Kofiko – notes for developers

Instructions on how to add a new paradigm

The first thing we would like to do is to add a new structure in the Kofiko XML file (typically located at \config\default.xml) . One can “copy paste” an existing paradigm and just modify the function names accordingly.

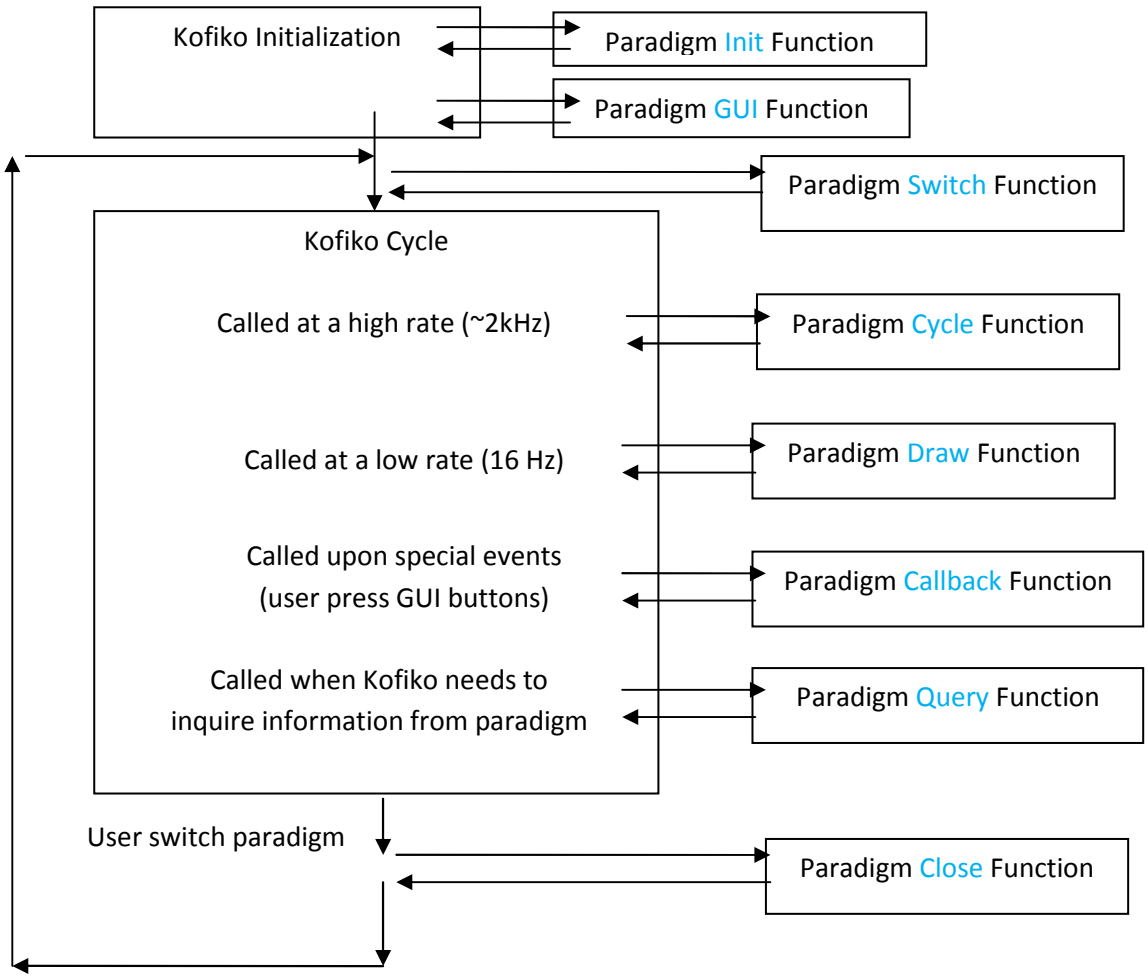
For example:

