



# **Bits# Technical Manual**

**Version R08**

# Table of Contents

---

<b>Chapter 1: About this Manual</b>	<b>6</b>
1.1 Syntax Notation Style	6
<b>Chapter 2: What's in the Bits# package?</b>	<b>7</b>
<b>Chapter 3: Bits# connections</b>	<b>9</b>
<b>Chapter 4: Bits# installation</b>	<b>11</b>
4.1 Hardware Requirements	11
4.2 Cable connections	12
4.3 Operating System Support	14
4.4 Operating System Configuration	15
4.5 Installing Using Windows Vista or Windows 7	16
4.6 Installing Using Mac OS X	19
4.7 Installing Using Linux	21
4.8 Communicating with Bits# using Windows	22
4.9 Communicating with Bits# using Mac	26
4.10 Communicating with Bits# using Linux	29
4.11 Communicating with Bits# using MATLAB	31
<b>Chapter 5: Introduction</b>	<b>35</b>
5.1 Overview	35
5.2 Bits	35
5.3 Pixel Values	36
5.4 Bits# Video Modes	37
5.5 Look-up table (LUT)	39
5.6 Formatting 8-bit into 14-bits representation	40
5.7 T-Lock code	42
5.8 Digital Pulse Triggers	43
5.9 Gamma Correction	44
5.10 Temporal Dithering	44
5.11 Response Box	45

---

<b>Chapter 6: Using Bits# with an LCD screen</b>	<b>47</b>
6.1 Overview	47
6.2 Bits# Temporal Dithering	47
<b>Chapter 7: Screen Calibration and Gamma Correction</b>	<b>51</b>
7.1 Overview	51
7.2 Taking Screen Calibration Measurements	52
7.3 Using measured luminance to create a Gamma Correction LUT	55
7.4 Applying Gamma Correction	57
7.5 Nullifying the computer graphics card LUT (using Psychtoolbox)	57
7.6 Using the ColorCAL MKII	58
7.7 Calculating a Gamma Correction LUT using MATLAB	62
7.8 Saving a 13-bit LUT to Bits# Hardware	66
<b>Chapter 8: Creating a Colour Look-up Table (cLUT)</b>	<b>69</b>
8.1 Overview	69
8.2 Creating a colour look-up table (cLUT) T-Lock	69
8.3 Creating the cLUT T-Lock Matrix using MATLAB	71
8.4 Using a cLUT T-Lock Matrix with MATLAB	74
8.5 Using the cLUT T-Lock Matrix with Psychtoolbox	75
<b>Chapter 9: Mono++ Mode</b>	<b>77</b>
9.1 Overview	77
9.2 Using Mono++ mode with MATLAB	78
9.3 Using Mono++ mode with Psychtoolbox	80
9.4 Drawing in Mono++ mode with Psychtoolbox	82
<b>Chapter 10: COLOUR++ Mode</b>	<b>85</b>
10.1 Overview	85
10.2 Formatting Colour++ using MATLAB	87
10.3 Using Colour++ mode with Psychtoolbox	88
10.4 Drawing in Colour++ mode with Psychtoolbox	90
<b>Chapter 11: Bits++ Mode</b>	<b>93</b>
11.1 Overview	93

---

11.2 Formatting Bits++ using MATLAB .....	94
11.3 Using Bits++ mode with Psychtoolbox .....	97
11.4 Drawing in Bits++ mode with Psychtoolbox .....	99
<b>Chapter 12: Communication and Synchronisation with external equipment .....</b>	<b>101</b>
12.1 Overview .....	101
12.2 Important Considerations .....	101
12.3 The T-Lock Code .....	103
12.4 Creating a trigger pulse using MATLAB .....	106
12.5 Creating a Trigger Pulse using Psychtoolbox .....	111
<b>Chapter 13: Stereo Display .....</b>	<b>113</b>
13.1 Overview .....	113
13.2 Specifying Stereo Image Matrices with MATLAB .....	116
13.3 Displaying Alternate Line (interleaved) Stereo Images using MATLAB .....	118
13.4 Displaying Interleaved Stereo Images using Psychtoolbox .....	119
13.5 Displaying Alternate Frame Stereo Images using MATLAB .....	120
13.6 Displaying Alternate Frame Stereo Images using Psychtoolbox .....	121
<b>Chapter 14: Analogue Data .....</b>	<b>123</b>
14.1 Overview .....	123
14.2 Outputting Analogue Signals .....	123
14.3 Receiving Analogue Signals .....	123
<b>Chapter 15: Response Box .....</b>	<b>125</b>
15.1 Overview .....	125
15.2 Setup .....	125
15.3 Using a response box with MATLAB .....	128
<b>Chapter 16: Appendix .....</b>	<b>133</b>
16.1 Using a Custom .edid File .....	133
16.2 CDC protocol for Bits# .....	133
16.3 Single-Link DVI frame rates and resolutions .....	137
16.4 Digital I/O Connector Pinout .....	138

## Document Revision History

Version	Date	Firmware version	Description of changes
1.00	10/ 2011		Initial draft.
1.01	12/ 2011		New sections and layout - Unreleased
1.02	02/ 2012		Edits
1.03	02/2013	2012/11/20	Revised
1.04			
1.05			
1.06	09/2013		Reformat, moved over to flare
1.07	03/2014	2012/11/20	Revised
1.08	06/2014	2012/11/20	removed hyphenation and corrected layout errors
R06	08/2014	2012/11/20	Corrected the description for encoding analogue output in the T-lock code. Changed version no. to R-type.
R07	02/2015	2012/11/20	Removed duplicate information about Analogue Out
R08	10/2015		Corrected information about Analogue Out and Trig Out encoding.

# Chapter 1: About this Manual

Bits# (pronounced bits sharp) replaces its predecessor, the Bits++ (pronounced bits plus plus). This system line converts an ordinary computer or laptop into a high dynamic range, calibrated visual stimulator and frame-synchronous data acquisition system. The Bits# design provides essential stimulus calibration and synchronisation features for vision research.

## 1.1 Syntax Notation Style

This manual includes many examples of code that can be used with MATLAB. Anything preceded by a '`>>`' is a command that can be entered, either at the command prompt or within a script.

If a function is included, such as those supplied by Psychtoolbox, there are several types of input arguments:

'DrawTexture' – If an input argument is in normal font and surrounded by single quotation marks, this is a string input argument and should be entered exactly as shown (including quotation marks).

myCLUT – An argument not in single quotation marks but in normal font is a variable name assigned elsewhere in the example being described. These variable names can be changed, however ensure they are changed consistently in every position used, including when they are first defined.

windowHandle – If an input argument is in italicized font, it is a compulsory argument, however they are not assigned values within the examples given at the time. It should be changed to the appropriate value or variable name, depending on your uses.

[, mode] – An input argument contained within square brackets is an optional argument and can be omitted entirely if its default setting is acceptable.

[] – Empty square brackets indicate that the optional argument for that position can be left to its default value. They are not necessary for every optional argument but only if including a later optional argument. For example, to specify the 5th input argument but leave the 4th as its default, empty square brackets are used at the 4th location to ensure that the value for the 5th input argument is in the 5th position.

## Chapter 2: What's in the Bits# package?

The Bits# package is supplied with the following parts:

CRS SKU	Quantity	Description
M2108	1	Bits# Control Unit
M2300	1	Friwo 7.5V, 1.7A AC/DC Adapter 7555M/08
M2301	1	Friwo Euro plug
M2302	1	Friwo Australia/China plug
M2303	1	Friwo USA/Japan plug
M2304	1	Friwo UK plug
M2406	1	2m USB Type A to B, cable, black
M2409	2	2m DVI D to DVI D cable, black
M2415	1	2m DVI A to HD15 cable, black

