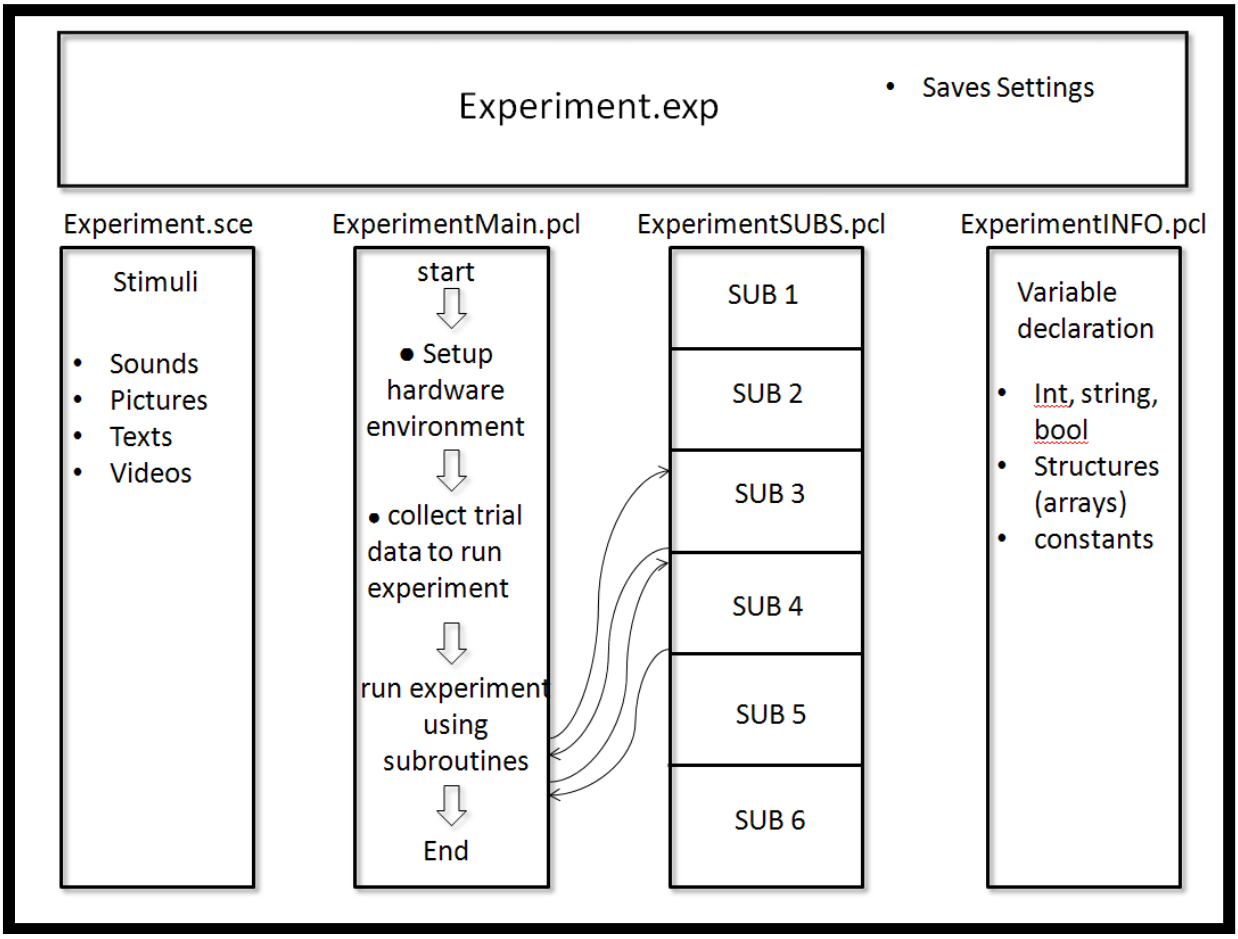
Programming with Presentation

An introductory programming guide for time-accurate experiments

Donders Institute, Nijmegen

Level 1



Contact:

Daan van Rooij (presentation-dccn@fcdonders.ru.nl)

Contents

[1] Introduction (Aim of the course)

Presentation (www.neurobs.com) is a MS Windows based programming tool that allows experimenters to set up and program all sort of experiments. It is the recommended software for time-accurate experiments and therefore supported by the institutes' (Social Sciences Faculty, the MPI and the Donders Centre for Cognitive Neuroimaging) technical groups.

The technical groups offer (PhD) students a couple of preprogrammed experiments (i.e. templates), which can be adjusted to build up own experiments. In this way, the (PhD) student can efficiently program experiments fitting an technical optimal environment.

This is what this course is all about. It aims at teaching programming skills, which are needed to modify the existing templates such that they meet your own demands. Because this can be quite complex, we start with short assignments, which address one basic and simple problem at a time. They will all contribute to the final assignment in which you will work on an existing template.

[1-1] SDL and PCL: Define and control

Programming in Presentation is somewhat different from programming in other languages (e.g. Matlab) because it actually consists of two different programming languages, SDL and PCL.**SDL** (Scenario Description Language) forms the core of the program Presentation and you could possibly programm your whole experiment in this language. But using SDL means that you are not very flexible and flexibility is what you get by making use of **PCL** (Program Control Language). PCL builds upon SDL and allows you to control elements you have defined in SDL.

In principle, you are free to choose how much SDL you use in your experiments. As said above you can use SDL with no PCL at all but you might also reduce SDL to its minimum by programming the **stimuli definitions** in SDL and everything else in PCL. We prefer the latter option because it gives us a maximum of control and flexibility.

Defining our stimuli in SDL means that we define here ***what*** kind of pictures (pictures, text) and sounds we want to use.

Then, we use PCL to **control** the definitions made in SDL, for example ***when*** we would like to present a defined picture or sound.

This is a bit different from what the developers of Presentation expect programmers to make use of their software. For that reason we advise not to start off by reading the Presentation website documentation as programming is taught there differently. Thus, our goal is to teach you to build your own experiment written in PCL. SDL is only for experiment initiation and defining stimuli. Let's have a closer look at these stimulus definitions now.

Stimulus Definition: What to present

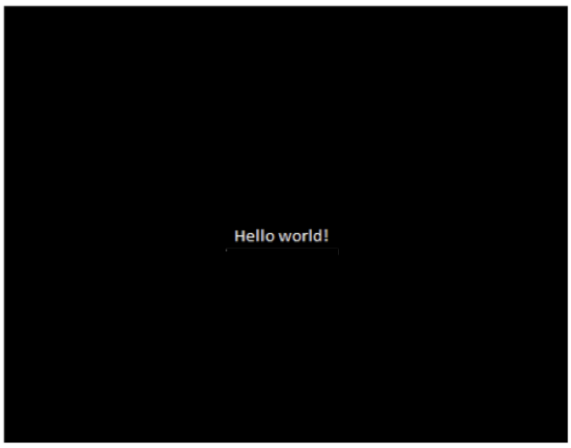
For most experiments, we want to present visual or acoustic stimuli or a combination of both. It is also possible to present videos, although this topic is outside the scope of this course. Thus, what we need to do is to define visual and acoustic stimuli.

For a visual stimulus, we have an object of the type '**picture'**, for acoustic stimuli we have a '**sound'**-object.

A visual stimulus definition would look like this:

```
picture
 {
        text { caption = " Hello world! "; font_size = 16;} t_Text1; x = 0; y = 0;
 } p_Picture1;
```

As we can see, our visual stimulus has the name 'p_Picture1'. Inside the curly brackets, it is defined, what our stimulus consists of. In this case, it consists of text, which has the caption 'Hello world!' and a certain font size. The text itself also has a name, which is called 't_Text1. And it has coordinates (x and y), which describe the position of the text relative to the centre of the screen (in pixels). In this example the text would be centred in the middle of the screen. The presentation of this picture looks like this:



An acoustic stimulus definion would look like this:

```
sound {
        wavefile { filename = "beep.wav"; preload = true;}w_Beep;
} s_Beep;
```

Our sound has the name ‘s_Beep’. It consists of a wave-file, which has a file name (‘beep.wav’) and the characteristic that it can be preloaded.

Picture and sound stimuli can have a lot more characteristics. We will come to that later (see section 4-2).