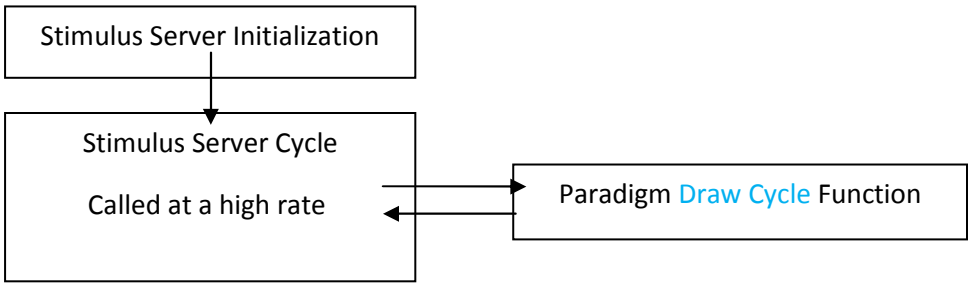
```
<Paradigm Name = "fMRI Block Design"
        Init = "fnParadigmBlockDesignInit"
        Cycle = "fnParadigmBlockDesignCycle"
        Draw = "fnParadigmBlockDesignDraw"
        DrawCycle = "fnParadigmBlockDesignDrawCycle"
        Callbacks = "fnParadigmBlockDesignCallbacks"
        GUI = "fnParadigmBlockDesignGUI"
        Close = "fnParadigmBlockDesignClose"
        ParadigmSwitch = "fnParadigmBlockDesignParadigmSwitch"
        Query = "fnParadigmBlockDesignQuery"
        StrobeCodes = "BlockDesignStrobeCodes.txt"

        Initial_JuiceTimeMS = "35"
        Initial_GazeTimeMS = "3000"
        Initial_BackgroundColor = "128 128 128"
        Initial_FixationSizePix = "7"
        Initial_GazeBoxPix = "150"
        Initial_StimulusSizePix = "128"
        Initial_StimulusON_MS = "200"
        Initial_StimulusOFF_MS = "200"
   > </Paradigm>
```

This defines a paradigm called “fMRI Block Design”. The first part includes a set of mandatory entries (function names) that are required from ALL paradigms. The second part defines several default values that will be used whenever Kofiko boots up.

To better understand why so many functions are needed, let us take a look at the following flow chart:





Let us now go over each of the required functions. When kofiko starts, many initializations are required. Among them, a paradigm-specific initialization which is only called once. This function should initialize all variables related to the paradigm (potentially, using the XML file). The way this is done is by updating a global variable called g_strctParadigm. For example, a typical Init code looks like this:

```
function fnParadigmBlockDesignInit()
%
% Copyright (c) 2008 Shay Ohayon, California Institute of Technology.

global g_strctParadigm g_strctStimulusServer

% This variable MUST be present in a paradigm
g_strctParadigm.m_iMachineState = 0;

iSmallBuffer = 100;

% Here we add "Timestamped" variables.
g_strctParadigm = fnTsAddVar(g_strctParadigm, 'JuiceTimeMS',
g_strctParadigm.m_fInitial_JuiceTimeMS, iSmallBuffer);
g_strctParadigm = fnTsAddVar(g_strctParadigm, 'GazeTimeMS',
g_strctParadigm.m_fInitial_GazeTimeMS, iSmallBuffer);
g_strctParadigm = fnTsAddVar(g_strctParadigm, 'CurrStimulusIndex', 0,
iLargeBuffer);
g_strctParadigm = fnTsAddVar(g_strctParadigm, 'GazeBoxPix',
g_strctParadigm.m_fInitial_GazeBoxPix, iSmallBuffer);
g_strctParadigm = fnTsAddVar(g_strctParadigm, 'StimulusSizePix',
g_strctParadigm.m_fInitial_StimulusSizePix, iSmallBuffer);
g_strctParadigm = fnTsAddVar(g_strctParadigm, 'ImageList', '',20);

if ~isempty(g_strctParadigm.m_strInitial_DefaultImageList)
    g_strctParadigm = fnTsSetVar(g_strctParadigm, 'ImageList',
      g_strctParadigm.m_strInitial_DefaultImageList);
    mssend(g_strctStimulusServer.m_iSocket,
{'LoadImageList',g_strctParadigm.m_strInitial_DefaultImageList});
    fnLOG('Loading images locally on Kofiko');
    acFileNames =
fnInitializeTextures(g_strctParadigm.m_strInitial_DefaultImageList);
    g_strctParadigm = fnTsAddVar(g_strctParadigm, 'ImageFileList',
acFileNames,20);
end;
```