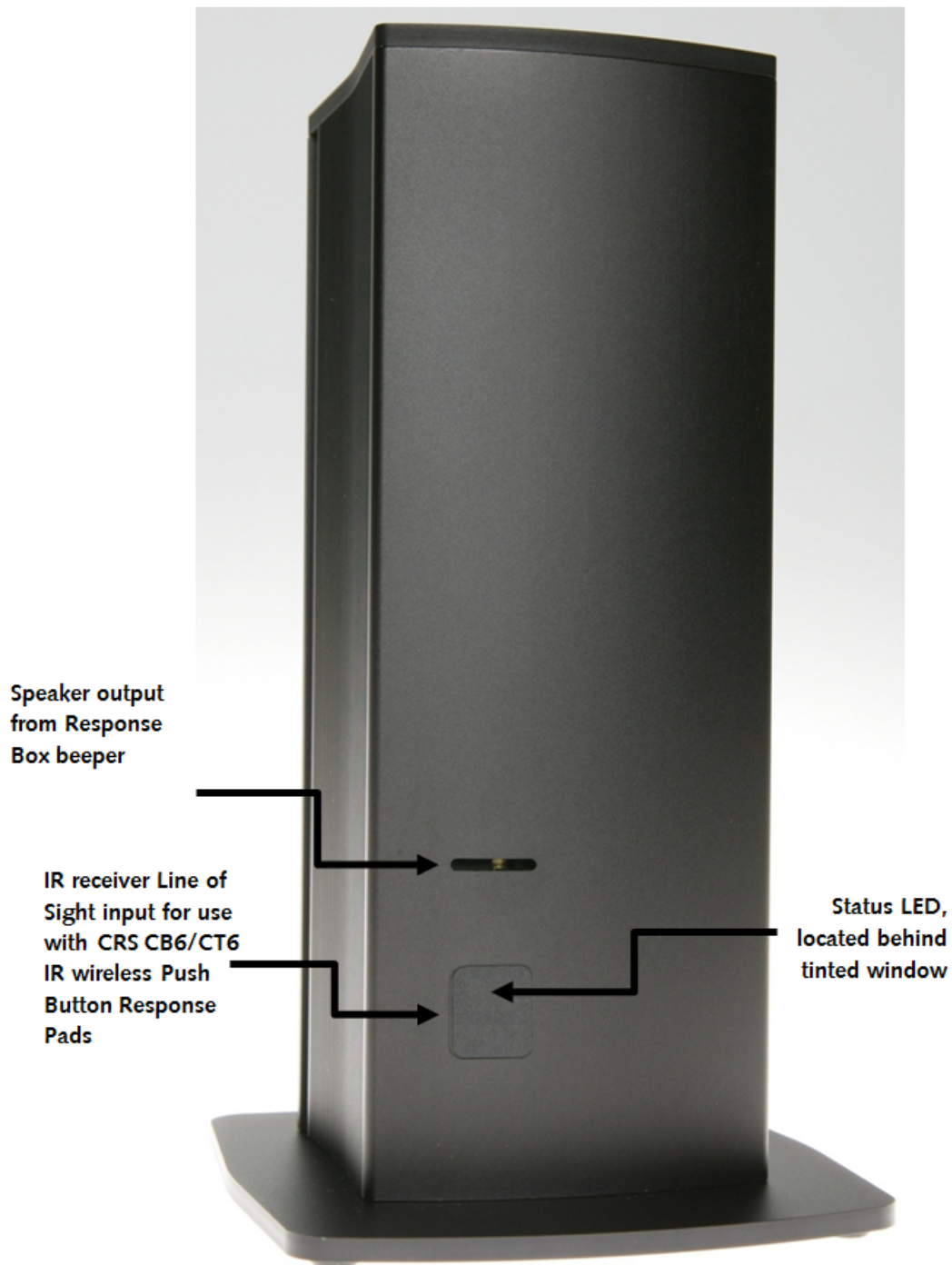


Bits# Package Contents

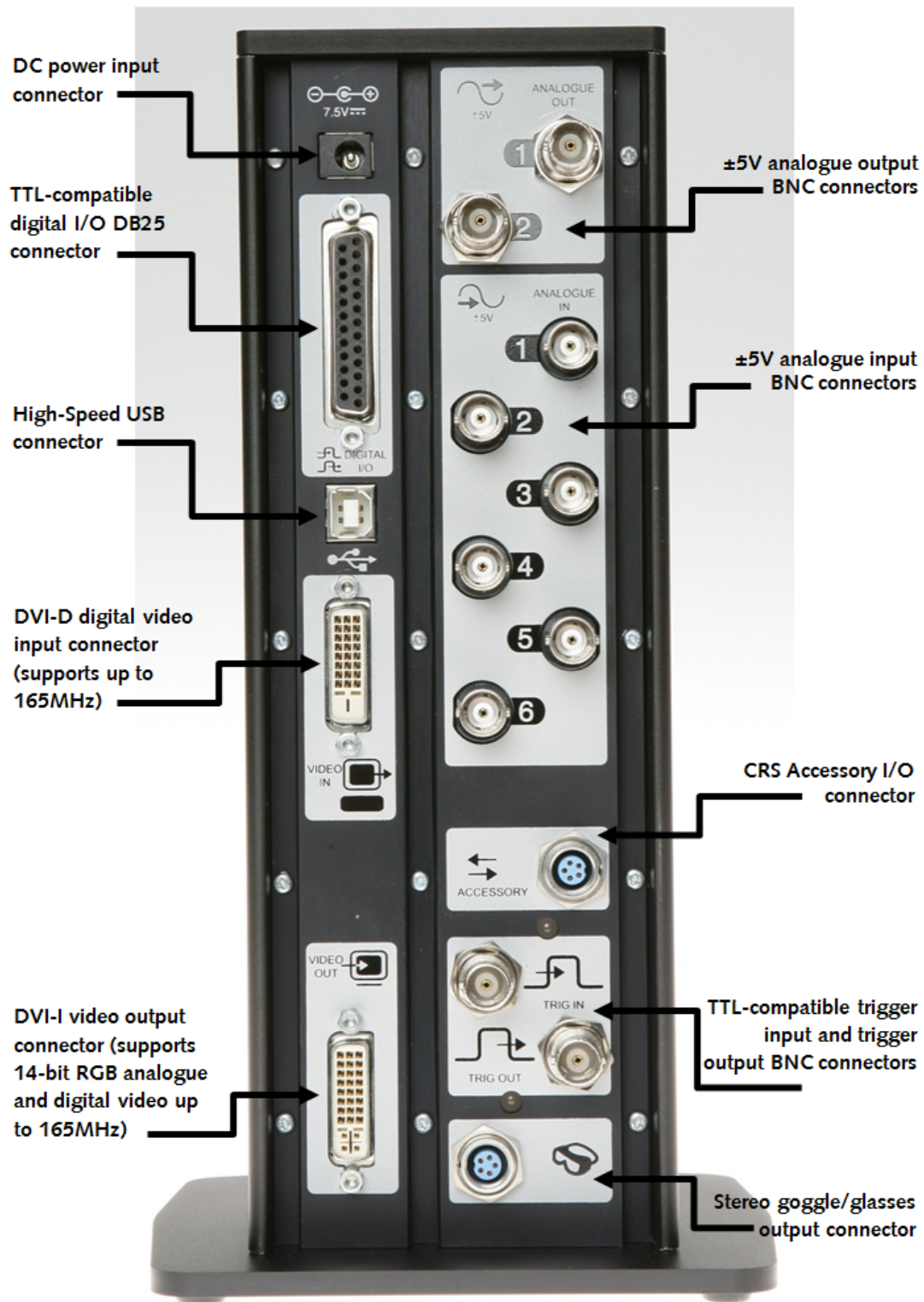




## Chapter 3: Bits# connections



Bits# Front Connections



Bits# Rear Connections

# Chapter 4: Bits# installation

## 4.1 Hardware Requirements

### 4.1.1 Graphics Card

Bits# can be used with any computer that has a graphics card with a digital video output. This includes graphics cards with DVI, HDMI, DisplayPort, Mini DisplayPort and Thunderbolt connectors. If your graphics card has an HDMI, DisplayPort, Mini DisplayPort or Thunderbolt output then you will need to obtain an adapter which converts the output to DVI. These adapters are passive in the sense that no signal conversion is required as DVI is directly compatible.

Please note that if your computer only has an analogue RGB video output, typically on a HD15 “VGA” connector, then you will not be able to use it with Bits#, even if you purchase a so-called VGA to DVI converter. These active electronic converters will not preserve the video well enough for Bits# to operate correctly.

### 4.1.2 USB port

A High-Speed USB 2.0 compatible port is required for device configuration, debugging and two way communication with the integrated response box, digital triggering and analogue I/O subsystems. Super-Speed USB 3.0 ports are backwards compatible with High-Speed USB 2.0.

If you need help identifying the connectors on your computer, please visit the following web pages:

[http://en.wikipedia.org/wiki/Digital\\_Visual\\_Interface](http://en.wikipedia.org/wiki/Digital_Visual_Interface)

<http://en.wikipedia.org/wiki/HDMI>

<http://en.wikipedia.org/wiki/DisplayPort>

[http://en.wikipedia.org/wiki/Thunderbolt\\_\(interface\)](http://en.wikipedia.org/wiki/Thunderbolt_(interface))

[http://en.wikipedia.org/wiki/VGA\\_connector](http://en.wikipedia.org/wiki/VGA_connector)

<http://www.vesa.org/>

<http://en.wikipedia.org/wiki/USB>

<http://www.usb.org/home>

### 4.1.3 Stimulus display

Bits# supports analogue CRT and digital DVI/HDMI input display devices (e.g. LCD monitors, LCD projectors, DLP projectors). Each display technology has its advantages and disadvantages; make your choice carefully based on the visual stimulus that you need to

display and the experimental measurements you want to make. It is important to remember that there is no single technology that will be suitable for all applications.

Bits# can be used with a computer that is configured for use with a single monitor (i.e. one that will alternate between the operating system desktop and the stimulus), or multiple monitors (i.e. one for the operating system desktop and one or more dedicated stimulus displays). You only need to have Bits# connected to the stimulus display monitor(s).

## 4.2 Cable connections

Make the following connections between Bits#, the host computer and the stimulus display. Start the procedure with all devices switched off.

1. Use one of the supplied M2409 DVI D video cables to connect the digital video output from your host computer graphics card to the Bits# VIDEO IN connector (see hardware requirements if you don't have a DVI connector on your host computer).
2. Connect your stimulus display to the Bits# VIDEO OUT connector. This can be done using an M2409 DVI D or M2415 DVI A video cable. If you are using a display device that has multiple input connections (e.g. LCD monitor) and DVI D is one of the available options, choose that one in preference to an HD15 connector.

