specifies the sound for both speakers, the volumes can be set by the attenuation and the task ‘SOUND\_2’ specifies what the relationship is between the output volumes in time. With this option a virtual sound source is created that can be static or move back and forth between the two speakers.

* All sounds ramp up and ramp down with a squared cosine function with a rise and fall time of 10 ms in order to avoid transient ‘clicks’.
* When a SoundB task is use directly after a SoundA task with the same delay, then the Sounds are perfectly synchronized.
* A sound at channel A starts 12 clock cycles after the start of the task.
* Most sounds at channel B start 6 clock cycles after the start of the task
* A noise sound at channel B starts 12 clock cycles after the start of the task

### ADC channels: recording

The ADC channels are numbered 1 to 10. Channels 1 and 2 are the standard ADC channels of the RZ6 and are accessible via two BNC connectors on the outside labelled IN-A and IN-B. Channel 3 and channel 4 are internally recording the signals send to the DAC outputs A and B. The channels 5 to 10 are only available when an external ADC (e.g. the RA8GA) is connected to the RZ6.

The minimum Acquisition time for a channel is 1 ms. The maximum acquisition time is limited due to storage capacity. When the time between starting and stopping the acquisition exceeds the specified acquisition time, the latest data will overwrite earlier data. The maximum acquisition time is determined by the size of the buffer divided by the sample frequency. The maximum buffer size is set to 1e6 samples. The 1 DSP version is running at 25 kHz by default, the maximum acquisition time is therefore 40 seconds. The 3 DSP and 4 DSP versions run at 50 kHz.

It is possible to extend the acquisition time in two ways. One: by using the parameter ‘divisor’ to lower the sample frequency. Two: by reading out the buffer repetitively during acquisition. This second Method provides real continuous acquisition.

## Digital I/O

### Introduction

The RZ6 has 24 bits of digital I/O. The bits are available by three bytes labelled A, B and C. The I/O is normally connected via a serial cable to a patch panel with the name ‘PP RZ6 Digital-I/O’ (see C1 Patch panel (PP\_RZ6\_Digital-I/O)). Bytes A and C are configured as inputs; byte B is configured as output. Bits 0…3 of byte A and Bits 0…3 of byte B are connected from left to right to the buttons and LEDs of the button box respectively (see C2 Response box) .

Bits 4…7 of byte A and byte B are connected to eight BNC sockets on the patch panel. Byte C is via a D-sub connector on the patch panel connected to the PM2R multiplexer.

### Byte A, input signals, triggers

A signal detected by an input bit of byte A can be used as a trigger when a ‘wait for trigger’ task is active. Which bits are use is specified by a bit mask parameter of the ‘wait for trigger’ command. When the task hold input is used, the input bits that became high at some moment will be stored and can be read out until they are reset. This is useful if you want to read out a user response to one of the response buttons. A single pushbutton connected to one of the BNC inputs A4..A7 can also be used as a user input.

### Byte B, Output signals, triggers

The output signals of byte B can be used in three different ways. The digital output task and the trigger task have a parameter that specifies the output bits that are used with a bit mask.

The digital output task just sets some bits high. This can for instance be used for used feedback by lighting LEDs on the button box. When is signal is set high, it stays high until it is set low again.

* The trigger task can give single pulse or double pulse output triggers. A single trigger pulse has the duration of only one RZ6 sample cycle (=20 µsec). A double trigger is two pulses after each other with a short delay in between. The delay time of the second pulse can be as low as 40 µsec, or two RZ6 clock cycles.
* The global parameter SGN\_Byte specifies the bit(s) at which a single trigger pulse of one RZ6 sample cycle (=20 µsec) is generated at the moment that the RCX program starts executing a task. This feature is useful for timestamping each task with the event recorder.

The resulting output byte of the trigger task (2) and the signalling byte (3) are combined in an iOR statement. The resulting byte (2V3) is combined with the bitmask of the output task (1) in an iXOR statement. The result is that when a trigger is executed at a certain bit by means of a trigger task or as a timestamp for the event recorder and the bit was already set high by the digital output task, then the trigger will be inverted. Since most triggers are defined as the rising flank of a pulse the detection of the trigger by another device (e.g. the event recorder) will be delayed one RZ6 sample cycle (=20 µsec).