Control over stimuli: When to present

As already mentioned, we use PCL to control our in SDL defined stimuli. For example, we want to present and manipulate our stimuli. There are a lot of existing PCL-functions for this, and it is possible to build every possible experiment upon these functions. Let’s start with one line of PCL code

```
p_Picture1.present();
```

Based on our SDL stimulus definition, PCL knows that p_Picture is an object of the type ‘picture’ and pictures can be presented on the screen. The function *‘present()’* puts the picture immediately on the screen. We would see it appearing, however, it wouldn’t stay there for long (just for a fraction of a second). We explicitly have to tell the programm to keep on running without doing anything. Another built-in function doing exactly that is ‘**wait_interval’**. *wait_interval* keeps the program as it is and waits for the amount of milliseconds we command to wait. For example, if we want the picture p_Picture1 to stay on the screen for 3 seconds, the command in PCL looks like this

```
wait_interval(3000);
```

[1-2] Presentation Programming Characteristics and Conventions

So far, you have seen some SDL variable definitions and function calls. To operate successfully, you have to keep some Presentation specific characteristics in mind. One of these is that variable definitions and function calls have to end with a semicolon. If you forget one, you will get a lot of errors. Another characteristic is that Presentation is case-sensitive. Case-sensitivity means that it does matter whether you write a variable or function with capital letters or small letters. This has the consequence that you have to pay attention to the names you use. For example, if you have defined a picture as *p_Picture1* in SDL, you can’t put in on the screen by calling p_picture1.present(). Instead, picture has to be written with a capital *P* (p_**P**icture1.present()).

These are the characteristics. Let’s move to the conventions or style issues. In principal, it is possible to give your variables and subroutines (see section 5-1) every possible name. But, in order to make your code easier to check and share between different programmers, we offer some guidelines in how you can make your code more consistent. You will find these guidelines in the appendix

Assignment 1: A simple program

Objective

Learning the basic structure of a presentation experiment

Let's start with a very simple programming exercise.

1. Open Presentation and switch to the build-in editor. In fact you could use any editor, but this editor has a syntax highlight and a debug function.

2. Make a simple program in Presentation. Therefore you have to create only one SDL file. You have to type the code underneath. You can leave out the comments, which are indicated by a "#" in front of them. If you don't understand every line of code, don't worry, you will get more information later on

```
#SDL:
# Initiation of the scenario and the main PCL-file
#pcl_file = "HelloPCL.pcl";
scenario = "Hello";
begin;                          #begin of SDL program
picture
  {
    text { caption = " Hello world! "; font_size = 16;} text1; x = 0; y = 0;
  } p_Picture1;

#PCL:
begin_pcl;  # begin of PCL program; only usable without additional pcl file
p_Picture1.present();                   #present picture on screen
wait_interval(5000);
```

3. Check / Debug your program until it is running. Use the key 'F7' or the checkbox next to the 'Run experiment' checkbox for this purpose ('Run experiment' is the green arrow at the top of your editor). The debugger will tell you where exactly the error comes from. You can also adapt the syntax coloring to easily highlight PCL or SDL code. This is another checkbox at the top of your editor.

4. Try to separate SDL and PCL code by making 2 files, a SDL file AND a PCL file. The line "*pcl_file = "HelloPCL.pcl";"* makes it possible that the stimuli, defined in SDL, are manipulated by the code in a different PCL file. Pay attention to the naming of the files, so that the SDL part finds what it needs. To test your experiment you **always** have to **run the SDL file** and not any PCL-file. Presentation will by itself run through the code of a PCL file if the *pcl_file* statement is used.

When the experiment is running without errors save all program files ('Save experiment as').

[2] Variables and arrays

[2-1] Variables

Variables are place holders to store values. You can choose almost any name for a variable, although it is common to use names that clearly indicate what the variables actually do. And it makes the code more readable.

Before a variable can be used, you have to declare it by specifying its data type. This means that you specify what kind of information you want to store in that variable. An example of a variable declaration would be:

```
int iCount;
string sMyWord;
```

In the first line of code, the variable with the name "iCount" is defined as an integer. An integer represents a whole-number data type, which is different from e.g. doubles which can also be decimals. PCL contains four basic variable types and many reference variable types. We will focus on the four basic types here, which are as follows:

- int : positive or negative whole number
- double : number with decimals
- bool : boolean variable (true or false)
- string : character string variables (word or sentences)

In some cases you should ensure that you initialize variables by assigning them an initial value. If you don"t assign an initial value to a new variable, it gets an arbitrary value.

Examples of variable declarations where initial values are assigned:

```
int iI = 5;
double dD = 10.9;
double dNewVolume = dD * 0.054;
string sMyName = "Bob";
bool bDoOver = false;
```

[2-2] Array

An important object in Presentation is an array. An array is a collection of values of the same type. The easiest way to visualize an array is to imagine an array as a table that contains only one row with one or (almost always) more columns. Each column contains a value, and a program can use the number of each column to access the value in it. The number of such a column is referred to as the index of that array.

In Presentation, the index begins with the integer 1 and is always written inside squared brackets. An array declaration has the following form:

```
array <int> aiButtonPressed [10];
array <int> aiPictureTime [5];
```

In the first example, aiButtonPressed consists of ten elements of the type integer. To address an individual element in it, you have to write the element's index in square brackets straight after the variable name. For example, to obtain the first value, you would need to use the first of the following commands, for the last value, you would need the second one:

```
Int iFirstValue = aiButtonPressed[1];
Int iLastValue = aiButtonPressed[10];
```

Here, one more example of an array declaration with initialization. The expression between the brackets specifies the size of the array.

```
array <int> aiWaitTimes[3] = { 41, 191, 491};
```

Arrays are not restricted to one dimension. For example, if you want to save values from a table in an array you could also add an extra dimension to the array:

```
array <int> aiTwoDimensionArray[10][5];
array <int> aiTDA[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
```

The first array corresponds to a table with ten columns a five rows. This means that 50 values of the type integer can be stored in this table.

[3] Control Statements

In Presentation, you can use control statements to alternate the sequence of your program. This is necessary to make your program flexible. In the following paragraphs, 2 types of control statements will be discussed: '**loop until**' and '**if else**'.

[3-1] The 'loop until'-statement

The 'loop until'-statement is a loop. A loop is a segment of program code that repeats itself. Loops are used, for instance, to run through an array from top to bottom. When working with loops, the programmer needs to ensure that the loop is closed such that the program can leave the loop. Otherwise, it will keep on looping forever.

In Presentation, 'loop-until'-statements contain three parts.

1. The statement '*loop'* indicates the start of the statement. This is followed by optional local variable declarations. It is good to realize that these variables just exist within the loop.
2. After the declaration part, the *'until'*-statement indicates the evaluation part.
3. The statement *'begin'* marks the body of the loop. All the code that follows this begin will be repeated. Finally, the statement '*end'* marks the end of the body.

As an example, here is a loop counting every second for 1 minute.

```
loop    int iSec = 0                        #integer declaration, loop starting from 0
until   iSec > 60                           #loop until integer iSec is bigger than 60
begin
        t_Text1.set_caption(string(iSec));  #change t_Text1 of p_Picture1. As captions can only contain strings,
                                            #we convert with the function string the integer iSec into a string
        t_Text1.redraw();                   #redraw the picture on the graphical card
        p_Picture1.present();               #show the picture on screen
        wait_interval(1000);
        iSec = iSec + 1;                    #add one to integer iSec
end;
```

Here some explanations:

int iSec = 0 - initialization of a temporary variable (count of seconds)

iSec > 60 -The test, when it proves true, the loop is exited.

iSec = iSec +1 -The variable is used to count the number of repetitions

The variable iSec is printed as a number to the monitor

[3-2] The if else statement

Often, it is important to let a program do things differently, depending on the value of a variable. For example, if a variable has a positive value, the program has to react differently than when it is negative.

```
if (someVariable > 0) then
  // do something
else
 // do something different
end;
```

In Presentation, the 'if else'-statement is used to manage choices whether to do something at a certain point, or to do something else. The choice is always made on basis of an evaluation within the statement. Possible evaluations are as follows:

```
x == y (x is equal to y)
x != y (x is not equal to y)
x <  y (x is less than y)
x >  y (x is greater than y)

x <= y (x is less than or equal to y)
x >= y (x is greater than or equal to y)
```

The following example code evaluates whether a student has passed an exam. If students receive a mark higher than 5.6 he has passed the test.

```
if (dResult >= 5.6) then
        t_Text1.set_caption("Pass");                    #change t_Text1
else
        t_Text1.set_caption("Fail");
end;
```

It is also possible to use the 'if'-part without a following 'else'-part. Here an example:

```
if (iTemperature < 0) then
        t_Text1.set_caption("Frozen\n");
end;
```

Each if-statement is actually a logical test. If the test is true, then the following line of code is executed. If the test is false, then the statement following the 'else' is executed (if present). After this, the rest of the program continues as normal.