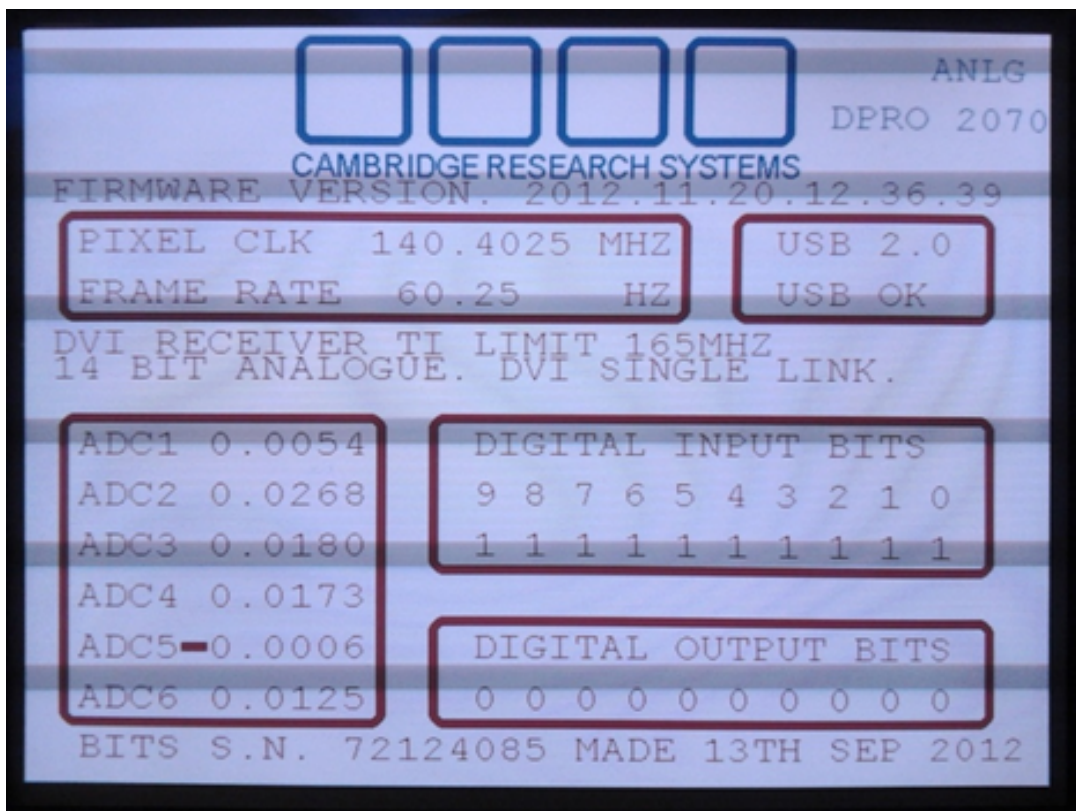
If you have a second monitor for your operating systems desktop, connect this monitor directly to your host computer either into a secondary video port on your graphics card (if applicable) or into a second graphics card (cable not included).

3. Connect the Bits# USB Standard-B port to the USB Standard-A host port on your computer with the provided M2406 USB cable.
4. Assemble the M2300 Friwo power supply with the appropriate M2301, M2302, M2303 or M2304 plug for connection to your AC mains outlet. The Friwo power supply supports a universal 100V-240V AC input and outputs low-voltage 7.5V DC.
5. Connect the assembled Friwo power supply and plug to an AC mains outlet, but do not switch on the mains supply.
6. Attach the Friwo DIN 45323 coaxial power connector to the Bits# power input connector.
7. Switch on the mains supply.
8. The Bits# status LED on the front of the hardware unit will illuminate red/orange while the hardware boots its device firmware. After a few seconds the boot process will have finished and the LED will be illuminated green. It will remain green until

the Bits# VIDEO IN port receives a suitable digital video input signal, or until the Bits# self-generated Diagnostic Display is switched off by sending it a command.

9. Switch on your stimulus display but do not switch on your computer yet. When the monitor has synchronised to the video signal output by Bits#, you should see the Bits# Diagnostic Display. The Diagnostic Display provides information about the current video input signal, the detected stimulus monitor, plus the status of the response box, digital I/O and analogue output interfaces. It will report the Bits# hardware serial number and the current firmware revision number.

Note that Bits# supports video data input with a pixel clock up to 165 MHz (single-link DVI). See the appendix for a table of frame rates and screen resolutions that conform to this standard.



Bits# Diagnostic Display on a CRT monitor

10. Now switch on your computer. When Bits# detects a video input signal, the Diagnostic Display will switch off. This will happen automatically when the graphics card that is connected to Bits# outputs a digital video signal.

11. If you are using a dual-display configuration and Bits# is connected to the secondary output of the graphics card, you might need to wait until the operating

system finishes booting before you can enable this output. Configuration of the graphics card outputs is done in the properties for the display driver (e.g. Windows Control Panel, Hardware and Sound, Display or Mac OS X System Preferences, Displays). Bits# will pass through the resolution and frame rate capabilities of the detected monitor and let you set those in the display driver software, however, make sure your frame rate and resolution stays within the Single-Link DVI limit or the display will not be shown correctly.

## 4.3 Operating System Support

All the Bits# video encoded interfaces, including the digital trigger interface, are designed to work with any operating system and graphics card driver that supports 24bpp (bits per pixel) or better, RGB encoded graphics. Some operating systems, notably Mac OS X call this graphics mode “millions of colors”.

The Bits# hardware configuration, response box, digital input and analogue I/O interfaces are controlled using High-Speed USB and will work with any operating system that has support for Mass Storage Device (USB MSD) and Communication Device Class, Abstract Control Model (USB CDC ACM) drivers. Examples of operating systems that have native USB MSD and USB CDC ACM class drivers include 32-bit and 64-bit editions of: Apple Mac OS 9, Mac OS X and GNU/Linux distributions with kernel 2.4 or later. Microsoft Windows XP SP3 and later comes with USB MSD support but needs to be installed (see Installation section).

Since Bits# uses industry standard computer interfaces like DVI, USB MSD and USB CDC ACM, no proprietary drivers, special configuration software or SDK is required. The Bits# features are all straightforward to access using the operating system APIs and standard developer tools like C and Python, or high level scripting tools like MATLAB. This open approach helps community-developed, free software tools like Psychtoolbox and PsychoPy to integrate with Bits# and take advantage of its special hardware features. The information in this technical manual can be used by software developers to write new custom software using graphics libraries like OpenGL and DirectX, or to extend existing proprietary software to support Bits#. We encourage feedback and suggestions for new features, so please send any comments to us at [crsltd@crsltd.com](mailto:crsltd@crsltd.com). If you want to find out more about the USB class drivers and what they do, please visit:

[http://en.wikipedia.org/wiki/USB\\_mass-storage\\_device\\_class](http://en.wikipedia.org/wiki/USB_mass-storage_device_class)

[http://www.lvr.com/mass\\_storage.htm](http://www.lvr.com/mass_storage.htm)

[http://en.wikipedia.org/wiki/USB\\_communications\\_device\\_class](http://en.wikipedia.org/wiki/USB_communications_device_class)

[http://www.lvr.com/usb\\_virtual\\_com\\_port.htm](http://www.lvr.com/usb_virtual_com_port.htm)

[http://www.usb.org/developers/devclass\\_docs](http://www.usb.org/developers/devclass_docs)



## 4.4 Operating System Configuration

When Bits# is switched on for the first time and the device firmware has booted using the factory supplied configuration, the USB interface will enumerate in Mass Storage Device (MSD) mode and it will be detected automatically by the host computer operating system as a Removable Storage device (e.g. just like a memory stick or a flash memory card). USB MSD mode exposes the Bits# internal 2 GB SD Card and mounts it as a FAT32 formatted volume, just like a USB hard disk drive. In this mode, the Bits# volume can be navigated with the usual operating system file manager (e.g. Windows Explorer, Mac OS X Finder) and the device configuration file config.xml (located in the folder firmware) edited using a standard text file editor (e.g. Windows Notepad, Mac OS X, TextEdit or GNOME gedit) or any other software tool that can open, read and write plain text format files. In addition to the config.xml file, the Bits# volume contains the device firmware files, a collection of monitor configuration EDID files, A Windows driver INF file to support the USB CDC ACM mode, documentation, ancillary software and utilities, and a collection of Bits# demonstration scripts for MATLAB.

*An Important Note: You can copy the contents of the Bits# volume to another disk drive if you want to make a backup, but under no circumstances should any of the files be deleted or the directory structure changed. This is particularly important for the files in the firmware folder.*

In normal operation, Bits# will be operated in USB Communications Device Class Abstract Control Model (CDC ACM) mode rather than MSD mode. The CDC ACM mode enumerates the Bits# USB interface as a virtual serial port (e.g. Windows COM port, Mac OS X /dev/tty.usbmodemxxxxxx port or Linux /dev/ttyACMx port) and will respond to the Bits# command set which is described in the appendix. You can open, read, write, and close the virtual serial port using a variety of software tools, the freeware PuTTY tool for Windows and Linux and the Mac OS X and Linux Terminal program called screen. Developer tools like C and Python and many other applications including MATLAB and contain integrated support for communicating with serial ports too.