### Byte C, Multiplexer interface

Byte C is connected to and controls up to four multiplexers. The multiplexer byte is set by the parameters of the multiplexer tasks. Each multiplexer can have only one channel open at a time. When a new channel is opened, the channel that was open before will be closed automatically. The reset command closes any open channel.

## DAC offset compensation

Each DAC has an offset that is mostly in the order of 0-10mV. Sometimes this is enough to generate audible clicks in a speaker or headset when changing the MUX channel. In order to avoid these clicks the offset has to be as low as possible. The program has the possibility to compensate the offset by setting the following parameters:

<OFS\_DAC-A1>: DAC channel A, attenuation range from 0dB up to 20dB

<OFS\_DAC-A2>: DAC channel A, attenuation range from 20dB up to 40dB

<OFS\_DAC-A3>: DAC channel A, attenuation range between 40dB and 60dB

<OFS\_DAC-B1>: DAC channel B, attenuation range from 0dB up to 20dB

<OFS\_DAC-B2>: DAC channel B, attenuation range from 20dB and 40dB

<OFS\_DAC-B3>: DAC channel B, attenuation range between 40dB and 60dB

Although the attenuation has also a separate circuit for the range of 60dB to 80 dB, a correction of an average offset will sent the output out of range. Therefore the correction is left out.

These offsets have to be measured for each device, each channel and each range individually.

The parameters will be written to the RZ6 by the constructor of the lab specific RZ6 object.

N.B. The manual attenuation (knob on front of RZ6) adds to the programmed attenuation in a linear way. However, change in the manual attenuation does not change the offset.

N.B. Changing the attenuation setting when a MUX channel is open channel can result in an audible click when the attenuation crosses the 20dB, 40dB or 60dB boundary. This is also the case when directly listening at one of the DAC outputs (without MUX).

## ITD experiments

Experiments with interaural time differences (ITD’s) can be performed with internally generated noise or by the use of WAV data.

## ITD with ‘noise’ sound

The ‘noise’ sound task can has an optional third parameter ‘ComSrc’ which means that the noise generated in channel B uses the same generator as channel A. The frequency band can still be set to different values. A fourth parameter determines the ITD. The ITD is given in milliseconds and is always positive. When channel A has an ITD the sound in channel A is delayed with respect to channel B. When channel B has an ITD it is the other way around. The smallest ITD that can be used is equivalent to one clockcycle. At 48.8 kHz that corresponds to 20 microseconds.

## ITD with WAV

When WAV data is used for ITD experiments, small must be accounted for in the data. When two identical data sets are uploaded for both channels both the WAVs will play exactly synchonous when started with the same specified delay. The smallest difference that can be used by starting the WAV sounds after each other is about 1 ms. Below that the delay is not well defined.

# Appendix A: Debugging info

## A1 Introduction

The task list is sent to the RZ6 in the form of a matrix. Each column in this matrix represents a single task. Each task has 7 parameters (rows) that together specify the task. The meaning of the first three parameters is fixed. The meaning of the 4th to 7th parameter is depending on the context. The information can be used for debugging purposes in combination with the debug option of the task list object.

When the words ‘not used’ are put in the right column, the corresponding value in the matrix is zero.

## A2 ‘Waitfortrigger’

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 0 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | *Not used* |  |  |
| PAR1 | Trigger type mask\* | byte | 0..63 |
| PAR2 | External trigger bit mask | byte | 0..255 |
| PAR3 | *Not used* |  |  |
| PAR4 | *Not used* |  |  |

\*the trigger type mask is the sum of the following trigger type ID’s:

|  |  |  |
| --- | --- | --- |
| **TYPE** | **ID** | **Description** |
| ‘ZbusA’ | 1 | Incoming zbus trigger A |
| ‘ZbusB’ | 2 | Incoming zbus trigger B |
| ‘External’ | 4 | Incoming trigger at input determined by the External trigger bit mask |
| ‘Soft1’ | 8 | Incoming software trigger 1 |
| ‘Soft2’ | 16 | Incoming software trigger 2 |