Sometimes, we wish to make a choice out of several conditions. The most general way of doing this is by using the '*elseif*' variant of the if- statement. This works by cascading several comparisons. As soon as one of these tests gives the result 'true', the following code is executed, and no further test is performed. In the following example grades are awarded depending on the result of an exam.

```
if (iResult >= 75) then
        t_Text1.set_caption("Passed: Grade A ");
elseif (iResult >= 60) then
        t_Text1.set_caption("Passed: Grade B");
elseif (iResult >= 55) then
        t_Text1.set_caption("Passed: Grade C");
else
        t_Text1.set_caption("Failed\n");
end;
```

Assignment 2: Grades

As you have just read, Presentation works with variables and control structures. Both are commonly combined, for example a loop that runs through all elements of an array. Or think of visual feedback, which depends on the value of a certain variable. Now, we will make a short program consisting of an array.
1.Try to run the next program in Presentation. Create an SDL file and a PCL file
2. Present all grades and the average grade on the screen. Present only *one* screen with all the grades and the average; don't make a new screen for each grade. For help see section [3-1].
3. Add a 3rd column to the screen, indicating whether the student has passed or failed. For help see section [3-2].

```
#SDL:
scenario = "Grades";
pcl_file = "GradesPCL.pcl";
begin;
picture
 {
   text { caption = "grades on screen "; } t_Grades; x = 0; y = 0;
```

```
} p_Grades;

#PCL:
array <double> adGrades[5] = { 7.1, 4.5, 6.3, 5.8, 8.2};
double dTotal, dAverage;
string sTextOnScreen;

dTotal = 0.0;
sTextOnScreen = ("studentnr:   grade:\n"); # \n refers to a next line or a ENTER.


int  iCount=1;
dTotal = dTotal + adGrades[iCount];
sTextOnScreen.append(string(iCount) + "        " + string(adGrades[iCount]) + "\n");# The function append
                                                    #appends a string to an already existing string
dAverage = dTotal ;
sTextOnScreen.append("the average is: " + string(dAverage));
t_Grades.set_caption(sTextOnScreen);
t_Grades.redraw();
p_Grades.present();
wait_interval(5000);
```

[4] Built-in objects and functions

[4-1] Built-in functions

When working on assignment 2, new code appeared. For example, adGrades.count() was used in the loop to return the length of the array 'adGrades'. In programming terms, *.count()* is a method of adGrades, which is an object of the type array. There are several kinds of methods for several types of objects. You can recognize a method by the dot before and the brackets after the method's name.

Tip: When programming, all available methods show up when you put a dot after your variable.

[4-2] Getting help

To make life easier, a lot of necessary and useful functions (methods) are already built into Presentation. To get an idea of what is there, Presentation has a very good help function. By opening the Documentation (F1), you can click on "Index" and then search on an object. Try this for the *string* object and select the PCL-type. If you managed that, you will get information about the function we used in our last assignment for the variable sTextOnScreen.

[4-3] Special built-in variables

Next to types as for example arrays or strings, which we already discussed, there are also some other types to mention. These are **clock** and the **response_manager**, and both are crucial for your experiment.

Let's start with **clock**. When you look up *clock* in the documentation, you will get to know that it is a PCL predefined variable with methods, returning the time at the moment the method is called.

The type r**esponse_manager** keeps track of the responses that have been given during an experiment. It has a lot of functions. Look these functions up in the documentation.

Assignment 3: Working with the help function

1. Find a built-in method that is suitable to measure that a response has been made (see section [4-2] and [4-3].

2. Look up what the built-in functions which we have used so far exactly do.

3. What makes the function picture.present() so important for us?

[5] Programming and structuring your experiment

[5-1] Subroutines

With all knowledge from previous chapters, we can start programming our experiment. When programming in Presentation, it is often the case that a certain sequence of operations which form a whole function is put together, forming a little module. Such a sequence can have a form like

1) Change the text of a stimulus
2) Refresh the memory where the stimulus is stored and
3) Put the new stimulus on the screen.

Such a module is called a **subroutine** and it forms the core of PCL programming. You can build your whole experiment from these little modules, because they can be accessed anywhere in your PCL code, or within another subroutine, via a call, using the name of the subroutine. This makes your code easily readable and very flexible.

A subroutine consists of

1. The subroutine header in which the name of the subroutine is defined and

2. The body of the subroutine, which contains the statements.

Here is an example of a subroutine:

```
sub ShowText( string sSubInput)
begin
        t_Text1.set_caption( sSubInput );
        t_Text1.redraw();
        p_Text.present();
end;
```

Let us examine the details of this subroutine.

The subroutine's header is introduced by the reserved word '**sub'** and followed by the subroutine's name. After that you can see brackets containing variables you want to pass to the subroutine, the so-called arguments. Each argument is separated from the next by a comma. A function can also be defined without arguments, then brackets have to be omitted.

The body of the subroutine starts with 'begin'. The procedure finishes with the word end. All statements between these words will be executed when the subroutine is called. Any variable declared here will be treated as local.

The subroutine described above can be called in PCL by a line like the following.

```
ShowText ("Hello World");
```

It seems like a useless function, but when something has to be printed several times, for instance, when computing a list, it can become very useful.

Let's take another subroutine, which can calculate square values of integers returning this value. It is important to ensure that your arguments are from the expected type. Otherwise the function will produce strange errors.

```
sub calculateSquare(int iNumber) begin

        iNumber = iNumber * iNumber;

end;
```

Assignment 4: **GetInput**

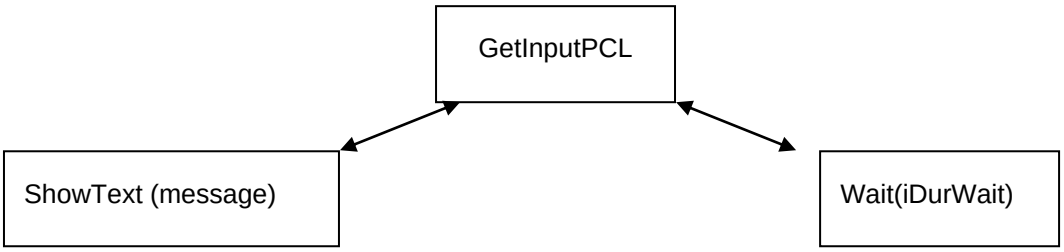
1.Try to run the next program in Presentation.

```
scenario = "GetInput";
pcl_file = "GetInputPCL.pcl";
begin;

picture {
        text { caption = "Enter something:"; }t_Text1; x = 0; y = 100;
        text { caption = " "; } t_Text2; x = 0; y = 0;
} p_Screen;


#PCL:
sub ShowText( string sSubInput)
begin
        t_Text2.set_caption(sSubInput);
        t_Text2.redraw();
        p_Screen.present();
end;

string sInput;
sInput = system_keyboard.get_input(p_Screen, t_Text2);
ShowText( "you pressed: " + sInput );
wait_interval(1000);
```

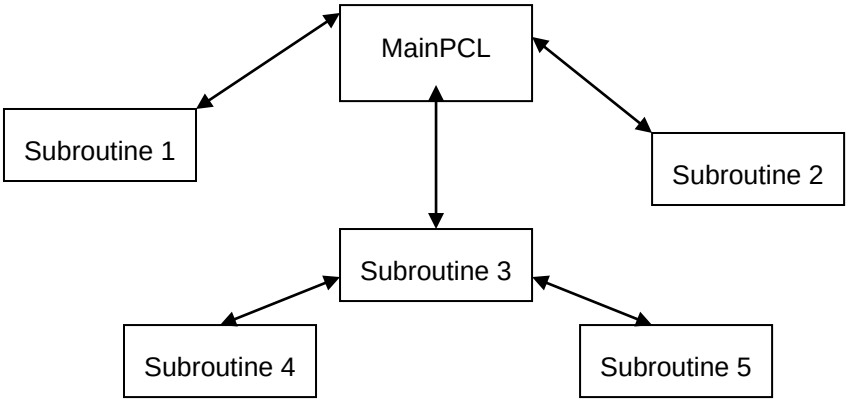
2. Create a program in which you have to enter your name, surname, age, and student number. Show on the monitor what you entered. Try to keep the program as short as possible. (see again [3-1])
3. Instead of the build-in function *wait_interval*, you can make a subroutine doing the same. Program such a subroutine, give it the name 'Wait (int iDurWait)' and use it as a substitute for wait_interval. (see [5-1])

[5-2] Include

In the last assignment you have learned to make a subroutine with a passed argument. The integer after 'Wait' was used to define how long the routine had to wait. This was done from the main programm.



But, subroutines can also be called from within another subroutine as in the example below.

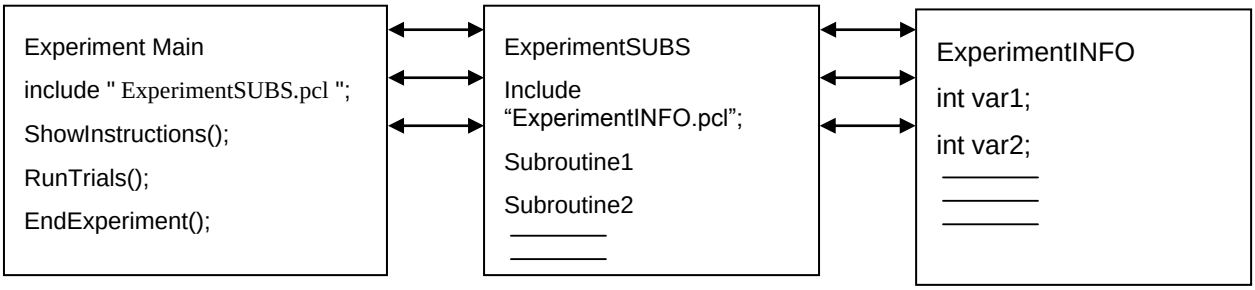


By this, all individual subroutines created for your experiment can be seen as building bricks forming your experiment.