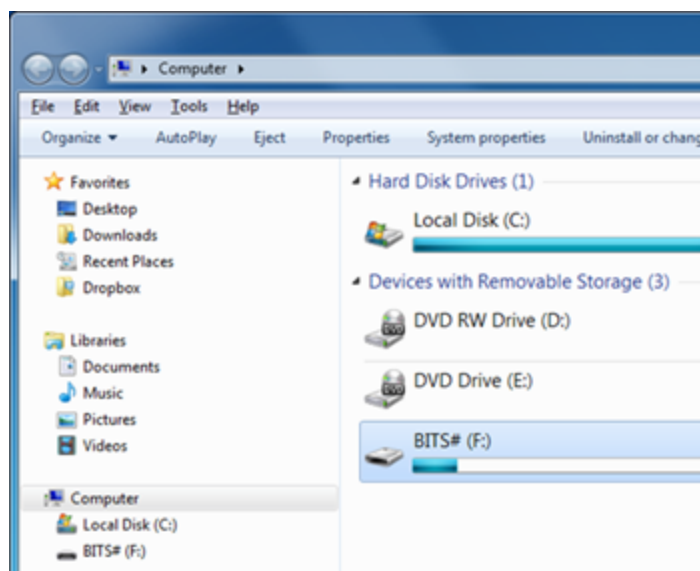
The remaining pages of this section describe how to access the Bits# config.xml file and enable the USB CDC ACM mode separately for Windows, Linux and MAC OSX operating systems.

If you want to use an operating system that is not described in this manual, it does not necessary mean that it is not supported. Please contact us for further guidance in that case. Bits# should detect your monitor automatically and apply the corresponding EDID configuration file. However it is possible to set it manually. Please refer to the appendix for a description of the procedure.

## 4.5 Installing Using Windows Vista or Windows 7

When you connect Bits# to the host computer by USB and switch it on for the first time, it will be in the factory default configuration which enables the USB Mass Storage Device (MSD) mode.

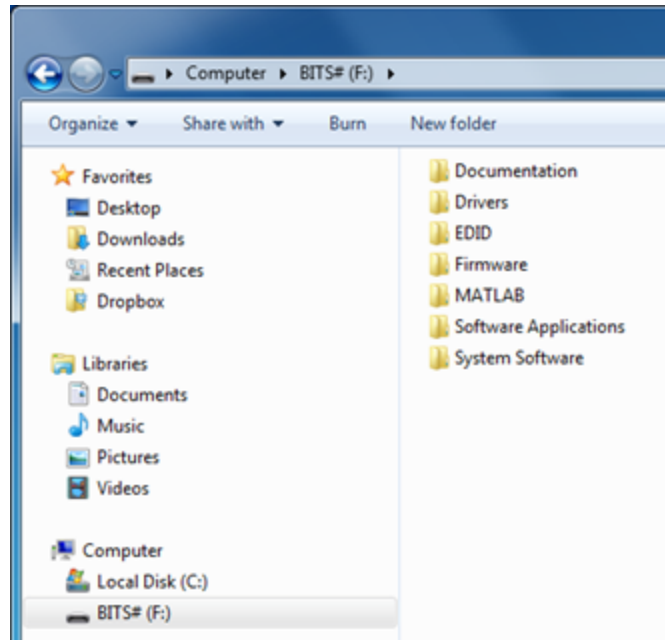
1. Use Windows Explorer to navigate to 'Computer' or 'My Computer' and look for the Removable Storage volume labelled 'Bits', 'Bits#' or 'VISAGE' (here labelled BITS#). A drive letter will have been automatically assigned to the Bits# volume by the operating system. In this example it has been assigned drive letter F.



Bits# in Mass Storage Device mode

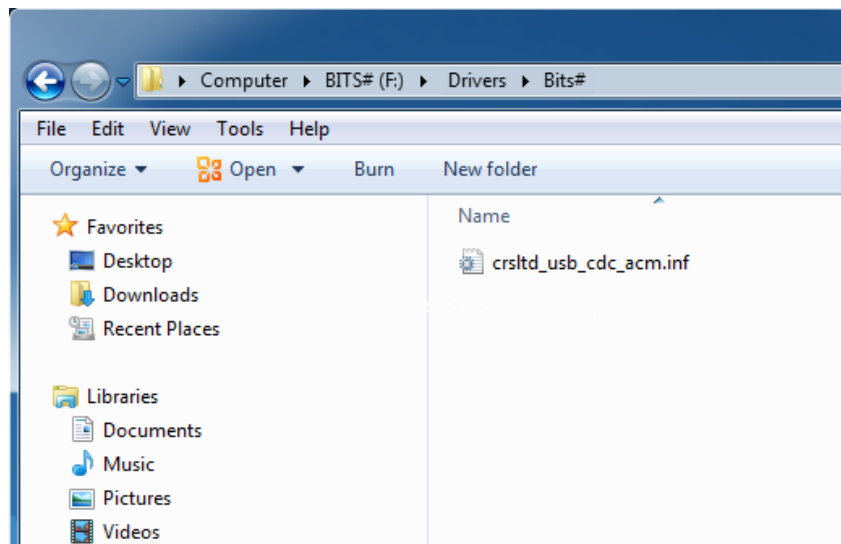
2. Open the 'Drivers' Folder in the main directory.





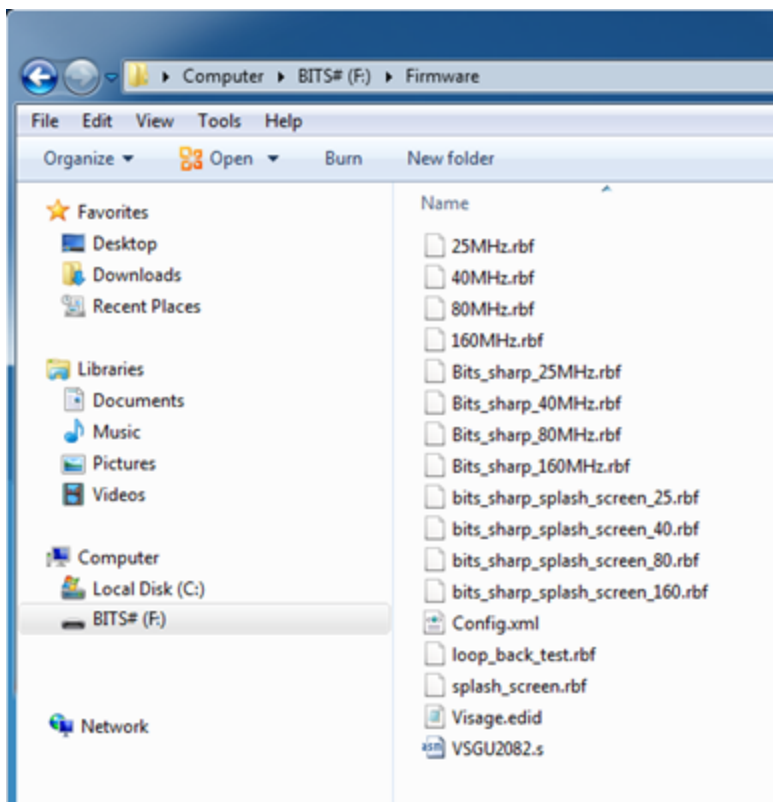
Bits# Drivers Folder

3. Open the 'Bits#' folder. Within this folder will be the `crsltd_usb_cdc_acm.inf` driver file to be used for Windows systems. Be sure to copy this file to your system to somewhere on your computer's own hard drive before continuing with the set-up. Note the location this file is saved to so that it can be located later.



Bits# USB CDC driver for Windows

4. Return to the main folder and open the Firmware folder. Open the Config.xml file in a text editor (e.g. Notepad).



Bits# Config.xml file

5. Near the top (lines 7, 8 & 9) you should find:

```
<Entry USB_MSD="ON" />  
<Entry USB_CDC="OFF" />  
<Entry USB_HID="OFF" />
```

Change the USB-MSD from "ON" to "OFF" and the USB\_CDC from "OFF" to "ON". Save the file and close it. Make sure that when you save the file you do not inadvertently change the file extension to .txt as this will cause Bits# will not recognise the file.

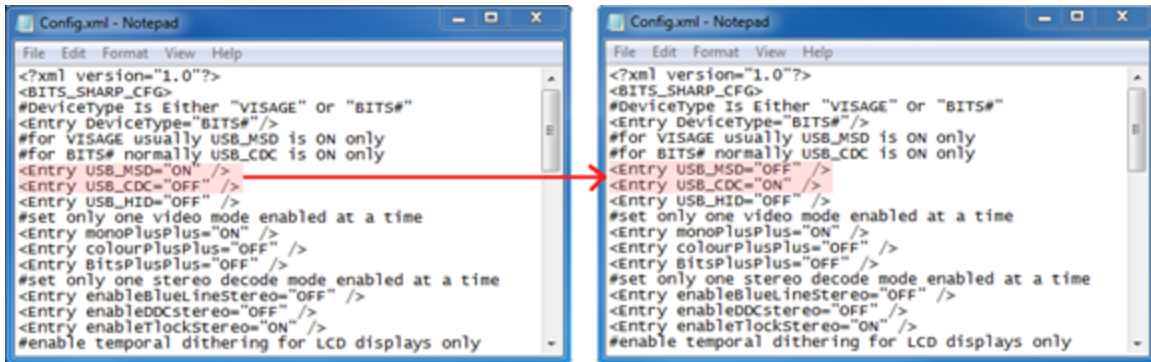


Figure 9 - Bits# Config.xml file

**An Important Note:** You must ensure that the Bits# configuration file name is always preserved as Config.xml. If an invalid Config.xml file is saved, the device will default to a mass storage mode and will not output any display (not even the status screen). Therefore if this happens, be sure to carefully check the Config.xml file is correct.