Let's go over some of the important concepts implemented here. First, we define the g_strctParadigm. All paradigm related information should ALWAYS be kept in this global structure. This structure must also have a variable called `m_iMachineState`, which represents the current state of the paradigm finite state machine (FSM).

Another important concept that we can see here is the usage of time-stamped variables. If we have tunable parameters and we want to keep track when they were changed, we can use an encapsulation that keeps the value and time-stamp using fnTsAddVar, fnTsGetVar and fnTsSetVar. In this example, I am adding several variables, such as "JuiceTimeMS",

"GazeTimeMS", and so on, and initializing them to default values that have been defined in the paradigm xml file. fnTsAddVar adds a structure with a defined buffer size. If the user adds more entries that the initial buffer size, it will be automatically doubled.

Another thing we see here is the usage of the global variable `g_strctStimulusServer`. This variable holds various information about the stimulus server, such as the resolution, refresh rate, and the port needed to communicate with the server (i.e., send messages).
In this example, the init function checks whether the user has supplied a default image list, and if so, it sends a message to the stimulus server, asking it to load that image list to memory:

```
mssend(g_strctStimulusServer.m_iSocket,
{'LoadImageList',g_strctParadigm.m_strInitial_DefaultImageList});
```

In addition, it also initializes the images locally, by calling `fnInitializeTextures`.

The next step is when Kofiko generates the graphical user interface (GUI) for each paradigm. This is the part where the user can specify various buttons , edit boxes, and scrollers.
Let's take a look at the GUI function for this paradigm:

```
function fnParadigmBlockDesignGUI()
%
% Copyright (c) 2008 Shay Ohayon, California Institute of Technology.

global g_strctParadigm

% Note, always add controllers as fields to
g_strctParadigm.m_strctControllers

[hParadigmPanel, iPanelHeight, iPanelWidth] = fnCreateParadigmPanel();
strctControllers.m_hPanel = hParadigmPanel;
iNumButtonsInRow = 3;
iButtonWidth = iPanelWidth / iNumButtonsInRow - 20;

% Add buttons
strctControllers.m_hLoadList = uicontrol('Parent',hParadigmPanel,
'Style', 'pushbutton', 'String', 'Load Image List',...
     'Position', [5 iPanelHeight-60 iButtonWidth 50], 'Callback',
[g_strctParadigm.m_strCallbacks,'(''LoadList'');']);

% Add combo text slider edit controller
strctControllers = fnAddTextSliderEditCombo(strctControllers, 90, ...
    'Gaze Time(ms):', 'Gaze',iPanelHeight, iPanelWidth, 30, 10000, [1,
50], fnTsGetVar(g_strctParadigm,'GazeTimeMS'));

g_strctParadigm.m_strctControllers = strctControllers;
return;
```

In this example, the paradigm first generates a panel (a space on screen where all user controllers will be placed). It then adds a push button, and a Text-Slider-Edit combo that will be used to control the gaze time. There are many parameters that are used here, such as the position on the screen, initialization value, text to display on the screen etc. Notice that we init the combo controller using information from the paradigm. That is – the current gaze time, by taking the most updated data entry using fnTsGetVar

The important thing to realize here is that whenever the user will press the add list button, a callback will be generated. This means that Kofiko will call the paradigm callback function with a parameter called "LoadList". Similarly, if the user changes the value of the gaze time, a callback will be called with the parameter "GazeEdit", or "GazeSlider" (depending whether the user changed the value in the edit box or shifted the slider. Each one of these events need to be handled (and coded accordingly in the paradigm callback function)