As we progress in programming our experiment and the code and number of functions increase, we need to organize the code. We can for example take all subroutines out of the main program file and make a separate *subroutine* file for them. Let's call this file ExperimentSUBS.pcl.

For the PCL-variables we can do the same. We take all of them and put them into a separate *variable* file. Let's call this file ExperimentINFO.pcl.

By doing this, we get a nicely structured experiment, allowing us to keep overview.



For sure, Presentation needs to know where to search for the variables and subroutines. Therefore, a link has to be made. The command *include* is the instruction we need to incorporate the entire contents of another file at the point we use the command.

The names of the files we want to include need to be enclosed by double quotes (i.e. "file name.pcl"). Presentation will then look for these files in the folder for the experiment profile definitions (This path can be found in the Main or Scenario's tab within Presentation). Here, an example:

```
# Include all the variables.
include "TemplateINFO.pcl";
# Include all the subroutines.
include "TemplateSUBS.pcl";
```

When we use the include command, the included pcl-file is read in first, and all the information is accessible in the main program, (PCL) file. Actually, it doesn't matter whether you put part of your code in an included PCL file or on top of your experiment. The advantage of the first option is that it gives us a much better overview.

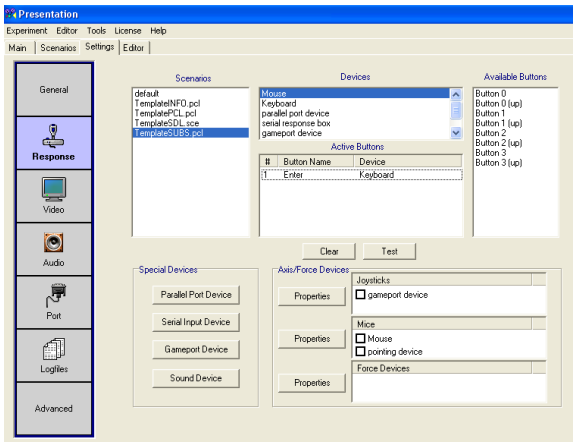
The only thing you have to be aware of is that the order in which you include and define variables and subroutines is very important. Code is read in hierarchically (top-down), so when you want to use a variable/subroutine, it has to be defined earlier (above) in the code.

[5-3] Settings and the Experiment File

So far, we have worked mostly with the editor of Presentation. This was sufficient as long as we just presented stimuli. However, for most experiments we also want to record button presses. Therefore, we have to define buttons which we can use during the experiment. We do this in the program's 'Response'-menu under **Settings (see next page).**

Here, all available input devices are listed under Devices (e.g. keyboard). All available Buttons can be found in the window 'Available Buttons' to the right. Under 'Active buttons', you find selected buttons. In this example, the 'enter'-button is defined as button number 1. Every extra added button will get the next possible button number. Finally, in the scenarios window, it is indicated to which scenario the defined buttons apply.

If you want to store your settings, you have to make use of the **experiment** files. In these files, all settings are stored, as for instance, which buttons you have defined, which ports to use, where to save the logfiles and so forth.



The experiment file is also very important for organizing your experiment. Make sure that you have for every experiment a separate experiment file. Otherwise, your experiment easily gets messed up with other settings than you want. To make experiment files, you have to choose ‘save experiment as’ in the experiment menu. You will get a file with the extension .exp, which can best be stored in the same folder as the SDL and PCL files.

Assignment 5: Reaction Time

1.Make a new experiment with the code below. Structure it in the way discussed in section [5-2].

```
#SDL
scenario = "ReactionTime";
pcl_file = "RTMAIN.pcl";

active_buttons = 1;
button_codes = 1;

begin;
######## PICTURES ########
picture {
        box { height = 1; width = 30; color = 255,255,255; } b_Horz; x = 0; y = 0;
        box { height = 30; width = 1; color = 255,255,255; } b_Vert; x = 0; y = 0;
} p_Fixation;

picture {
        box { height = 30; width = 30; color = 255,255,255; } b_Box; x = 0; y = 0;
} p_Box;


#PCL:
int iCountOld;
#----------------------------Wait----------------------------------
sub Wait(int iDurWait) begin
        int iTimeStamp;
        loop
                iTimeStamp = clock.time();
        until (clock.time() - iTimeStamp >= iDurWait)
        begin
        end;
end;

#----------------------------WaitForAllButtonpress----------------------------------
sub WaitForAllButtonpress begin
#insert code
end;
```

```
############################## TRIAL ########################
sub RunTrials begin
	int iTrialCount;
	int iRunAmountOfTrials = 10;

	loop iTrialCount = 1
	until iTrialCount > iRunAmountOfTrials
	begin
		p_Fixation.present();
		Wait(random(500,1000)); #random(a,b) is a function returning a random value >= a and  <=b
		p_Box.present();
		Wait (2000);
		iTrialCount = iTrialCount + 1;
	end;
end;
RunTrials();
```

2. Define one keyboard key as active button and run the experiment (see [5-3]).

3. Add the code for the subroutine 'WaitForAllButtonpress'. This subroutine should run unless a button is pressed (for help see section [4-3] and [5-1])

3. What would happen if you put an integer within the brackets of the function you have used to record the button press? Would it make a difference with leaving it empty? Where can you look it up if you wouldn't know?

4. Write another subroutine which stops a) after a button is pressed OR b) after 1 second has passed. Call this subroutine 'WaitForAllButtonpressTime(int iDurWait)' and use it instead of the WaitForAllButtonpress subroutine.

[6] Data Import and Export

So far, we have focussed on stimuli presentation and responding to these stimuli. We let Presentation create the stimuli and put the results on the screen. But we need to record the data in external logfiles and sometimes we have already existing stimuli lists, we want to use for our experiment. For these functions we need data import and export .

[6-1] Export

Exporting in Presentation is very easy and works in two steps. What you have to do first is to define an output file where Presentation can direct output to. You do this by using the **output_file** type, which is another special type. You can make such a definition by this code:

```
output_file OutputFile = new output_file;
```

Second, you have to open the output file every time you want to write to it. If you want to open an new, empty output file, this can be done with the built-in function *.open()*.
For *.open()*, you need the file name (i.e. how to call the file on the hard disk) as argument and possibly a Boolean. The Boolean helps protecting your files if a file with that name already exists. If you don't want to open a new file, but just add information to an exisiting file, you use *.open_append()*.This will open an existing output file, and past any new information directly behind the existing content.

There are two more functions needed, *print()* and *close()*. With *.print()* you add a string to your output file on the hard disk, with *.close()*, you close your file.

```
OutputFile.open( "subject1.txt", false ); # don't overwrite  subject1 if it exists
OutputFile.print("Subject Number\t Trial\n");
OutputFile.close();
```

[6-2] Importing

Importing or reading in data is almost as easy as exporting and works in a comparable manner. First you have to define a new special type, the **input_file** type. Code for such a definition looks like this:

```
input_file InputFile = new input_file;
```

Then, you open this type by calling the build-in function .open('filename'). The argument 'filename' refers to the name of the file on hard disk.

```
InputFile.open( "stimuli.txt");
```

The last part, reading in your data, depends heavily on the format of your input file. For instance, if you have a header, you have to skip this; or depending on the delimiter, you have to tell presentation which one it is (The help-function can give you information about useful functions).