## A3 ‘SoundA’, ‘SoundB’ or ‘SoundAB’

‘SoundA’ = 1, ‘SoundB’ = 2 ‘SoundAB’ = 16

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 1 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | IDENTIFIER | integer | 0..6 |
| PAR1 | Sound type dependent |  |  |
| PAR2 | Sound type dependent |  |  |
| PAR3 | Sound type dependent |  |  |
| PAR4 | Sound type dependent |  |  |

‘SoundB’

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 2 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | IDENTIFIER | integer | 0..7 |
| PAR1 | Sound type dependent |  |  |
| PAR2 | Sound type dependent |  |  |
| PAR3 | Sound type dependent |  |  |
| PAR4 | Sound type dependent |  |  |

‘SoundAB’

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 16 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | IDENTIFIER | integer | 0..6 |
| PAR1 | Sound type dependent |  |  |
| PAR2 | Sound type dependent |  |  |
| PAR3 | Sound type dependent |  |  |
| PAR4 | Sound type dependent |  |  |

Sound type options:

|  |  |  |
| --- | --- | --- |
| **Name** | **ID** | **Description** |
| ‘Stop’ | 0 | Stop sound |
| ‘Tone’ | 1 | Pure or modulated tone |
| ‘Sweep’ | 2 | Frequency sweep (whoop) |
| ‘Noise’ | 3 | Gaussian noise |
| ‘Ripple’ | 4 | Ripple sound |
| ‘WAV’ | 5 | Array of floating point data @ 48 kHz |
| ‘MultiTone’ | 6 | Sum of up to four pure tones |
| ‘B=A’ | 7 | A and B have the same sound |

## A4 ‘MUX’

For the MUX byte (RZ6 byte C) the following bit assignments are used:

* Bit 0-3 = Channel Number 0...15
* Bit 4-5 = Device Number 0...3
* Bit 6 = Set
* Bit 7 = Reset

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 3 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | *Not used* |  |  |
| PAR1 | MUX byte | byte | 0..255 |
| PAR2 | *Not used* |  |  |
| PAR3 | *Not used* |  |  |
| PAR4 | *Not used* |  |  |

## A5 ‘HoldInput’

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 4 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | *Not used* |  |  |
| PAR1 | Bit mask for byte A | byte | 255 |
| PAR2 | *Not used* |  |  |
| PAR3 | *Not used* |  |  |
| PAR4 | *Not used* |  |  |

## A6 ‘Daq’

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 6 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | *Not used* |  |  |
| PAR1 | Start = 1; Stop = 0 | integer | 0..1 |
| PAR2 | Acquisition channels (10 bits) | integer | 0..1023 |
| PAR3 | Sample rate divisor | integer | 1..∞ |
| PAR4 | *Not used* |  |  |

## A7 ‘DIOout’

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 7 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | *Not used* |  |  |
| PAR1 | byte B | byte | 0..255 |
| PAR2 | *Not used* |  |  |
| PAR3 | *Not used* |  |  |
| PAR4 | *Not used* |  |  |

## A8 ‘TRGout’

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 8 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | *Not used* |  |  |
| PAR1 | byte B | byte | 0..255 |
| PAR2 | Delay time [s] | float | 0..10e7 |
| PAR3 | *Not used* |  |  |
| PAR4 | *Not used* |  |  |

## A9 ‘ConfigureA’ or ‘ConfigureB’

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | A=11; B=12 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND TYPE IDENTIFIER | *Not used* |  |  |
| PAR1 | Tone index | integer | 1..4 |
| PAR2 | Frequency [Hz] | float | 0..24400 |
| PAR3 | Phase [degrees] (float =) | float | -180….180 |
| PAR4 | Amplitude | float | 0..10 |

## A10 ‘Reset’

|  |  |  |  |
| --- | --- | --- | --- |
| TASK TYPE | IDENTIFIER | integer | 9 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | *Not used* |  |  |
| PAR1 | *Not used* |  |  |
| PAR2 | *Not used* |  |  |
| PAR3 | *Not used* |  |  |
| PAR4 | *Not used* |  |  |

## A11 ‘Ready’

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 10 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | *Not used* |  |  |
| PAR1 | *Not used* |  |  |
| PAR2 | *Not used* |  |  |
| PAR3 | *Not used* |  |  |
| PAR4 | *Not used* |  |  |