Let us take a look at the callback function:

```
function fnParadigmBlockDesignCallbacks(strCallback)
%
% Copyright (c) 2008 Shay Ohayon, California Institute of Technology.

global g_strctParadigm g_strctStimulusServer

switch strCallback
    case 'LoadList'
       mssend(g_strctStimulusServer.m_iSocket,
      {'PauseButRecvCommands'});
        fnHidePTB();
      [strFile, strPath] =
      uigetfile([g_strctParadigm.m_strInitial_DefaultImageFolder,'*.txt
      ']);
       fnShowPTB()
        if strFile(1) ~= 0
            g_strctParadigm.m_strNextImageList = [strPath,strFile];
            g_strctParadigm.m_iMachineState = 6; %
      end

    case 'GazeEdit'
        strTemp =
      get(g_strctParadigm.m_strctControllers.m_hGazeEdit,'string');
        iNewGazeTimeMS = str2num(strTemp);
        if ~isempty(iNewGazeTimeMS)
      fnUpdateSlider(g_strctParadigm.m_strctControllers.m_hGazeSlider,
      iNewGazeTimeMS);
      g_strctParadigm =
            fnTsSetVar(g_strctParadigm,'GazeTimeMS',iNewGazeTimeMS);
            fnDAQWrapper('StrobeWord', fnFindCode('Gaze Time
            Changed'));
            fnLOG('Setting gaze to %d', iNewGazeTimeMS);
        end;
end;

return;
```

The first part handles the event whenever the user wants to load a new image list. The second part is takes care of the event when the user edits the gaze time number. It first obtains the new value from the controller, update the slider controller (so they are in sync) and then update the paradigm variable that holds this value using fnTsSetVar. It then sends a strobe word to plexon that represents this event and logs the event in Kofiko log file.

The most important function of a paradigm is the cycle function. This function is called at a very high rate, and thus, need to return as fast as possible without blocking. This function implements the FSM that controls the behavior. Let us take a look at the cycle function:

```
function [strctOutput] = fnParadigmBlockDesignCycle(strctInputs)
%
% Copyright (c) 2008 Shay Ohayon, California Institute of Technology.

global g_strctParadigm g_strctStimulusServer

fCurrTime = GetSecs;

switch g_strctParadigm.m_iMachineState
    case 1 % Run some tests that everything is OK. Then goto 2
    case 2
end;

%% Reward related stuff
if g_strctParadigm.m_iMachineState > 0 % i.e., we are running
     pt2iFixationSpotPix =
g_strctParadigm.FixationSpotPix.Buffer(:,:,g_strctParadigm.FixationSpot
Pix.BufferIdx);
     iGazeBoxPix =
g_strctParadigm.GazeBoxPix.Buffer(:,:,g_strctParadigm.GazeBoxPix.Buffer
Idx);
    aiGazeRect = [pt2iFixationSpotPix-
iGazeBoxPix,pt2iFixationSpotPix+iGazeBoxPix];

    bInsideGazeRect = strctInputs.m_pt2iEyePosScreen(1) > aiGazeRect(1)
&& ...
        strctInputs.m_pt2iEyePosScreen(2) > aiGazeRect(2) && ...
        strctInputs.m_pt2iEyePosScreen(1) < aiGazeRect(3) && ...
        strctInputs.m_pt2iEyePosScreen(2) < aiGazeRect(4);
    if ~bInsideGazeRect
        g_strctParadigm.m_fInsideGazeRectTimer = fCurrTime;
    end;
   iGazeTimeMS =
g_strctParadigm.GazeTimeMS.Buffer(g_strctParadigm.GazeTimeMS.BufferIdx)
;
    if fCurrTime - g_strctParadigm.m_fInsideGazeRectTimer > iGazeTimeMS
/ 1000
        fnParadigmToKofikoComm('Juice',
g_strctParadigm.JuiceTimeMS.Buffer(g_strctParadigm.JuiceTimeMS.BufferId
x));
        g_strctParadigm.m_fInsideGazeRectTimer = fCurrTime;
        g_strctParadigm.m_afCorrectTrial =
[g_strctParadigm.m_afCorrectTrial, fCurrTime];
    end;
end;

strctOutput = strctInputs;
return;
```