In any case, we have to make a loop to read in the whole file until the file end is reached. Presentation has for this the built-in function *.end_of_file()*. Code could look like this:

```
loop iTrialCount = 1
until InputFile.end_of_file() || !InputFile.last_succeeded()
begin
	asTrialInputData[iTrialCount] = InputFile.get_string();
	iTrialCount = iTrialCount + 1;
end;
```

It can also happen that an input file is not complete (e.g. an empty field) or an error occurred. Therefore, it is also important to check this. We know this by using the *.end_of_file()* or *.last_succeeded()* function.

```
if !InputFile.last_succeeded() then
	term.print( "Error - not enough lines in stimuli List\n" )
elseif !InputFile.end_of_file() then
	term.print( "Error - too many lines in stimuli List\n" )
end;
```

Finally, you need to close your input file by calling:

```
InputFile.close();
```

Assignment 6: Red-Green

Let's start making another real experiment. The following program will show a fixation cross during a randomized period between 200 and 600 milliseconds. Then a red or green box is presented, this box will be randomly picked and placed on your screen. When it is presented the participant has to respond as fast as possible.

```
scenario = "Red Green";
pcl_file = "Red GreenMAIN.pcl";

active_buttons = 1;
button_codes = 1;

begin;

picture {
	box { height = 1; width = 30; color = 255,255,255; } b_Horz; x = 0; y = 0;
	box { height = 30; width = 1; color = 255,255,255; } b_Vert; x = 0; y = 0;
} p_Fixation;

picture {
	box { height = 30; width = 30; color = 255,255,255; } b_Box; x = 0; y = 0;
} p_Box;
```

```
picture {
        text { caption = " "; font_size = 16; } t_Text1; x = 0; y = 0;
        text { caption = " "; font_size = 16; } t_Text2; x = 0; y = -35;
} p_Text;

picture {
        text { caption = "End of the experiment."; font_size = 16; } t_End1; x = 0; y = 0;
        text { caption = "Thanks for your participation!"; font_size = 16; } t_End1b; x = 0; y = -35;
} p_End;
```

#PCL:
#Main:

```
RunTrials();                                    #a subroutine call
EndExperiment();                                #a subroutine call
```

#SUBS:

```
#Insert 1) ShowText 2)Wait 3) WaitForAllButtonpress from the last assignment.
# Make sure that the order is correct, then go on with following code

sub CreateHeaderOutputFile(string sFileName ) begin
end

sub WriteTrialToOutputFile(string sFileName) begin
end;

sub RunTrials begin
        loop
                int iTrialCount = 1
        until iTrialCount > iRunAmountOfTrials
        begin
                p_Fixation.present();
                aiFixationTime[iTrialCount] = clock.time();   #use of an array!
                Wait(random(400,800));
                iBoxColor = random(1,2);

                if iBoxColor == 1 then
                        b_Box.set_color(255,0,0); #change to red
                else
                        b_Box.set_color(0,255,0); #change to green
                end;
                p_Box.set_part_x(1, random(-300,300));     #change the x pos
                p_Box.set_part_y(1, random(-300,300));     #change the y pos
                p_Box.present();

                WaitForAllButtonpress();

                iTrialCount = iTrialCount + 1;
        end;
end;

sub EndExperiment begin
        p_End.present();
        WaitForAllButtonpress()
end;
```

#INFO:

```
# ---------------------- VARIABLES INITIATION ----------------------------
int iCountOld;
int iRunAmountOfTrials = 24;
int iTotalAmountOfTrials = 24;
int iTimeStamp;



int iButtonRed = 1;
```

```
int iButtonGreen = 2;
int iTimeWait;
int iBoxColor;

#---------- LOGDATA ----------;
array <string> asWord[iTotalAmountOfTrials];
array <int> aiFixationTime[iTotalAmountOfTrials];
array <int> aiPictureTime[iTotalAmountOfTrials];
array <int> aiButtonPressed[iTotalAmountOfTrials];
array <int> aiButtonTime[iTotalAmountOfTrials];
```

1. Create an Eperiment, SDL, PCL, INFO and SUBS file of this code, including the three subroutines from your previous assignment. Store them correctly with experiment, SCE, MAIN, INFO and SUBS pcl file (see [5-3])
2. Store the reaction time and the color of the box and write it to the screen after every trial.
3. Change the reaction buttons of red and green to 'r' and 'g'. Make sure that the program can only respond to these two buttons.
4. Make sure that the participant has to respond within 1 second. If there is no response within one second write to the screen 'Please respond faster'.
5. Extend your program that it 1) asks for an output file name and 2) exports this file name, trial number, box colors and reaction times to a file with a given name. If you have difficulty you can look up the help file and search for output file. (see [6-1])
6. Make with excel or another program a file, containing the condition 'red' or 'green'. Use this file to determine at forehand the color of the presented boxes and run the experiment. To debug your program, you can call the 'ShowText'- and 'Wait' function after each row of read-in data to put the imported data on the screen. (see [6-2])

[7] Stimuli

[7-1] Visual

Computer screens are updated according to their refresh rate. What happens then is that the whole screen is redrawn line by line all the time. For example, with a refresh rate of 60Hz, the screen is redrawn 60 times per second (1000ms) and the duration it takes to redraw it line by line is 16.66 ms (1000/60). When attempting to redraw the screen while it is currently already being updated (the lines are drawn) the result might lead to artifacts, since the update occurs immediately, leading to parts of both, the new and and the old screen content, being visible. What is even worse is that you will never know in which phase of the redrawing the new redraw was started. Thus, you cannot be sure when exactly the new content is fully visible on screen.

The first step towards getting around this problem is to synchronize the actual redraw to the vertical retrace of the screen. This means that a change in content will never happen immediately, but always only when the retrace is at the top left position. When synchronizing to the vertical retrace, the graphic card is told to update the screen the next time it starts redrawing the first line. While this will solve the problem of artifacts, you will still face the problem of not knowing when exactly something was visible on the screen, since the graphic card handles this synchronization itself in the background.

Presentation solves this problem with the *.present()* function. It allows waiting for the vertical retrace to actually happen before proceeding with the code that tells the graphic card to update the screen (this is also known as blocking on the vertical retrace). This means that whenever something should be presented on screen, no matter in which line the redraw is at this point in time, the graphic card will wait for the redraw to be in the first line and then present the stimulus. Since the code is blocking, the time presentation reports the stimulus to be presented on screen will always be the time when the redraw is starting at the first line.

SDL:
```
picture {
        box { height = 30; width = 30; color = 255,255,255; } b_Box; x = 0; y = 0;
} p_Box;
```

PCL:
```
p_Box.present();                    #pay attention to the order of p_Box.present() and clock.time()
iTimePicture=clock.time();
```

It is important to switch off power saving schemes of your graphic card's driver, and use a special graphical card provided by your technical support!

[7-2] Audio

To play back audio you have to create an audio handle in SDL.

SDL:
```
sound {
        wavefile { filename = "beep.wav"; preload = true;} w_beep;
} s_Beep;
```

PCL:
```
# The order of clock.time() and present is different from when a picture is presented!
iTimeSound = clock.time();              #log the exact time a sound is played
s_Beep.present();                       #play the sound
```

The *present()* method in PCL for auditory stimuli (s_Beep.present())will be executed immediately. Unfortunately, there will still be a delay before the audio can be heard, since the audio stream has to be sent to the hardware. However, it is assumed that this latency is relative stable (Tests showed a constant delay around 5 ms).

It is important to set your sample rate, bit depth and audio buffer size correctly. Setting the buffersize too low will result in distorted audio! And again use a special sound card provided by your technical support!

[7-3] Video

Video presentation is a tricky subject. In presentation, the *present()* or *play()* methods of a video stimulus will start playback and present the first (current) frame on the screen. Thus, visual onset of this frame will be synchronized with the vertical retrace (see visual stimulus presentation above). Each following frame has to be plotted on the screen and the screen has to be updated. The process of plotting each frame might take longer than one refresh rate, which will result in dropping frames (e.g. frames not being presented at all). Tests have been done with indeo video and mpeg-1. They work most of the time but not always. Synchronizing video with sound is also a difficult task.

Example code: play video fully in PCL.

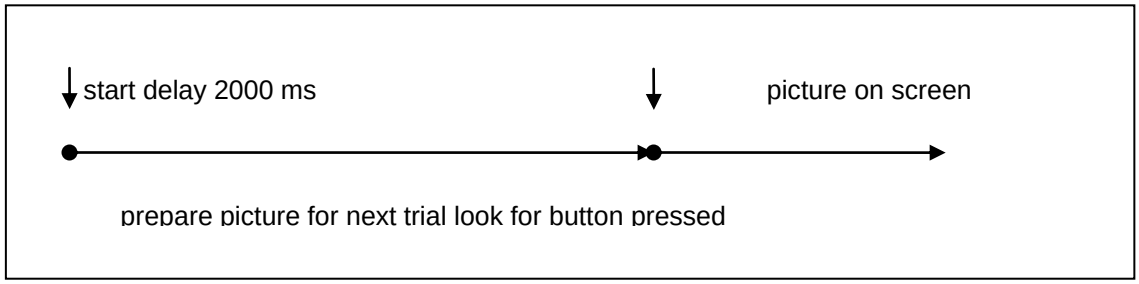
```
video v_Vid2 = new video;
v_Vid2.set_filename( "00158.avi" );
v_Vid2.prepare();
video_player.play( v_Vid2, "video1" );
```

```
loop until !video_player.playing()
begin
end;
v_Vid2.release();
```

Right now, if you would like to use video, contact your technical support for more details! Video will not be discussed in more detail in the remaining of the course.

[7-4] Visualizing your experiment: A time line

In order to keep track of the desired timing of your experiment, it might be useful to visualize it by drawing a time line. A time line gives a nice overview how your experiment is build up and when and what has to happen. It is advised to draw a timeline of the experiment before you start to program. In this manner you can decompose your program into event pieces and start programming these step by step (or bit by bit ;-).



In this example you start with creating a delay. Then you prepare your picture. When a button is pressed, the picture occurs on your screen. Programming done!