## A12 ‘Att’

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 13 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | *Not used* |  |  |
| PAR1 | Attenuation A [dB] | float | 0..80 |
| PAR2 | Attenuation B [dB] | float | 0..80 |
| PAR3 | Scaling factor A [ ] | integer | 0..10 |
| PAR4 | Scaling factor B[ ] | integer | 0..10 |

## A13 ‘ITD’

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 14 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | *Not used* |  |  |
| PAR1 | delay A [s] | float | 0..0.01 |
| PAR2 | delay B [s] | float | 0..0.01 |
| PAR3 | *Not used* |  |  |
| PAR4 | *Not used* |  |  |

## A14 ‘MIX’

Par1: Stop=0, AtoB=1, BtoA=2 and Mixed=3

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 15 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND TYPE IDENTIFIER | *Not used* |  |  |
| PAR1 | Mixing type | integer | 0..3 |
| PAR2 | Mixfactor | float | 0..1 |
| PAR3 | *Not used* |  |  |
| PAR4 | *Not used* |  |  |

## A15 ‘SoundMov’

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 5 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | *Not used* |  |  |
| PAR1 | Start = 1; Stop = 0 | integer | 0 or 1 |
| PAR2 | Number of speakers in array |  |  |
| PAR3 | Period of movement [s] |  |  |
| PAR4 | Starting phase [degrees] (-180…180) |  |  |

## A16 ‘Stop’

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 1 or 2 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | IDENTIFIER | integer | 0 |
| PAR1 | *Not used* |  |  |
| PAR2 | *Not used* |  |  |
| PAR3 | *Not used* |  |  |
| PAR4 | *Not used* |  |  |

## A17 ‘Tone’

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 1 or 2 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | IDENTIFIER | integer | 1 |
| PAR1 | Tone or centre frequency [Hz] |  |  |
| PAR2 | Modulation frequency [Hz] |  |  |
| PAR3 | Modulation band width [Hz] |  |  |
| PAR4 | *Not used* |  |  |

## A18 ‘Sweep’

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 1 or 2 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | IDENTIFIER | integer | 2 |
| PAR1 | Base frequency [Hz] | float | 1..24400 |
| PAR2 | Number of octaves | integer | 1.. |
| PAR3 | Period of complete sweep [s] | float | 0..∞ |
| PAR4 | *Not used* |  |  |

## A19 ‘Noise’

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 1 or 2 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | IDENTIFIER | integer | 3 |
| PAR1 | Low pass frequency [Hz] | float | 1..24400 |
| PAR2 | High pass frequency [Hz] | float | 1..24400 |
| PAR3 | *Not used* |  |  |
| PAR4 | *Not used* |  |  |

## A12 ‘Ripple’

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 1 or 2 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | IDENTIFIER | integer | 4 |
| PAR1 | Start frequency [Hz] | float | 1..24400 |
| PAR2 | Modulation in time [Hz] | float | 0..24400 |
| PAR3 | Modulation in frequency (Phase/Octave) | float |  |
| PAR4 | Attenuation [dB] | float | 0..80 |

## A21 ‘WAV’

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 1 or 2 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | IDENTIFIER | integer | 5 |
| PAR1 | Continue = 0 or Reset = 1 | integer | 0 or 1 |
| PAR2 | *Not used* |  |  |
| PAR3 | *Not used* |  |  |
| PAR4 | *Not used* |  |  |

## A22 ‘MultiTone’

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 1 or 2 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | IDENTIFIER | integer | 6 |
| PAR1 | *Not used* |  |  |
| PAR2 | *Not used* |  |  |
| PAR3 | *Not used* |  |  |
| PAR4 | *Not used* |  |  |

## A23 ‘B=A’

|  |  |  |  |
| --- | --- | --- | --- |
| TASK | IDENTIFIER | integer | 2 |
| DELAY | Delay [s] | float | 0..81920 |
| SOUND | IDENTIFIER | integer | 7 |
| PAR1 | ‘Fixed’ = 1, ’Sine’ = 2, ’Linear’ = 3 | integer | 1..3 |
| PAR2 | Period [s] | float | 0..∞ |
| PAR3 | Phase | float | -180 … 180 |
| PAR4 | *Not used* |  |  |