This is a degenerate version of a cycle function since the machine state never changes. However, this is typically where you would implement the logic of your paradigm. i.e., if something happens go to state X. If monkey fixates go to state Y, etc.
The bottom part represents a short code that implements the logic behind the juice reward. It gets the fixation spot, eye position, gaze area rect and time needed to fixate to get a reward and if everything is in order (i..e, monkey has fixated for enough time inside the gaze area rect), then the paradigm informs Kofiko to give juice reward (`fnParadigmToKofikoComm('Juice',..)` by opening the valve for certain amount of time.

The draw function that is executed on Kofiko is being called not by the Cycle function, but from Kofiko at a low rate. This means, that in any given moment, Draw needs to know what to display on screen. Typically, this is done by keeping a variable that holds what is the current stimulus. The draw function should never call Flip, because screen updates on Kofiko are always async' to keep the high cycle rate.

```
function fnParadigmBlockDesignDraw()
%
% Copyright (c) 2008 Shay Ohayon, California Institute of Technology.


global g_strctPTB g_strctParadigm g_strctStimulusServer

pt2iCenter = g_strctStimulusServer.m_aiScreenSize(3:4)/2;

aiFixationRect = [pt2iCenter-3 pt2iCenter+3];

Screen('FillArc',g_strctPTB.m_hWindow,[255 255 255],
aiFixationRect,0,360);

fGazeBoxPix = fnTsGetVar(g_strctParadigm, 'GazeBoxPix');
aiGazeRect = [pt2iCenter-fGazeBoxPix,pt2iCenter+fGazeBoxPix];
Screen('FrameRect',g_strctPTB.m_hWindow,[255 0 0], aiGazeRect);

fStimulusSizePix = fnTsGetVar(g_strctParadigm, 'StimulusSizePix');
aiStimulusRect = [pt2iCenter-fStimulusSizePix,
pt2iCenter+fStimulusSizePix];

Screen('FrameRect',g_strctPTB.m_hWindow,[255 255 0], aiStimulusRect);

iCurrStimulusIndex = fnTsGetVar(g_strctParadigm, 'CurrStimulusIndex');
if iCurrStimulusIndex > 0
    % Missing code here….
end
```

In this example, what is being displayed on the screen (in the Kofiko machine) is just the gaze box, the stimulus size box and the fixation point (at the center of the screen).

The DrawCycle function, on the other hand, is the function that handles drawing on the stimulus server. This function is assumed to be very accurate and uses sync' flips.

Here is a short code snippet from a DrawCycle function. The first part handles the communication from Kofiko. It gets messages, such as LoadImageList, or "StartDisplay" and handles them accordingly.

```
function fnParadigmBlockDesignDrawCycle(acInputFromKofiko)
%
% Copyright (c) 2008 Shay Ohayon, California Institute of Technology.

global g_strctPTB g_strctDraw g_strctNet g_strctServerCycle

fCurrTime = GetSecs();

if ~isempty(acInputFromKofiko)
    strCommand = acInputFromKofiko{1};
    switch strCommand
        case 'LoadImageList'
            strImageList = acInputFromKofiko{2};
            fnInitializeTextures(strImageList);
            g_strctServerCycle.m_iMachineState = 0;
            g_strctDraw.m_iNextImageToShow = 1;
        case 'StartDisplay'
            g_strctServerCycle.m_iMachineState = 1;
    end
end;
```