Assignment 7: Time line

1.Draw a timeline of the last assignment. Note that a timeline of an experiment is needed to get a clear overview of your experiment and to divide the total program into smaller programming parts.
2.Try to READ the next program. This program comes close to the final template we are going to use for the final assignment.
3. Create a timeline for this experiment.

#INFO file

```
# ---------- VARIABLES INITIATION ----------
int iCountOld;
int iDurWait;
int iTimeStamp;
int iTrialCount;
int iTotalTrialCount;
string sKeyboardInput;

# ---------- FILES ----------
string sOutputFilename;

# ---------- INPUT FILE VARIABLES INITIATION ----------;
int iTotalAmountOfTrials = 24;
int iRunAmountOfTrials = 1;

# ---------- OUTPUT FILE VARIABLES INITIATION ----------;
output_file OutputFile = new output_file;
string TrialData;
array <int> iFixationTime[iTotalAmountOfTrials];
array <int> iPictureTime[iTotalAmountOfTrials][2];
array <int> iButtonPressed[iTotalAmountOfTrials];
array <int> iButtonTime[iTotalAmountOfTrials];

int iSelectStimulus;
```

#SDL file

```
####### INITIATION #######

scenario = "Template";
pcl_file = "TemplateMAIN.pcl";

active_buttons = 1;
button_codes = 1;

default_background_color = 0, 0, 0;
default_font = "arial";
default_font_size = 20;
default_text_color = 235, 235, 235;
default_text_align = align_left;

begin;

######## PICTURES ########

picture { } p_Default;                          #zwart scherm

picture {
                box { height = 1; width = 30; color = 255,255,255; } b_Horz;
                x = 0; y = 0;
                box { height = 30; width = 1; color = 255,255,255; } b_Vert;
                x = 0; y = 0;
} p_Fixation;

picture { text { caption = "Welkom!"; } t_Welcome1a; x = 0; y = 0;
        text { caption = "We gaan zo beginnen.";}t_Welcome1b; x = 0; y = -35;
                text { caption = "De flanker taak zal ongeveer 20 minuten duren.";}t_Welcome1c; x = 0; y = -
70;
                text { caption = "[Druk op de knop om verder te gaan]";}t_Welcome1d; x = 0; y = -140;
} p_Welcome;

picture { text { caption = " "; } t_Info1; x = 0; y = 0;
        text { caption = " "; } t_Info2; x = 0; y = -40;
        text { caption = "Press [ENTER] to confirm or [Esc] to abort. "; } t_info3; x = 0; y = -150;
} p_Info;


######## STIMULI #########
text { caption = "HH"; font_size = 16; } t_FlankerHH;
text { caption = "SS"; font_size = 16; } t_FlankerSS;
text { caption = "H"; font_size = 16; } t_FlankerH;
text { caption = "S"; font_size = 16; } t_FlankerS;
```

```
picture {
} p_Flanker;

############ END ############
```

#PCL MAIN file

```
####### INITIATION #######

include "TemplateSUBS.pcl";              # Include all the subroutines.

GetKeyboardInput("Give ppn(01..99): ");
sOutputFilename = sKeyboardInput + "_logfile.txt";

p_Welcome.present();
WaitForAllButtonpress();

##########################start experiment#################################
iRunAmountOfTrials = 6;
iTotalTrialCount = 1;

RunTrials();
                                # Run first block of trials

RunTrials();
                                # Run first block of trials

############ END ############
```

#SUBS file

```
################ CONDITION INFO SUBS ###############

Include "TemplateINFO.pcl";                    # Include all the variables

#-------------------------Get_Kbd_Input-------------------------------------
sub GetKeyboardInput(string inpstr1) begin

        t_Info1.set_caption(inpstr1);
        t_Info1.redraw();
        t_Info2.set_caption(" ");
        t_Info2.redraw();
        p_Info.present();
        system_keyboard.set_case_mode(3); # Hoofdletters mogelijk.
        sKeyboardInput = system_keyboard.get_input( p_Info, t_Info2 ); # Hier zit de backspace ingebakken.

end;

#####################################################

sub ShowText( string message ) begin

  t_Info1.set_caption( message );
  t_Info1.redraw();
  p_Info.present();

end;

#----------------------------WaitSubjectReady----------------------------------
sub WaitForAllButtonpress begin

  loop
                iCountOld = response_manager.total_response_count()
  until response_manager.total_response_count() > iCountOld
  begin
        end;

end;

#----------------------------WaitForAllButtonpressTime-------------------------------------
sub WaitForAllButtonpressTime( int iDurWait2 ) begin
```

```
  loop
                        iCountOld = response_manager.total_response_count();
                        iTimeStamp = clock.time();
                until ((response_manager.total_response_count() > iCountOld) || (clock.time() - iTimeStamp >= iDurWait2))
  begin
                end;

end;

#----------------------------WaitForAllButtonpress-----------------------------------
sub WaitForButtonpress( int iButton ) begin

  loop
                        iCountOld = response_manager.total_response_count(iButton)
  until response_manager.total_response_count(iButton) > iCountOld
  begin
                end;

end;

################### LOGFILE ################

#----------------------------WriteOutput-------------------

sub WriteTrialToOutputFile(string str ) begin

                TrialData = "";

                TrialData.append(string(iTotalTrialCount) + "\t");   #write variable followed by a tab
                TrialData.append(sOutputFilename + "\t");
                TrialData.append(string(iFixationTime[iTotalTrialCount]) + "\t");
                TrialData.append(string(iPictureTime[iTotalTrialCount][1]) + "\t");
                TrialData.append(string(iPictureTime[iTotalTrialCount][2]) + "\t");
                TrialData.append(string(iButtonPressed[iTotalTrialCount]) + "\t");
                TrialData.append(string(iButtonTime[iTotalTrialCount]) + "\t");
                TrialData.append(string(iSelectStimulus) + "\n");

  OutputFile.open_append ( str ); # append
  OutputFile.print( TrialData );
  OutputFile.close();

end;

################## TRIAL ###############

#----------------------------runtrials-----------een voor een presenteren-----------------------
sub RunTrials begin

                loop iTrialCount = 1
  until iTrialCount > iRunAmountOfTrials
  begin
                        p_Fixation.present();
                        iFixationTime[iTotalTrialCount] = clock.time();
                        wait_interval(992); #delay 1000 ms

                  iSelectStimulus = random(1,4);

    #select Flanker
                if iSelectStimulus == 1 then
                                        p_Flanker.add_part(t_FlankerHH, -28, 0);
                                        p_Flanker.add_part(t_FlankerHH, 28, 0);
     elseif iSelectStimulus == 2 then
                                        p_Flanker.add_part(t_FlankerSS, -28, 0);
                                        p_Flanker.add_part(t_FlankerSS, 28, 0);
     elseif iSelectStimulus == 3 then
                                        p_Flanker.add_part(t_FlankerSS, -28, 0);
                                        p_Flanker.add_part(t_FlankerSS, 28, 0);
     elseif iSelectStimulus == 4 then
                                        p_Flanker.add_part(t_FlankerHH, -28, 0);
                                        p_Flanker.add_part(t_FlankerHH, 28, 0);
     end;
                        p_Flanker.present();
```

```
                iPictureTime[iTotalTrialCount][1] = clock.time();

                wait_interval(92); #delay 100 ms

                #select target letter
                if iSelectStimulus == 1 then
                        p_Flanker.add_part(t_FlankerS, 0, 0);
                elseif iSelectStimulus == 2 then
                        p_Flanker.add_part(t_FlankerS, 0, 0);
                elseif iSelectStimulus == 3 then
                        p_Flanker.add_part(t_FlankerH, 0, 0);
                elseif iSelectStimulus == 4 then
                        p_Flanker.add_part(t_FlankerH, 0, 0);
                end;
                p_Flanker.present();
                iPictureTime[iTotalTrialCount][2] = clock.time();

                wait_interval(391); #400 ms
                default.present();

                #wacht op response
                WaitForAllButtonpressTime(1200);

                if (iPictureTime[iTotalTrialCount][2] + 1200) > clock.time() then
                        #a button has been pressed
                        iButtonPressed[iTotalTrialCount] = response_manager.last_response();
                        iButtonTime[iTotalTrialCount] = clock.time() - iPictureTime[iTotalTrialCount][2];
                else
                        #no button pressed
                        iButtonPressed[iTotalTrialCount] = 0;
                        iButtonTime[iTotalTrialCount] = 0;
                end;

                #ITI
                if (iButtonTime[iTotalTrialCount] <= 800 && iButtonTime[iTotalTrialCount] > 0) then
                        wait_interval(random(400,800)); #
                else
                        wait_interval(391); #400 ms
                end;

                p_Flanker.clear();
                WriteTrialToOutputFile(sOutputFilename);
                iTrialCount = iTrialCount + 1;
                iTotalTrialCount = iTotalTrialCount + 1;
   end;

end;

############ END ############
```

Got it? Let your timeline be checked.
This was the basic programming part. Let's continue with some crucial technical information. You don't have to 'know' all the details, but it is important that you get the idea.