# Appendix B: Matlab classes

The following classes are defined in the BIOX common files library:

|  |  |
| --- | --- |
| **Class** | **Parent class** |
| biox\_abstract\_client |  |
| biox\_rz6\_client | biox\_abstract\_client |
| biox\_rz6\_1c | biox\_rz6\_client |
| biox\_rz6\_3c | biox\_rz6\_client |
| biox\_rz6\_4c | biox\_rz6\_client |
| biox\_rz6\_4c\_mov | biox\_rz6\_client |
| biox\_rz6\_4c\_ripple | biox\_rz6\_client |
| biox\_rz6\_tasklist | handle |
| biox\_inputparser | inputparser |
| rz6\_dummy | handle |
| zbus\_dummy | handle |
| biox\_EEG\_NIRS | biox\_rz6\_3c |
| biox\_SPK\_ARM | biox\_rz6\_4 |
| biox\_VST\_CHR | biox\_rz6\_4c\_mov |
| biox\_PAT\_LAB | biox\_rz6\_3c |

## B1 biox\_abstract\_client

The biox\_abstract\_client class does the following:

* scaling of ADC input (protected)
* write to RZ6 (abstract)
* read from RZ6 (abstract)
* soft triggers for RZ6 (abstract)
* all other read and write functions (including write\_tasklist)

## B2 biox\_rz6\_client

The biox\_rz6\_client class does the following:

* implementation of all abstract functions of biox\_abstract\_client
* creates an activeX control for the RZ6
* loads and starts the RCX-circuit to the RZ6

## B3 biox\_rz6\_<x>c (and others)

The biox\_rz6\_3c class does the following:

* Is used for RZ6 with a specific number <x> of DSP cores.
* Knows which RCX file has to be used

## B4 biox\_PAT\_LAB (and others)

The biox\_PAT\_LAB class (and others) does the following:

* Is used for a specific lab (Patient Lab)
* Stores the DAC offsets for the specific RZ6 used in the Patient Lab

# Appendix C: Hardware accessories

## C1 Patch panel (PP\_RZ6\_Digital-I/O)

The patch panel is a break-out panel for the digital I/O output port of the RZ6.



Figure 2 Patch Panel

The digital I/O output port is connected with a D-Sub-25 connector cable to the Bytes A,B,C input of the patch panel. The bytes A, B and C are designated as follows

* Byte A: digital inputs
* Byte B: digital outputs
* Byte C: digital outputs

|  |  |
| --- | --- |
| **Byte A** |  |
| A0 | RBOXRZ6 Button left |
| A1 | RBOXRZ6 Button left middle |
| A2 | RBOXRZ6 Button right middle |
| A3 | RBOXRZ6 Button right |
| A4 | PP RZ6 Digital-I/O BNC |
| A5 | PP RZ6 Digital-I/O BNC |
| A6 | PP RZ6 Digital-I/O BNC |
| A7 | PP RZ6 Digital-I/O BNC |

|  |  |
| --- | --- |
| **Byte B** |  |
| B0 | RBOXRZ6 LED left |
| B1 | RBOXRZ6 LED left middle |
| B2 | RBOXRZ6 LED right middle |
| B3 | RBOXRZ6 LED right |
| B4 | PP RZ6 Digital-I/O BNC |
| B5 | PP RZ6 Digital-I/O BNC |
| B6 | PP RZ6 Digital-I/O BNC |
| B7 | PP RZ6 Digital-I/O BNC |

|  |  |
| --- | --- |
| **Byte C** |  |
| C0 | MUX Channel bit 0 |
| C1 | MUX Channel bit 1 |
| C2 | MUX Channel bit 2 |
| C3 | MUX Channel bit 3 |
| C4 | MUX ID bit 0 |
| C5 | MUX ID bit 0 |
| C6 | MUX set |
| C7 | MUX reset |

## C2 Response box

A0..A3 and B0..B3 are connected to the RBOXRZ6 response box. A4..A7 and B4..B7 are available via the BNC connectors on the patch panel. When the command RZ6.write\_buttonbox\_echo(true) is given, the LEDs will echo the button input signals.