6. To make the new settings active, just turn the Bits# power off and back on again by removing and reinserting the coaxial power connector. This will make the Bits# USB interface start up in CDC ACM mode. The first time this happens you will be prompted for a driver by the Windows Add New Hardware wizard.
7. A window should appear regarding installing the drivers. These are the files you saved in Step 3. Use the 'browse' option to specify the location and allow it to install. Bits# should now be installed and you can proceed with the section Communicating with Bits# using Windows.

## 4.6 Installing Using Mac OS X

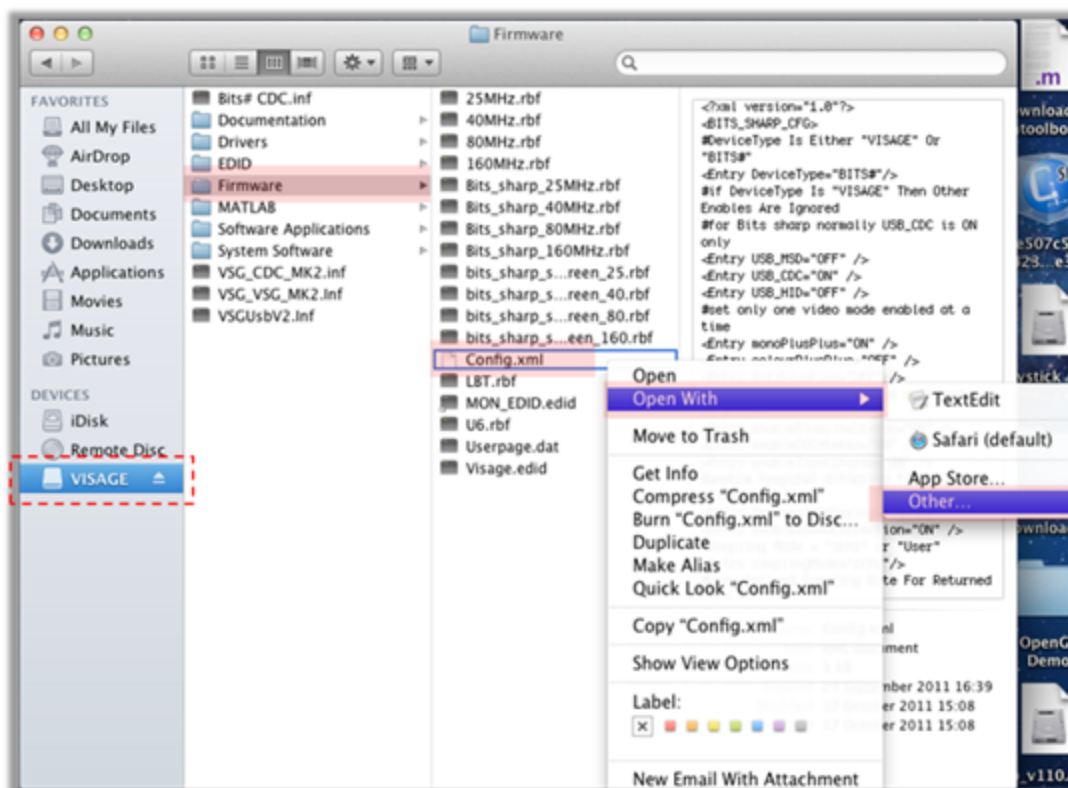
When you connect Bits# to the host computer by USB and switch it on for the first time, it will be in the factory default configuration which enables the USB Mass Storage Device (MSD) mode.

1. If a window does not automatically pop up after you connect Bits#, use 'Finder' in the bottom left of the screen:



Bits# Config.xml file

2. Look for a folder under 'DEVICES' called either BITS, BITS#, or VISAGE and left click on it. Then left click on the Firmware folder. RIGHT click on Config.xml (depending on your settings, to "right click" may be to click with two fingers simultaneously or to click while holding the 'ctrl'-key), left click on 'open with' and choose 'TextEdit' (if 'TextEdit' is not already an option, left click 'other' and choose TextEdit from the list that appears).

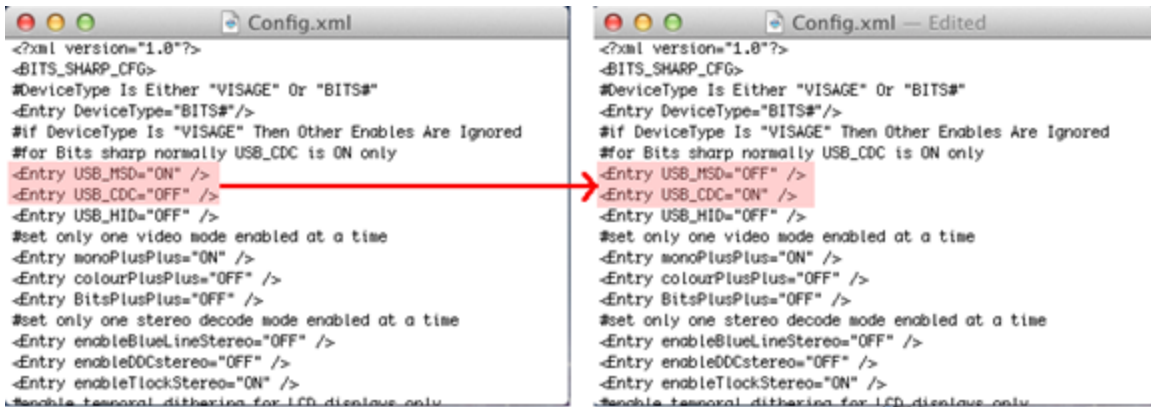


Bits# Config.xml file

3. Near the top (lines 7, 8 & 9) should read (see image below):

```
<Entry USB_MSD="ON" />
<Entry USB_CDC="OFF" />
<Entry USB_HID="OFF" />
```

Change the USB-MSD from “ON” to “OFF” and the USB\_CDC from “OFF” to “ON” (see image below). Save and close the file.



Bits# Config.xml file

**An Important Note:** You must ensure that the Bits# configuration file name is always preserved as Config.xml. If an invalid Config.xml file is saved, the device will default to a mass storage mode and will not output any display (not even the status screen). Therefore if this happens, be sure to carefully check the Config.xml file is correct.

4. To make the new settings active, just turn the Bits# power off and back on again by removing and reinserting the coaxial power connector. This will make the Bits# USB interface start up in CDC ACM mode. Bits# should now be installed and you can proceed with the section Communicating with Bits# using Mac.

## 4.7 Installing Using Linux

The following describes the installation using Ubuntu 12.04 LTS.

When you connect Bits# to the host computer by USB and switch it on for the first time, it will be in the factory default configuration which enables the USB Mass Storage Device (MSD) mode.

1. If a window does not automatically pop up after you connect Bits#, use ‘Home’ in the top left of the screen. Look for a folder under ‘DEVICES’ called either BITS,

BITS#, or VISAGE and left click on it. Then left click on the Firmware folder. RIGHT click on Config.xml, left click on 'open with' and choose 'gedit'.

2. Near the top (lines 7, 8 & 9) should read (see image below):

```
<Entry USB_MSD="ON" />  
<Entry USB_CDC="OFF" />  
<Entry USB_HID="OFF" />
```

Change the USB-MSD from "ON" to "OFF" and the USB\_CDC from "OFF" to "ON" (see image below). Save and close the file.

*An Important Note: You must ensure that the Bits# configuration file name is always preserved as Config.xml. If an invalid Config.xml file is saved, the device will default to a mass storage mode and will not output any display (not even the status screen). Therefore if this happens, be sure to carefully check the Config.xml file is correct.*

3. To make the new settings active, just turn the Bits# power off and back on again by removing and reinserting the coaxial power connector. This will make the Bits# USB interface start up in CDC ACM mode. Communicating with Bits# using Linux.