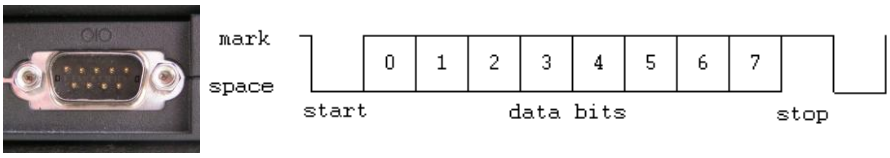
[8] Ports

[8-1] Ports: Description

Presentation can handle a lot of different external devices, e.g. serial and parallel ports and the BITSI.

The serial port

The serial port is a physical interface through which information transfers in or out one bit at a time. This serial port uses an RS-232 protocol. In RS-232, data is sent as a time-series of bits. Since transmitting and receiving data are separate circuits, the interface can operate in a full duplex manner, supporting concurrent data flow in both directions. The most common arrangement is an asynchronous link sending eight bits. When used in this way, the bit order consists of a start bit, eight data bits sent least significant bit first, an optional parity bit, and a stop bit.



serial port on a pc and configuration sending bits(byte)

Parallel port Output "outdated"

The parallel port is mostly used to send markers to EEG. Over a parallel port, binary information is transferred in parallel: each bit in a particular value is sent simultaneously as an electrical pulse across a separate wire, in contrast to a serial port, which requires each bit to be sent in series over a single wire.

A parallel printer port

Parallel port connectors usually have at least 25 pins, most of which are used, resulting in thick cables. These cables are also limited in length to a maximum of 3-8 meters, depending on the specific port and cable characteristics

BITSI box

The technical group of the Donders institute has developed an interface called BITSI. BITSI stands for Bits to Serial Interface. We use the BITSI box to send and receive trigger and response signals (markers) in EEG, MEG, fMRI and behavioral experiments. It's a simple building block that has a serial port and two eight bit ports: input and output. The two ports can be used for responses (input) and stimulus triggers (output). By using the serial port, the BITSI box can be used platform independent: it works on Windows, Linux and Mac OSX. Most programming environments and stimulus packages support serial communication.

The input port can be used to interface eight buttons maximally. Button presses are translated to serial output characters/bytes according to the following table:

Signal / Button	**Code** (rising / falling)
1	A / a

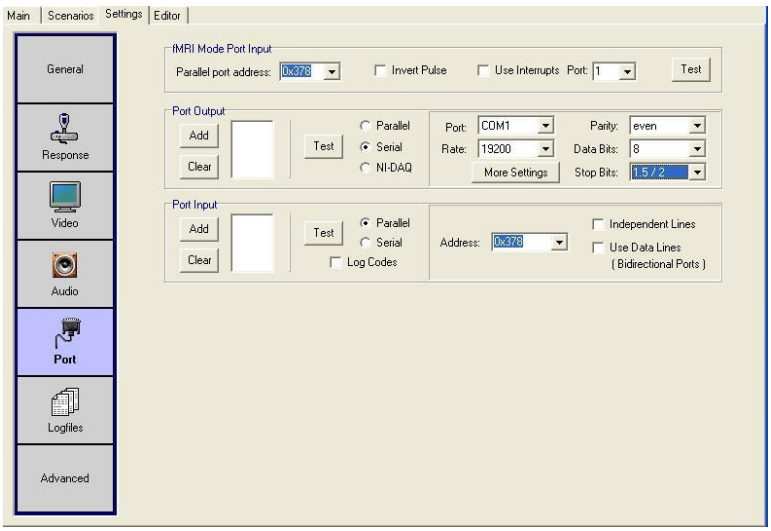
2	B / b
3	C / c
4	D / d
5	E / e
6	F / f
7	G / g
8	H / h

This means that when signal 1 gets high, a capital A will be sent to the serial port. A lowercase 'a' will be sent when the signal gets low again. Mechanical buttons can be connected directly. All inputs are debounced in software.

Every byte sent to the BITSI over the serial port, is represented at the 8 bit output.

[8-2] Port settings

To setup a port device, you have to click on ‘Port’ in the menu ‘Settings’ and then on ‘Add’ for Port output or Port input (see picture).



Serial port:

Speed

Serial ports use two-level (binary) signaling, so the data rate in bits per second is equal to the symbol rate in baud. Common bit rates per second for asynchronous start/stop communication are 300, 1200, 2400, 9600, 19200 baud, etc. The port speed and device speed must match.

The speed includes bits for framing (stop bits, parity, etc.) and the effective data rate is lower than the bit transmission rate. For example for 8-N-1 encoding only 80% of the bits are available for data (for every eight bits of data, two more framing bits are sent).

Parity

Parity is a method of detecting some errors in transmission. When parity is used with a serial port, an extra data bit is sent with each data character. These arranged in such a manner that the number of 1 bits in each character, including the parity bit, is always odd or always even. If a byte is received with the wrong number of 1 bits, then it must have been corrupted. If parity is correct there may have been no errors or an even number of errors. The parity of the serial protocol can be none, odd and even.

Stop bits

Stop bits are sent at the end of every byte transmitted in order to allow the receiving signal hardware to resynchronise. Electronic devices usually use one stop bit.

Parallel port

A parallel port can easily be added and has no specific settings.

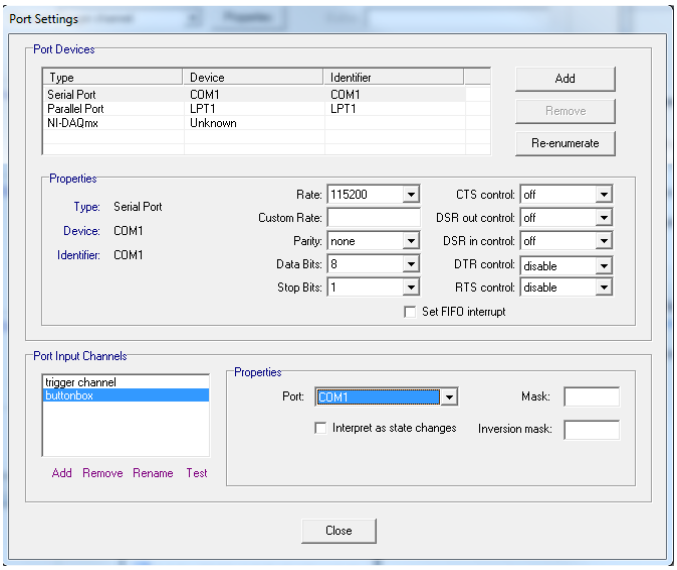
BITSI

The hardware design of the BITSI allows the device to be connected to the computer's USB or Serial port. When connected to the serial port, the use of an external power supply is required. In both cases, the BITSI is treated as a serial port and the following settings apply:

Baudrate	115200
Parity	none
Data bits	8
Stop bits	1
Flow control	none

How to add the BITSI

As an example, we explain how the BITSI is added as a port device.



1. Open the settings menu for 'Port' and choose 'Port Properties' at the bottom of the window
2. A new window occurs, called 'Port settings', with 4 fields in it: Port devices, Properties, Port Input channel and a second Properties. From the Port devices menu, select the correct serial port. If you don't know the correct port, remove the BITSI and reattach it, and look which Port has been added.
3. Now, you can adapt the values in the Properties-field. Fill in the values which are indicated above.
4. In the field 'Port input channels', choose 'Add' to add a new channel. Call this channel 'buttonbox'.
5. Choose in the properties field to the right the COM-port that you just adjusted.
6. You can test the BITSI by clicking on 'Test'. The button codes are sent to the screen.
7. You can close this window and select 'add at the output port field. Choose the newly created channel.(works also for the input port)
8. Move to the Response field in the settings menu.
9. Under Devices, right click and select 'Add port device'

10. Select under ‘Input channel’ the ‘buttonbox’ and give the new device a name.

When following this procedure, the device appears in the field ‘Devices’. When clicked on, you can select Buttons and add them to the Active buttons.

[8-3] Data Markers

Both types of ports can be used to send data markers. Data markers are used to synchronize stimuli events from your Presentation script with other devices, for example an EEG recording. When a marker is sent from Presentation, it is recorded time synchronized with the external device.

A data marker is defined by a number ranging between 1 and 255. This marker is sent to your data recording setup, and represents a single data point within your recorded data. Every event within your Presentation script can be associated with an individual marker, such that you can easily process your data after recording.