Figure 3 Response Box

## C3 RA8GA Adjustable Gain Preamp

An external ADC can be used with the RZ6. In some cases this will be the RA8GA Adjustable Gain Preamp. When used in combination with the custom ADC inputs patch panel (see figure) six of the eight analog inputs are available with BNC connectors. The BNC connectors are numbered 1..6 and they correspond to the data acquisition channels 5..10.



Figure 4 RA8GA with custom patch panel

When the setup uses magnetic field coils, the BNC inputs 1, 2 and 3 are used for H, V and F fields.

The ‘To Base’ optical in and output should be connected to the ‘Optical In’ of the RZ6. The RA8GA needs a Zbus caddy with FO5 optical bus in order to function.

## C4 Response button

A response button can be used with digital input A4, A5, A6 or A7. The button does not need any battery. The button should be configurated ‘normally open’.



Figure 5 Example of a response button

# Appendix D: Examples

## D1 Simple Trial

This MATLAB example starts with creating a rz6 and task list object, followed by writing the signal byte. Then it assigns values to the parameters used in the tasks. Several tasks are defined and added to the task list object <tl>. At last the task list object is written to the RZ6.

The task list start with sending a trigger to a led controller. This switches a LED on. Then it waits for the subject to press a button, which is registered as an external input trigger at input A4. As soon as the button press is registered a second trigger is send to the LED controller in order to switch the LED off. Right after that the attenuation for channel A is set and the mux is instructed to open a speaker channel. After 100 ms delay (measured from the button press) a noise sound is started at channel A. After 1000 ms delay (measured from the button press) the sound is stopped. 100 ms after that the mux is instructed to close the speaker channel and the trial is set ready.

|  |
| --- |
| %create rz6 and tl objects  rz6 = biox\_PAT\_LAB; %create rz6 object  tl = biox\_rz6\_tasklist; %create task list  %set global parameters  signalbyte = bin2dec('0100 0000'); %signal @ B6  rz6.write\_signalbyte(signalbyte); %write signal byte  %declare task parameters and assign values  trgoutbyte = bin2dec('1000 0000'); %trg to LEDs @B7  trginbyte = bin2dec('0001 0000'); %trg from button @A5  attenuation = 5.0; %attenuation 20dB  channel = 10; %speaker channel 10  mux = 2; %multiplexer 2  lpf = 3000; %noise low pass filter  hpf = 200; %noise high pass filter  %add tasks to list  %timing block  tl.add\_task(0.000,'AttA', attenuation);  tl.add\_task(0.000,'TrigOut', trgoutbyte);  tl.add\_task(0.000,'WaitForTrigger','External', trginbyte);  %timing block  tl.add\_task(0.000,'TrigOut', trgoutbyte);  tl.add\_task(0.000,'MUX', mux, 'set', channel);  tl.add\_task(0.100,'SoundA','noise', hpf, lpf);  tl.add\_task(1.000, 'SoundA', 'stop');  tl.add\_task(1.100,'MUX', mux, 'Reset');  tl.add\_task(1.100,'Ready');  %add tasks to list  rz6.write\_tasklist(tl); |

## D2 Moving sound and reaction time

In this example the trial will wait for a button press by the subject. The button press starts a moving noise sound, beginning in the middle and moving right or left, and the subject has to press the right or left button of the response box, that corresponds to the direction of the sound. After the response button is pressed the trial is finished and the response button and response time can be read from the RZ6.

The middle buttons of the response box are indicated by '0000 0110', the outer buttons by '0000 1001'.