The second part takes care of the finite state machine that displays the stimulus with high accuracy:

```
switch g_strctServerCycle.m_iMachineState
    case 0
        % Do nothing
    case 1
        mssend(g_strctNet.m_iCommSocket, {'DisplayStarted'});

        Screen('FillRect',g_strctPTB.m_hWindow,
g_strctServerCycle.m_strctDrawParams.m_afBackgroundColor);
        aiFixationRect =
[g_strctServerCycle.m_strctDrawParams.m_pt2fFixationSpotPix-
g_strctServerCycle.m_strctDrawParams.m_fFixationSizePix,...

g_strctServerCycle.m_strctDrawParams.m_pt2fFixationSpotPix+g_strctServe
rCycle.m_strctDrawParams.m_fFixationSizePix];
 …

        aiStimulusRect = [iStartX, iStartY, iEndX, iEndY];
        Screen('DrawTexture', g_strctPTB.m_hWindow,
g_strctPTB.m_ahTextures(g_strctDraw.m_iNextImageToShow),[],aiStimulusRe
ct, g_strctServerCycle.m_strctDrawParams.m_fRotationAngle);

        Screen('FillRect',g_strctPTB.m_hWindow,[255 255 255], ...
            [g_strctPTB.m_aiRect(3)-
g_strctServerCycle.m_strctDrawParams.m_iPhotoDiodeWindowPix ...
            g_strctPTB.m_aiRect(4)-
g_strctServerCycle.m_strctDrawParams.m_iPhotoDiodeWindowPix ...
            g_strctPTB.m_aiRect(3) g_strctPTB.m_aiRect(4)]);

    % Draw Fixation spot
        Screen('FillArc',g_strctPTB.m_hWindow,[255 255 255],
aiFixationRect,0,360);

        g_strctServerCycle.m_fLastFlipTime =
Screen('Flip',g_strctPTB.m_hWindow); % This would block the server
until the next flip.
        mssend(g_strctNet.m_iCommSocket,
{'FlipON',g_strctDraw.m_iNextImageToShow});
        g_strctServerCycle.m_iMachineState = 2;
```

In this example, state 0 means that we display nothing. State 1 means we start displaying an image. Remember that this code runs on the stimulus server, thus, if it wants to inform Kofiko about various events (such as, image was displayed, and so on), it needs to transfer this data over the network using mssend function.
In this example, the stimulus is displayed and the exact time the image was flipped to the screen is saved in `g_strctServerCycle.m_fLastFlipTime`
The FSM then moves to state 2, where it waits until the required amount of time has elapsed and then turns off the stimulus:

```
   case 2
        if (fCurrTime - g_strctServerCycle.m_fLastFlipTime) >
g_strctServerCycle.m_strctDrawParams.m_fStimulusON_MS/1e3 - (0.2 *
(1/g_strctPTB.m_iRefreshRate) )
            % Turn stimulus off
            Screen('FillRect',g_strctPTB.m_hWindow,
g_strctServerCycle.m_strctDrawParams.m_afBackgroundColor);

            aiFixationRect =
[g_strctServerCycle.m_strctDrawParams.m_pt2fFixationSpotPix-
g_strctServerCycle.m_strctDrawParams.m_fFixationSizePix,...

g_strctServerCycle.m_strctDrawParams.m_pt2fFixationSpotPix+g_strctServe
rCycle.m_strctDrawParams.m_fFixationSizePix];

            Screen('FillArc',g_strctPTB.m_hWindow,[255 255 255],
aiFixationRect,0,360);

            Screen('FillRect',g_strctPTB.m_hWindow,[0 0 0], ...
                [g_strctPTB.m_aiRect(3)-
g_strctServerCycle.m_strctDrawParams.m_iPhotoDiodeWindowPix ...
                g_strctPTB.m_aiRect(4)-
g_strctServerCycle.m_strctDrawParams.m_iPhotoDiodeWindowPix ...
                g_strctPTB.m_aiRect(3) g_strctPTB.m_aiRect(4)]);
            Tmp = g_strctServerCycle.m_fLastFlipTime;
            g_strctServerCycle.m_fLastFlipTime = Screen('Flip',
g_strctPTB.m_hWindow); % Block.

            mssend(g_strctNet.m_iCommSocket, {'FlipOFF'});
            g_strctServerCycle.m_iMachineState = 3;
    case 3
        if (fCurrTime - g_strctServerCycle.m_fLastFlipTime) > ...

(g_strctServerCycle.m_strctDrawParams.m_fStimulusOFF_MS)/1e3 - (0.2 *
(1/g_strctPTB.m_iRefreshRate) )
                mssend(g_strctNet.m_iCommSocket, {'DisplayFinished'});
                 g_strctServerCycle.m_iMachineState = 0;
        end
```