Code for sending markers:

In PCL code you can program a *handle* to send a marker:

```
#handle:
output_port OutputPort = output_port_manager.get_port( 1 );
```

To send a marker:

```
OutputPort.send_code(100);          #create a marker
```

Moreover, you can add code to your script, checking whether you have added the necessary port to your settings. An example for such code is this:

```
if (output_port_manager.port_count() < 1) then
  t_Info1.set_caption( "Forgot to add an output port!" );
  t_Info1.redraw();
  p_Info.present();
  loop until false begin end
end;
```

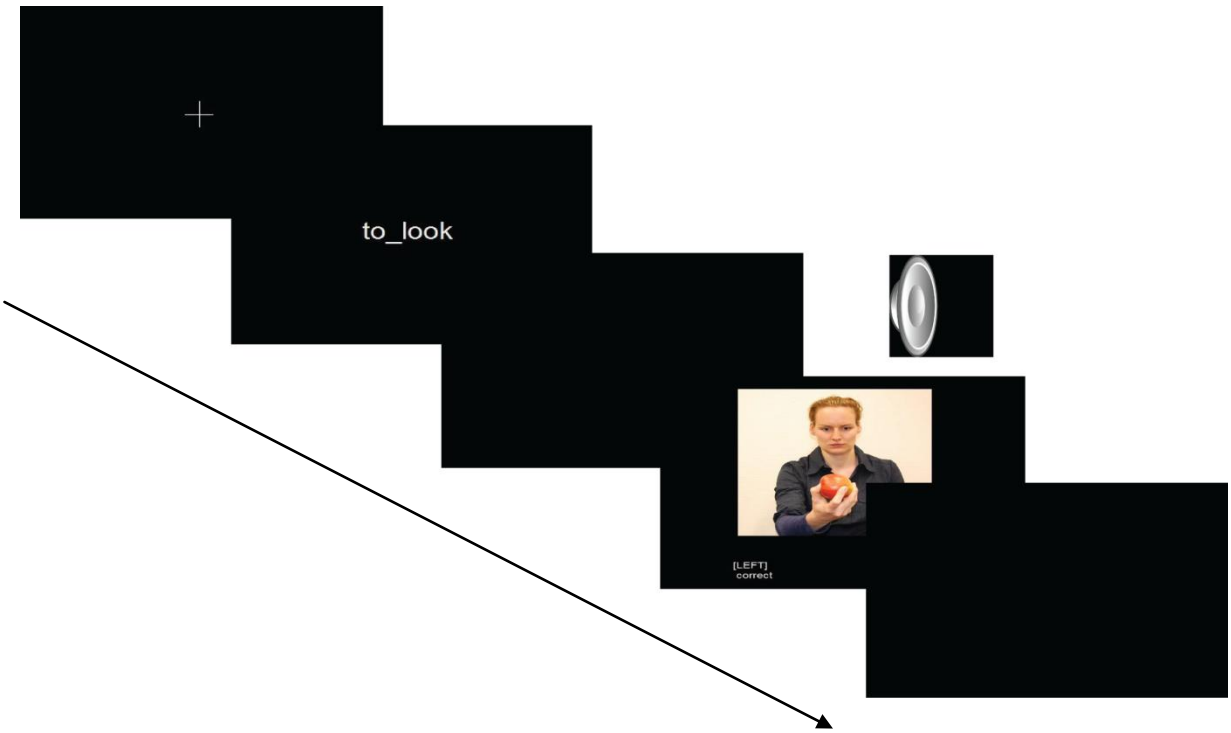
Assignment 8: Use the BITSI (optional)

1. Reuse the program of the red- green experiment. Change the reaction buttons of red and green to ‘a’ and ‘b’ from the BITSI button box. (see [8-2])
2. Send the code of the color of the box and the code of the pressed button to the BITSI. (See [8-3])

[9] Experiment Template

Important!
When you are sure you don't have any more questions, ask for the template to start with the final assignment on the next page

[9-1] Final assignment:



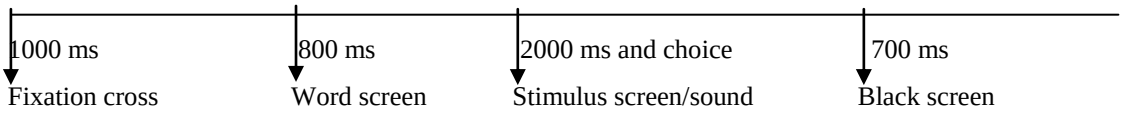
Experiment:

Does the prime match the end goal of the action, irrespective of the object involved?

Specifications:

1. Present fixation, Wait (1000)
2. Present prime for 800 ms ("to taste", "to smell", "to listen", "to look")
3. Black screen Wait (jitter 500 – 1500),
4. Present action picture for (2000 ms) with response
5. Play correct or false sound
6. Black screen for 700ms

Time line:



[A] Use the template to make a well - structured experiment.

[B] Use the timeline to build up the experiment bit by bit, start with presenting a fixation cross.

[C] Create a 1000ms delay and use the frequency of the monitor to create the correct time slot

[D] Build the prime screen in SDL(only build the screen, later PCL will implement the correct words)

[E] present the word screen for 800mS

[F] present the black screen with a jitter of 500-1500ms

[G] Create a screen with a **bitmap** picture on it, and **text** "correct" and "incorrect" on the same screen.

[H] Present this picture for 2000ms.

[I] Add the keyboard "y" and "n" in **settings->response -> active buttons**

[J] Add the buttons in SDL **active_buttons** and **button_codes**

[K] Make sure the presentation of the picture ends when one of the buttons is pressed, ór after 2000ms have passed.

[L] Create an incorrect and a correct **sound** in SDL

[M] If a button was pressed after the picture was presented play a sound at the time the button was pressed

[N] Present a black screen for 700 ms

[O] Find the already made tab delimited file(Templatestim.txt) the trials are defined in here.

[P] Read in a tab delimited tab file with the subroutine "**Get_stimulus_file**"

[Q] Replace the words on the word screen with the words you got from the tab delimited file

[R] Change the bitmaps on the picture screen with the picture names you got from the tab delimited file

[S] Check with the input files last row(1 or 2) if the answers given to the picture are correct and give feedback with a correct or incorrect sound

[T] Log which button is pressed and the time the button is pressed.

[U] Log all the times when pictures are presented

[V] Write all the logged data to the hard disk **WriteOutput( )**

[W] Check if the output file is correct. If you think the experiment is working correctly, ask one of the teachers to check it too.

Appendix

Programming convention

- Stimuli definitions in SDL begin with the initial lowercase letter to indicate the type followed by an underscore

  p_Picture (p = picture); w_Beep (w = wav) ; b_Box1 (b=box);

- Variables in PCL begin with the lowercase letter to indicate the type of the variable, followed by a name starting with an uppercase letter

  iCount (i = integer) ; sName (s = string); adGrades (ad = array doubles);

- In Loops and Subroutines the code between begin and end is indented by a tab

```
sub Wait(int iDurWait)
begin
	loop
		iTimeStamp = clock.time();
	until (clock.time() - iTimeStamp >= iDurWait)
	begin
	end;
end;
```

- Each new word of a name of a subroutine starts with a capital letter

  WaitForButtonpress(); ExportDataToLogfile();

Timing tests:

In presentation all inputs (keyboard, mouse, gameport, serial port, parallel port) can be checked by directly polling them. This allows for the most accurate timing possible (ms).

(1) Keyboards:

Keyboards (PS2 and USB) are known to have poor timing accuracy. Usually these are in the range of several 100th of a second.

Test results:

We tested the timing of a Logitec USB keyboard in Windows XP SP3 using optical tracking. Our results revealed a timing accuracy between 20 and 26 ms.

(2) Mouse

On most OS, USB mice are polled at a rate of 8 ms. Mice with special drivers might be set to poll more often.

Test results:

We tested the mouse accuracy of a standard USB mouse on Windows XP SP3 by measuring the time between reported position changes.
Our results revealed: The expected standard accuracy of 8 ms.
Using a Logitec G500 USB mouse with a dedicated driver, polling rates could be reduced, leading to an increased accuracy of 1 ms.

(3) Serial port:

The serial port is very accurate and thus suited for timing accurate measurements. If a computer does not have a serial port, USB-to-serial converter can be used (e.g. from Sweex or Keyspan). However, the timing accuracy of these depends on the implementation and drivers used! USB has to time share their ports, be aware how this is implemented on your computer. Big timing differences have been found.
It is important to deactivate any additional FIFO buffers or delays, provided by the port driver!

Test results:

We tested the timing of a UART 16550A serial port (a real one, not a USB-to-serial converter!) on Windows XP SP3 by sending a byte to a connected loopback device which immediately sends the byte back. We then measured the time between sending and receiving. We repeated this process 1000 times.
Our results revealed:
-With a baudrate of 115200, the maximal measured time between sending and receiving a byte was 0.28 ms.
-With a baudrate of 19200, the maximal measured time between sending and receiving a byte was 0.68 ms.

(4) Parallel port

The parallel port works by directly applying a current (writing) and measuring if a current is applied (sending) to several pins on the connector. Time delay is too short to measure.