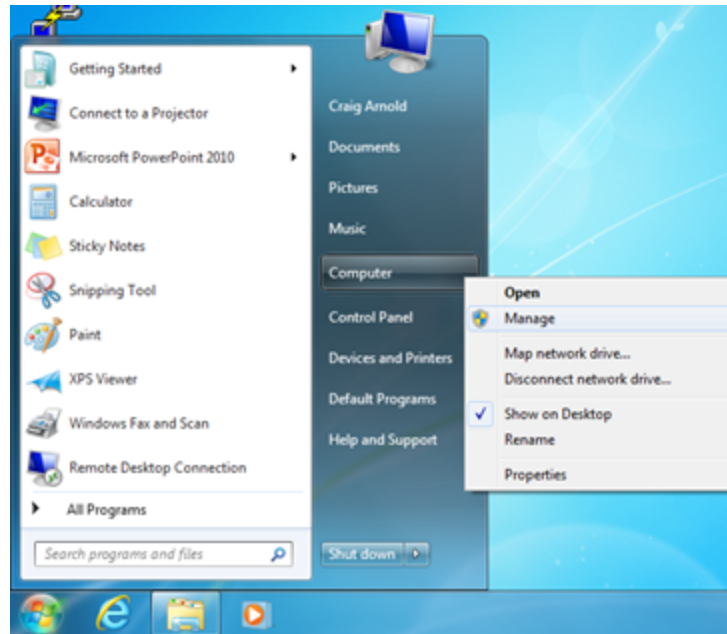
## 4.8 Communicating with Bits# using Windows

Bits# is connected to the computer via a USB port but when set to CDC mode (see installation) communication occurs as though it were a serial port. In this mode Bits# can be communicated with using a specific set of commands (listed in the appendix).

### 4.8.1 Determining the Port

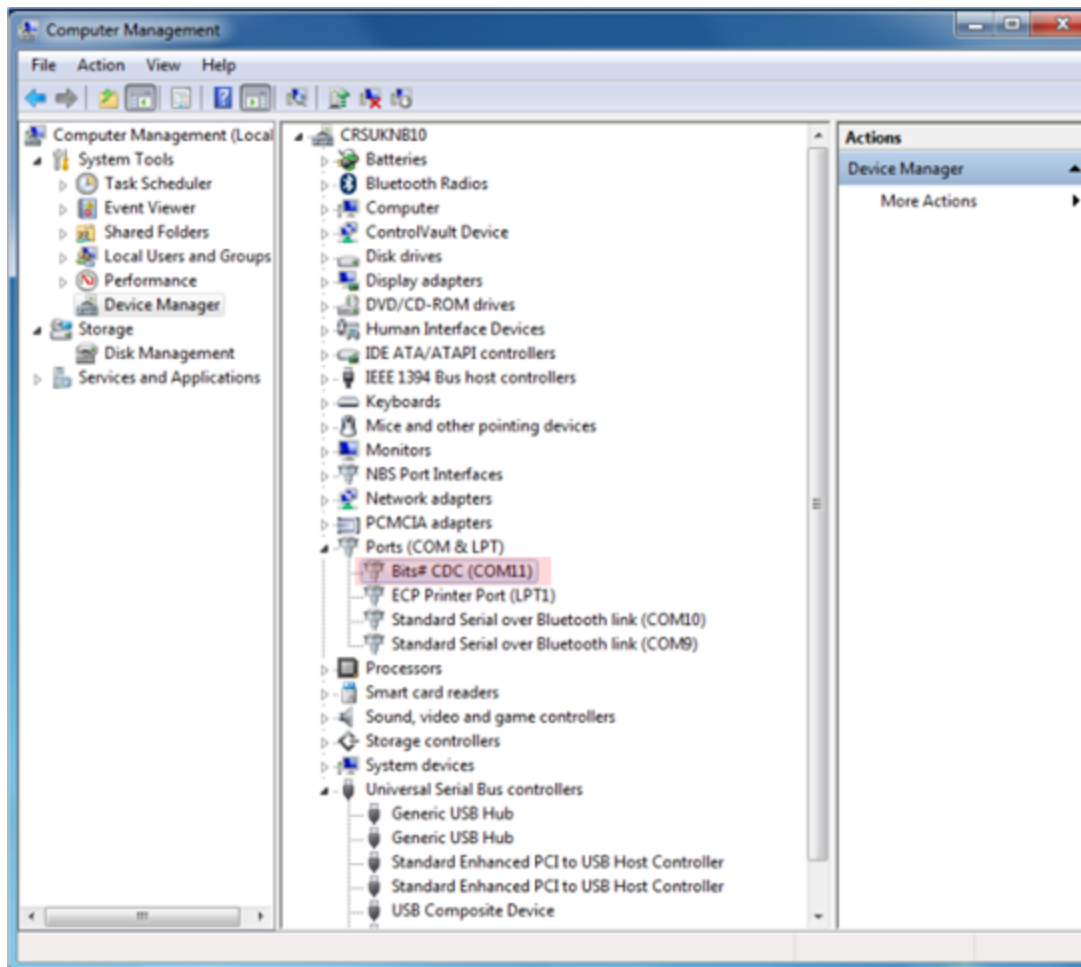
Before setting up the connection to communicate with Bits#, it needs to be determined which port it is connected to. If this is already known, skip ahead to the 'Using the serial console' section. Otherwise, it can be obtained from the device manager:

1. Click Start
2. Right click on 'Computer' or 'My Computer' and select 'manage':



### Open Device Manager

3. In the “Computer Management” window, select ‘Device Manager’ from the list on the far left.
4. Select ‘Ports’ from the middle list that then appears.
5. Look for “Bits# CDC” and take a note of the value inside the brackets (COM11 in the below example):



Determining the serial port

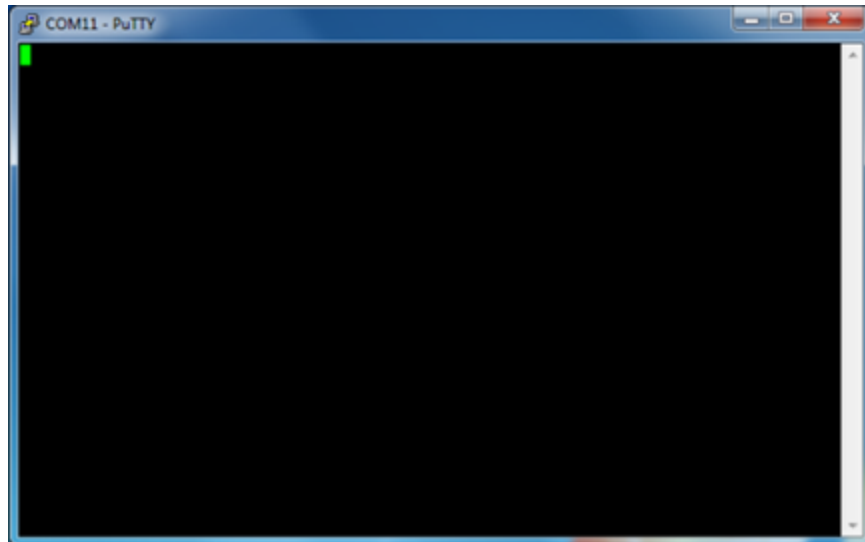
## 4.8.2 Using the serial console

### Setting up a connection

1. In the following we will describe the use of PuTTY which is a free, open source program that can be used to connect to Bit# from both Windows and Unix platforms. It can be downloaded from [www.putty.org](http://www.putty.org).
2. When installed and opened, change the 'connection type' to 'Serial'.
3. After changing the connection type to serial, enter the name of the port Bits# is connected to (COM11 in this example). If you are unsure, check in Device Manager (see above).



4. The speed in bytes per second can be specified, but unless reason to change it, as the serial port is simulated, it can be left as the default value. For future convenience, these setting can be saved by entering a name into the 'Saved Sessions' box and clicking 'save'.
5. Bits# should now be connected via a serial-like port and the main PuTTY window (see image below) should be open and ready to be written to.



PuTTY console window

## Communicating with Bits#

Once the serial port has been created, commands can be typed into PuTTY. Commands are entered as words, always preceded by either a dollar (\$) or a sharp (#) symbol (a # symbol on a mac keyboard is [Alt, 3]). Note that the commands entered will not be shown in the console window. To see a full list of possible commands, see either the appendix of this manual or type:

```
>> $Help
```

*An Important Note: You will not be able to see what you type as you type it, but key presses will be registered. If you enter a command and nothing happens, try typing it again taking care to ensure correct spelling. Commands are case sensitive!*

## Changing Bits# mode

To return the Bits# tower back to USB mass-storage mode, open PuTTY and create a connection with the appropriate port. Enter the following:

```
>> $USB_massStorage
```

An Important Note: When Bits# is set to USB mass-storage mode the serial connection is lost and it is not possible to return to CDC mode via the serial console. To return to CDC mode make sure the config.xml file is set to boot in CDC mode (see Bits# installation section) and unplug/replug the power cord.

Other modes include:

>> \$monoPlusPlus

>> \$colourPlusPlus

>> \$BitsPlusPlus

>> \$statusScreen

## 4.9 Communicating with Bits# using Mac

Bits# is connected to the computer via a USB port but when set to CDC mode (see installation) communication occurs as though it were a serial port. In this mode Bits# can be communicated with using a specific set of commands (listed in the appendix).

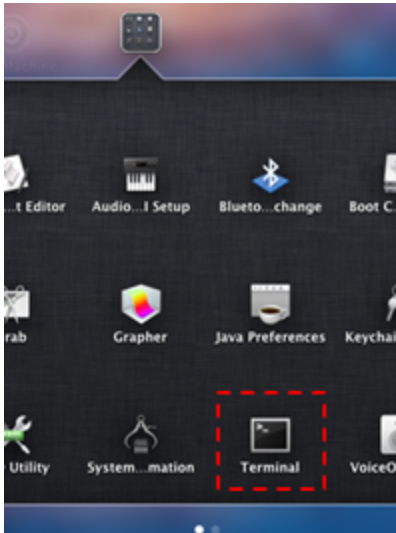
### 4.9.1 Setting up a connection

1. Bits# can be communicated with using the terminal. To use this, open 'Launchpad' in the bottom left of the screen.



Launchpad

2. Select Utilities and then click on 'Terminal' (see images below).



Open the terminal

3. Regardless of which directory Terminal may be in when you open it, to get to the directory needed, type “cd /dev”:

```
Tests-Mac-mini:/ test$ cd /dev
Tests-Mac-mini:dev test$
```

Go to the /dev folder

4. Type “ls” (LS) and from the list that it presents, find the name of the location Bits# is connected to (‘tty.usbmodemfa131’ in the below example).