|  |
| --- |
| %create rz6 and tl objects  rz6 = biox\_PAT\_LAB; %create rz6 object  tl=biox\_rz6\_tasklist; %create task list    %set global parameters  signalbyte = bin2dec('0100 0000'); %signal @ B6  rz6.write\_signalbyte(signalbyte); %write signal byte    %declare task parameters and assign values  trgoutbyte = bin2dec('1000 0000'); %trigger out @ B7  trginmask = bin2dec('0000 0110'); %trigger in @ A2 or A3  responsemask = bin2dec('0000 1001'); %response @ A1 or A4  AttA = 7.5; %attenuation 7.5dB  AttB = AttA; %attenuation 7.5dB  lpf = 3000; %noise low pass filter  hpf = 200; %noise high pass filter  NumSpeakers = 21; %Use max nr of speakers  Period = 5000; %Period in milliseconds  Phase = 0; %start middle mov right  %Phase = 180; %start middle mov left    %add tasks to list  %timing block  tl.add\_task(0.000,'AttA', AttA);  tl.add\_task(0.000,'AttB', AttB);  tl.add\_task(0.000,'WaitForTrigger','External', trginmask);  %timing block  tl.add\_task(1.000,'HoldInp', responsemask);  tl.add\_task(1.000,'SoundA','noise', hpf, lpf);  tl.add\_task(1.000,'SoundMOV','Start', NumSpeakers, Period, Phase);  tl.add\_task(1.000,'WaitForTrigger','External', responsemask);  %timing block  tl.add\_task(0.000,'SoundMOV','Stop');  tl.add\_task(0.000,'Ready');    %add tasks to list  rz6.write\_tasklist(tl);    %wait for the task to be ready  while ~rz6.read\_trialready()  % wait  end    responsebutton = rz6.read\_inputholdbyte() %get the response button  reactiontime = rz6.read\_responsetime() %get the response time    %clean up  delete(tl);  delete(rz6); |

## D3 Recording head movement and playing ‘WAV’-file

In this example is playing an uploaded sound (WAV) from a virtual sound source in between two speakers. The sound and the X,Y,Z movement of the head is recorded for 3 seconds with a sample rate of 5 kHz. A led is on during the recording.

|  |
| --- |
| %create rz6 and tl objects  rz6 = biox\_PAT\_LAB; %create rz6 object  tl = biox\_rz6\_tasklist; %create task list  tl.debug(true);  %set global parameters  signalbyte = bin2dec('0100 0000'); %signal @ B6  rz6.write\_signalbyte(signalbyte); %write signal byte    chanlist = 1; %write only to ch1 (=chA)  fn = which('mysound.wav');  afr = dsp.AudioFileReader(fn);  WAVdata = step(afr)'; %read the first 1024 samples  rz6.write\_wavdata(WAVdata(1,:), chanlist); %write wav data to RZ6    %declare task parameters and assign values  deviceA = 0;  deviceB = 1;  channelA = 5;  channelB = 13;  trgoutbyte = bin2dec('1000 0000'); %led trigger @ B7  trginmask = bin2dec('0001 0000'); %trigger in @ A4  indicator = bin2dec('0100 0000'); %indicator @ B6  daqXYZ = bin2dec('00 0111 0000'); %select daq chans 5+6+7  daqSND = bin2dec('00 0000 0100'); %select daq chans 3  off = bin2dec('0000 0000'); %off  divXYZ = 100; %store one per 100 cycles  divSND = 1; %store one per 1 cycle    %add tasks to list  %timing block  tl.add\_task(0.000, 'WaitForTrigger', 'External', trginmask);  %timing block  tl.add\_task(0.500, 'MUX', deviceA, 'Set', channelA);  tl.add\_task(0.500, 'MUX', deviceB, 'Set', channelB);  tl.add\_task(1.000, 'Daq', 'Start', daqXYZ, divXYZ);  tl.add\_task(1.000, 'Daq', 'Start', daqSND, divSND);  tl.add\_task(1.000, 'SetDIO', indicator);  tl.add\_task(1.100, 'SoundA', 'WAV', 'Reset');  tl.add\_task(1.100, 'SoundB', 'B=A', 'Fixed');  tl.add\_task(2.000, 'SoundA', 'Stop');  tl.add\_task(2.000, 'SoundB', 'Stop');  tl.add\_task(4.100, 'Daq', 'Stop', daqSND);  tl.add\_task(4.100, 'Daq', 'Stop', daqXYZ);  tl.add\_task(4.100, 'SetDIO', off);  tl.add\_task(4.500, 'MUX', deviceA, 'Reset');  tl.add\_task(4.500, 'MUX', deviceB, 'Reset');  tl.add\_task(5.000,'Ready');    %add tasks to list  rz6.write\_tasklist(tl);    %wait for the task to be ready  while (rz6.read\_trialready() == 0)  % do nothing  end    chanlistXYZ = [5 6 7]; %specify the chans to read from  acqdataXYZ = rz6.read\_acqdata(chanlistXYZ); %read acq data from RZ6  chanlistSND = 3; %specify the chans to read from  acqdataSND = rz6.read\_acqdata(chanlistSND); %read acq data from RZ6    delete(tl);  delete(rz6); |