Notice that after we turn off the stimulus, we send a message to Kofiko, and then change the machine state to 3, which represents the wait period until the “OFF” interval is finished. We then go back to state zero, which means we wait for a new “StartDisplay” Command.

The remaining paradigm functions we have not discussed are the Query, Switch and Close.

The Query function is a general mechanism that allows Kofiko->Paradigm communication:

```
function acAnswer = fnParadigmPassiveFixationQuery(strQuery)
%
% Copyright (c) 2008 Shay Ohayon, California Institute of Technology.
% This file is a part of a free software. you can redistribute it

global g_strctParadigm

acAnswer = [];
switch strQuery
    case 'NumStimuli'
        acAnswer = g_strctParadigm.m_iNumStimuli;
    case 'CategoryNames'
        acAnswer = g_strctParadigm.m_acCatNames;
end

return;
```

For example, Kofiko has support to display PSTH. However, it needs to know how many stimuli are used by the paradigm. Therefore, it can call the paradigm query function with a parameter "`NumStimuli`".

The Paradigm switch function is used to re-initialize various structures when we switch BACK to a paradigm. Normally, this function should be empty. However, there are special cases in which specific re-initialization is in order. For example, when many textures have been allocated by PTB. These textures need to be re-allocated when we switch back to the paradigm.

Similarly, the "Close" Function should also be empty, unless some special "closing" or "de-allocation" is in order.

Kofiko Data Structures

When Kofiko shuts down, it saves all paradigm related variables, and non-paradigm related variables into a single .MAT file. Below is a short description for some of the fields. The most important ones are colored.

If, for example, we were to load "strctKofiko = load('100109_133934_Rocco.mat')", for example:

strctKofiko =

g_astrctAllParadigms: {[1x1 struct] [1x1 struct] [1x1 struct] [1x1 struct] [1x1 struct]}

g_strctDAQParams: [1x1 struct]

g_strctLog: [1x1 struct]

g_strctSystemCodes: [1x1 struct]

g_strctAppConfig: [1x1 struct]

g_strctEyeCalib: [1x1 struct]

g_strctStimulusServer: [1x1 struct]

g_astrctAllParadigms is a cell array that contains all the paradigms that were loaded to Kofiko. Each paradigm should have all the time-stamped variables. For example,

strctKofiko.g_astrctAllParadigms{2}.JuiceTimeMS has two members:

Buffer and TimeStamp, that hold the data and when it was changed.

The strctDAQParams contains various triggered events.

strctKofiko.g_strctDAQParams.LastStrobe , for example, contains all the strobe words that were sent to Plexon.

strctKofiko.g_strctDAQParams.m_astrctExternalTriggers(1).Trigger, contains external triggers (for example, ones we get from the MRI scanner)

The number of recorded sessions, and their corresponding time stamps, can be retrieved, for example using this code:

```
aiKofikoStartInd = find(
      strctKofiko.g_strctDAQParams.LastStrobe.Buffer ==
            strctSystemCodes.m_iStartRecord);
aiKofikoStartRecTS =
strctKofiko.g_strctDAQParams.LastStrobe.TimeStamp(aiKofikoStartInd);
```

strctKofiko.g_strctStimulusServer, contains information about the stimulus server (screen resolution, refresh rate, etc).

strctKofiko.g_strctEyeCalib holds information about the eye calibration used. It has five fields: CenterX, CenterY, GainX, GainY, which represent the linear transformation that converts raw voltages to screen pixel values. All these values are time-stamped. In addition, the raw-eye signal can be saved to the file as well. In that case, it is stored in g_strctEyeCalib.EyeRaw