5. Finally, type ‘screen’ followed by the name of the port Bits# is connected to.

For example:

```
>> screen tty.usbmodemfa131
```



Find the Bits# port

6. This should then blank the previous input commands and any output they displayed, leaving a blank screen. This is the open connection with Bits#.

#### 4.9.2 Communicating with Bits#

Once the serial port has been created, commands can be typed into the terminal. Commands are entered as words, always preceded by either a dollar (\$) or a sharp (#) symbol (a # symbol on a mac keyboard is [Alt 3]). To see a full list of possible commands, either refer to the appendix or type:

>> \$Help

**An Important Note:** You will not be able to see what you type as you type it, but key presses will be registered. If you enter a command and nothing happens, try typing it again taking care to ensure correct spelling. Commands are case sensitive!

Other modes include:

>> \$monoPlusPlus

>> \$colourPlusPlus

>> \$BitsPlusPlus

```
>> $statusScreen
```

### 4.9.3 Closing the connection with Bits#

To exit the 'Screen' function (freeing up the serial port for future use in other scripts), press 'Ctrl' + 'A', followed by 'Ctrl' + '\'. A black box should appear at the bottom of the screen asking if you are sure you want to quit. Press 'y' to quit.

To exit the terminal and return to the desktop, either use a 3-finger swipe across a track pad from left to right. This will leave terminal open in the background and can be returned to by a 3-finger swipe in the opposite direction. Alternatively, to close terminal completely, press 'Cmd' + 'Q' (The Cmd key is the one with the symbol ).

## 4.10 Communicating with Bits# using Linux

Bits# is connected to the computer via a USB port but when set to CDC mode (see installation) communication occurs as though it were a serial port. In this mode Bits# can be communicated with using a specific set of commands (listed in Appendix A).

### 4.10.1 Setting up a connection

1. In the terminal window you can check all ports and their permissions by typing:

```
ls -l /dev/tty*
```

The returned list might be long and difficult to view in the window and since Bits# it likely to be called something starting with "ttyACM" you can try typing:

```
ls -l /dev/ttyACM*
```

This will return a list of all ports of starting with ttyACM. For example:

```
crw-rw---- 1 root dialout 166, 0 Apr 12 17:20 /dev/ttyACM0
```

Now, identify the port Bits# is connected to. If you are unsure, you can try to disconnect the USB cable and look if it disappears.

2. Next you will need to add yourself to the group of the port. Notice that the port is in the group "dialout". Check which groups you belong to by typing:

```
groups
```

This will return the groups the current user belongs to. For example:

```
user adm cdrom sudo dip plugdev lpadmin sambashare
```

To add yourself to the group "dialout" type:

```
sudo usermod -a -G dialout user
```

- where "user" is your user name. You will need the root password to do this. To check your user name you can use the command: whoami
- 3. Log out and back in for the group change to take effect. After logging back in try typing "groups" again. You should now see "dialout" among your groups:

```
user adm dialout cdrom sudo dip plugdev lpadmin sambashare
```

#### 4.10.2 Communicating with Bits#

1. If your Bits# is connected to the port "ttyACM0" type :

```
screen /dev/ttyACM0
```

If you don't have "screen" installed you can install it by typing:

```
apt-get install screen
```

2. Once in "screen" type (note that you will not be able to see what you type) :

```
$ProductType
```

You should now get the return from Bits#:

```
#ProductType;Bits_Sharp;
```

To see a full list of possible commands, either refer to the appendix or type::

```
$Help
```

To quit press "Ctrl-a" and then "\

You will be asked:

```
Really quit and kill all your windows [y/n]
```

Press "y" to quit

**An Important Note:** You will not be able to see what you type as you type it, but key presses will be registered. If you enter a command and nothing happens, try typing it again taking care to ensure correct spelling. Commands are case sensitive!

#### 4.10.3 Changing Bits# mode

To return the Bits# tower back to USB mass-storage mode, enter the following:

```
>> $USB_massStorage
```

**An Important Note:** When Bits# is set to USB mass-storage mode the serial connection is lost and it is not possible to return to CDC mode via the serial console. To return to CDC mode make sure the Config.xml file is set to boot in CDC mode (see Bits# installation section) and unplug/replug the power cord.

Other modes include (see the appendix or use the #Help command for a full list):

```
>> $monoPlusPlus
>> $colourPlusPlus
>> $BitsPlusPlus
>> $statusScreen
```

## 4.11 Communicating with Bits# using MATLAB

### 4.11.1 Introduction

In MATLAB, information is passed to, and read from, Bits# the same way as though writing to, or reading from, a file using `fprintf` and `fscanf`.

### 4.11.2 Creating a serial port object and handle

First, create a serial port object and handle where port is the name of the USB port Bits# is connected to. If you are unsure of this, see the 'setting up Bits#' chapter above, appropriate for your operating system.

```
>> s1 = serial('port');
```

The syntax of serial ports varies across platforms. Some examples include:

```
>> s1 = serial('COM5'); % Microsoft Windows
>> s1 = serial('/dev/tty.usbmodemfa121'); % Mac OS X
>> s1 = serial('/dev/ttyACM0'); % Linux
```

**An Important Note:** On Linux systems, MATLAB might not recognise ports of the type "ttyACM". In this case you can do one of two things:

A) Make a file called "java.opts" containing a line like this:

```
-Dgnu.io.rxtx.SerialPorts=/dev/ttyS0:/dev/USB0:/dev/ttyACM0
```

In this example the three ports "ttyS0", "USB0" and "ttyACM0" are added for MATLAB to recognise. It is assumed that "ttyACM0" is the port of Bits#. If not, change it

correspondingly. Place the file in the directory you start MATLAB from. Alternatively, place it in your home folder if you have a desktop launcher for MATLAB. The latter might be easier since it does not require write permissions.

B) Create a symbolic link from a port that MATLAB recognises e.g. /dev/ttyS101 to /dev/ttyACM0. See the operating system documentation for further information on symbolic links.

### 4.11.3 Opening the serial port for communication

Once it is created, to open the port for communication, use:

```
>> fopen(s1);
```

#### 4.11.4 Writing to Bits# via the serial port

While open, Bits# can be written to using `fprintf`. For example, to set Bits# to mono++ mode:

```
>> fprintf(s1, ['$monoPlusPlus' 13]);
```

**An Important Note:** All Bits# commands start with either a dollar (\$) or 'sharp' (#) symbol (the # key on a mac keyboard is 'Alt + 3'). Also, the '13' is the terminator character (13 is a carriage return) and should be added to the end of all input commands.

#### 4.11.5 Reading from Bits# via the serial port

Some input commands will generate a reply from Bits# which can be read using `fscanf`. One can check whether there is information available to be read using:

```
>> s1.BytesAvailable
```

```
ans = 0
```

Positive values indicate available information. For example, '#Help' should return a list of possible commands, including the different modes Bits# can be set to (see below):

```
>> fprintf(s1, ['$Help' 13]);
```

```
>> s1.BytesAvailable
```

```
ans = 117
```

```
>> helpOutput = fscanf(s1);
```

```
>> helpOutput
```

This will only return one line (one command) at a time. An easier way to list multiple line responses would be to use the serial console (PuTTY or 'screen') as the buffer is automatically emptied to the screen (a list of all the commands is also to be found in the appendix).

You can also empty the buffer in MATLAB with the `fread` command:

```
>> fread(s1,s1.BytesAvailable)
```

Note that when using the `fread`, you receive the ASCII codes of the character string. Use the command "char" to convert.

**An Important Note:** Both the input and output buffers have a limited size (default = 512 bits). Attempting to write or read more bits than there is space will result in an error. The current size of the buffers can be checked using:

```
>> s1.InputBufferSize
```

```
ans = 512
```

```
>> s1.OutputBufferSize
```

```
ans = 512
```

If larger buffers are needed, these sizes can be changed. For example:



```
>> s1.InputBufferSize = 1000;
>> s1.OutputBufferSize = 1500;
```

#### 4.11.6 Changing Bits# modes

As with the serial console, you can use MATLAB to return Bits# to USB mass-storage mode:

```
>> fprintf(s1, ['$USB_massStorage', 13]);
```

**An Important Note:** When Bits# is set to USB mass-storage mode the serial connection is lost and it is not possible to return to CDC mode via the serial connection. To return to CDC mode make sure the Config.xml file is set to boot in CDC mode (see Bits# installation section) and unplug/replug the power cord.