## D4 Recording of the task timing

In this example acquisition channel 11 is used at a sample rate of 1000 Hz in order to show the timing of all the tasks.

tl=biox\_rz6\_tasklist;

rz6=biox\_PAT\_LAB;

tl.debug(false);

chanlist = [11];

samplefrequency = 1000; %Hz

samplerate = rz6.read\_samplerate()

divisor = samplerate/samplefrequency;

% create tasklist here

signalbyte = 128; %128 corresponds to output bit B7

rz6.write\_signalbyte(signalbyte); %write signal byte to RZ6

tl.add\_task(0.000, 'Daq', 'Start', chanlist, divisor);

tl.add\_task(0.000,'SoundA','Tone',250);

% wait for trigger input A4

tl.add\_task(0.000,'WaitForTrigger','External',bin2dec('0001 0000'));

tl.add\_task(0.000,'SoundA','Stop');

tl.add\_task(0.500,'SoundB','Tone',300);

tl.add\_task(1.500,'SoundB','Stop');

tl.add\_task(2.000, 'Daq', 'Stop', chanlist);

tl.add\_task(2.000,'Ready');

%

rz6.write\_tasklist(tl);

rz6.read\_tasklist(tl);

while ~rz6.read\_trialready()

%wait

end

acqdata = rz6.read\_acqdata(chanlist);

data1 = acqdata{1};

figure;

plot(data1);

delete(tl);

delete(rz6);

The result of running this example is the following figure:



The figure shows a spike for every task with a resolution of 1000 samples per second. The A4 trigger is an external trigger activated by a button press.

# Appendix E: Notes on ITD or ILD measurements

## E1 Notes on ITD measurements

For ITD measurement you have to use the task ‘ITD’ in order to specify a delay for the A or B channel and use the task ‘SoundAB’ in order to start the sounds. The sounds can be played in this way with an exact ITD.

When the specified ITD is a fraction (or n times plus a fraction) of the RZ6 clock pulse (~20.5 µsec) the waveform will be resampled, therefore the ITD can be as small as 1 µsec.

The only exception is the use of ‘Noise’ sound. When different LP or HP filters are specified for the A and B channels, the filters introduce different group delays depending on the cut-off frequencies. Furthermore the filters will introduce different phase shifts near the cut-off frequencies, which makes it imposible to correct for. The only way to use noise with different filters for ITD measurement is to prepare phase shift free samples in Matlab and use the ‘WAV’ sound option.

* The ITD-delay comes on top of the task delay.
* When ‘SoundA’ and ‘SoundB’ tasks are used with the same task delay instead of ‘SoundAB’, they are not started exactly at the same time destroying the ITD accuracy.
* ITD can be mixed with ILD.

N.B. For pure ITD measurements you have to make sure that the attenuation and pre-scaling is the same for both channels. By default they are the same.

## E2 Notes on ILD measurements

For ITD measurement you have to use the task ‘Att’ in order to specify the attenuation and pre-scaling of the A and B channels. You have to use the task ‘SoundAB’ in order to start the sounds. You can use this with all sound types.

By default the sounds ‘Tone’ and ‘Sweep’ are scaled to a peak output level of 1V. The peak output level of ‘Multitone’ depends on the parameters of used for ‘ConfigurationA’ or ‘ConfigurationB’. The output level of the ‘Noise’ sound is determined by the RMS level which is 1V by default. The output levels of the ‘WAV’ sounds is determined by the data specified in the function ‘write\_wavdata()’. For all the sounds the output level can be adjusted by the ‘Att’ task.

* For sounds that need **upscaling** of the output level the pre-scaling parameter should be used. But be aware that clipping will occur whenever the peak outputlevel exceeds 10V.
* For sounds that need **downscaling** of the output level the attenuation parameter should be used. If the pre-scaling parameter is used instead for downscaling the signal will loose more bit-depth if the attenuation is more than 20dB.

N.B. For pure ILD measurements you have to make sure that the ITD is the same for both channels. By default they are zero.

N.B. The ‘Att’ task has to be set before the ‘Mux’ task when a multiplexer is used, otherwise clicks will be generated.