The other modes of Bits# can be set in a similar way:

```
>> fprintf(s1, ['$monoPlusPlus', 13]);
>> fprintf(s1, ['$colourPlusPlus', 13]);
>> fprintf(s1, ['$BitsPlusPlus', 13]);
```

#### 4.11.7 Closing the Serial Port

Once the serial port has finished being written to, it should be closed as soon as possible to minimise the chances of errors occurring if the script aborts part way with the port still open:

```
>> fclose(s1);
```

**An Important Note:** If a script does abort without closing the port properly, the port can get stuck open. It may have no handle assigned to it with which to close it, and further attempts to reopen it will likely fail as it is already open. You can either try the command: "delete (instrfindall)" or if that fails closing and then re-opening MATLAB again should clear all ports, allowing for it to then be used again.

#### 4.11.8 Disconnect the Serial Port

Once a port is finished with (such as at the end of a script), it should be disconnected and closed properly to avoid potential problems when trying to open it again in future:

```
>> delete(s1);
>> clear s1;
```



# Chapter 5: Introduction

## 5.1 Overview

This section contains background information on a range of terms and concepts referred to regularly throughout this manual. If you are confident with the concepts discussed in the subheadings here (e.g. gamma correction and temporal dithering), much of this chapter can be skipped. However, it is still highly recommended that you read the Bits# subsection as this gives an introduction to how the Bits# system works, an overview of its main ‘modes’ of operation and their uses.

Further chapters will then provide more practical explanations and demonstrations on how to use Bits#. Please note that many MATLAB functions, such as those in Psychtoolbox, will likely have more advanced options and input arguments that are not outlined fully in this manual. Please refer to the developer’s own documentation for full details.

## 5.2 Bits

A ‘bit’, referred to often throughout the manual, can be thought of as a binary number (can be either 0 or 1). The more bits one has, the more combinations of values are possible. For example, 1 bit can be set to only two possible values (0 or 1) while 2 bits can be set to 4 different combinations (00, 01, 10 or 11). Each of these combinations will relate to a different ‘condition’.

Usually, the user does not need to be concerned with the actual combination of 0s and 1s that relate to the desired condition and can instead simply specify each different condition as a single number. For example, if using 8-bits, to indicate the first condition, instead of specifying ‘00000000’, the user need only specify ‘0’. Similarly, to call the 256th condition, instead of specifying ‘11111111’, the user need only specify ‘255’. Note that values start at 0, not 1.

The number of combinations possible for a given number of bits is equal to  $2^n$  (2 to the power of n) where n is the number of available bits. Important number of available bits, referred to throughout the manual, include:

Bits	Number of combinations of values possible
8	256
13	8192
14	16,384
16	65,536
24	16777216
42	~4398000000000

### 5.3 Pixel Values

Each pixel on your display monitor typically has three sub-pixels or channels associated with it: One for each of the three (additive) primary colours, red, green and blue (RGB). These ‘colour channel’ values specify what intensity the respective ‘colour gun’ should fire at. For example, to make a pixel display the maximum intensity of red, the R value should be set to maximum while the G and B values should be set to 0. Some set-ups will also have a 4th value associated with each pixel – its alpha value. This relates to the transparency of the pixel, and determines the weightings when averaging between the existing pixel values and the new pixel values. This value is not supported in all modes however.

A typical graphics card will be able to specify each of the three colour channels with 8-bit accuracy (256 levels), giving an overall 24-bit (3 × 8-bits) resolution of values when combined across the three channels (16,777,216 different combinations).

An example of how the three colour channels combine to display colour is shown in the figure below. To display a very small (3 × 2 pixels) image of the flag of Belgium on an 8-bit display, the relevant values of the colour channels would need to be set to the values shown in the figure. Whereas the red pixels (rightmost two pixels) simply receive full signal from the red channels, the yellow pixels (middle) contains a mix of signals from both the red and green channel to generate the yellow colour. The leftmost two black pixels simply does not receive any signal.

It is worth noting that, with the red pixels for example, although 2 of the 3 colour channels are set to black (0), this does not darken the value of the red channel, they simply do not add anything further to this red light. Pixel values do not average, they are additive.

However, to display a grey pixel, all three colour channels should be set to the same value: if all three values are set to 0, the pixel will display black, if all three values are set to maximum, the pixel will display white and if all three values are the same but in between 0 and maximum, it will display some level of grey. However, this means that a typical display will only be able to specify levels of grey at 8-bit resolution (256 levels, including black and white).

(0,0,0)	(255,255,0)	(255,0,0)
(0,0,0)	(255,255,0)	(255,0,0)

Example of six 8-bit pixel values combined to show black, yellow and red colours.

## 5.4 Bits# Video Modes

In vision science, many uses may require intensities of colour, or shades of grey, to be specified to a greater accuracy than 8-bit (for example, to display a stimulus with improved colour representation or smoother transitions between values). Bits# has three different video modes (called Mono++, Colour++ and Bits++), which include a number of ways of specifying both colour or greyscale values at 14-bits precision per colour channel. This allows for 14-bit resolution (16777216 levels) of greys and an overall combination of 42-bit resolution (~4398000000000) of colours across all 3 channels. The three modes are shortly outlined in the following and a more detailed description of each of the video modes are provided in later chapters.

### 5.4.1 Mono++ mode

Mono++ mode allows the user to specify levels of grey with 14-bits resolution by combining the values of two 8-bit colour channels (red and green) of each pixel and converting them into a grey scale value (all three colour channels equal).

This video mode is most useful for displaying greyscale images, however there is also an option of combining the grey scale with pixels of colour using the third (blue) colour channel using a colour look-up table (cLUT; see LUT section below).

#### Examples of use:

- Displaying complex greyscale images (such as faces) with far more intermediate shades of grey, allowing for improved detail, smoother transitions between shades and reducing banding.
- Displaying greyscale stimuli composed of a narrow band of greyscale ranges, such as a very low contrast Gabor grating, with much smoother transition and reduced banding at the peaks

### 5.4.2 Colour++ mode

Colour++ mode allows the user to specify colour levels to a much higher resolution (42-bits) than normal displays would allow (24-bits). This allows both for a much deeper colour range when displaying images and a much finer manipulation of colour.

Colour++ mode achieves this increased resolution by combining values specified across two horizontally adjacent pixels, and applying that colour to both (see the Colour++ Chapter below for a more detailed explanation). By default, this would therefore mean that when formatting an image to fit to Colour++ mode, it would be stretched horizontally so that it is twice as wide as

the original. However, there are multiple ways of adjusting for this, which will be discussed in the Colour++ chapter.

### **Examples of use:**

- Displaying complex images (such as photographs of scenes) with a much deeper colour range, allowing for more accurate colour representations.
- Subtly manipulating the colour of a stimulus to a much finer level than a normal display would allow.
- Creating a very low contrast colour Gabor grating with much smoother transition than a normal display would allow and reduced banding at the peaks.

### **5.4.3 Bits++ mode**

Bits++ mode uses palette-based drawing. This means that rather than changing the pixel values drawn to the screen, these remain constant and only the values indexed in a saved cLUT (see LUT section below) are manipulated.

This has a number of advantages to drawing in other video modes. If drawing a large or complex stimulus, which requires a lot of pixel values to be changed between frames, then it can sometimes take a long time for the redraw to occur and may result in dropped frames. Using palette-based drawing allows for a much faster updating of the image as updating the 256 entries of a cLUT is almost always faster than redrawing the thousands (or hundreds of thousands!) of pixels that may make up a single frame of stimulus.

However this therefore also limits the number of different colours that can be displayed simultaneously on the screen to 256 (although each of these 256 colours can be specified to a 14-bit resolution).

### **Examples of use:**

- Having drawn an image consisting of a lot of shapes (e.g. circles, lines or geometric shapes) and wanting the same spatial stimuli on successive frames but to change colours. These shapes could be drawn with different pixel values, the cLUT populated with 256 arbitrary colours and cycled through on successive frames.
- Drawing a rectangle with a linear ramp and then populating the cLUT with sin-wave values so that a sinusoidal grating appears on the screen. If on successive frames, these values are cycled through the cLUT, the updated colours displayed to the screen give the impression the grating is drifting or moving, with the same effect as if the grating was being drawn on each frame with updated values for its new position.

## 5.5 Look-up table (LUT)

A look-up table is an  $N \times 3$  table of RGB values where  $N$  is the number of entries (usually 256, although higher precision LUTs may have more). Each of these entries (rows) can then be indexed by a single number. For example, '0' would index the first row and display the corresponding RGB indicated in this row. LUTs are utilised in a number of instances by Bits#.

***An Important Note:** If a LUT specifies colour, it can also sometimes be referred to as a colour look-up table (cLUT).*

A crude and basic example is shown in the figure below (showing a  $3 \times 2$  pixel rectangle, and only 3 rows of a 256 row LUT). In the first frame, the grey levels display what one might expect, given the pixel values. However notice how in subsequent frames, although the grey levels change, the pixel values remain constant. Instead, the RGB values saved in the LUT change so that they are shifted down, with the bottom row being moved up to the top to form a cycle. The pixel values index rows of the LUT so that, for example, a pixel value of 0 will always display the RGB values saved in the first row of the LUT (LUT index 0), regardless of what values these are.

Frame

Pixel Values

Saved cLUT

1


0	128	255
0	128	255

LUT Index	Red Value	Green Value	Blue Value
0	0	0	0
128	0.5	0.5	0.5
255	1	1	1

2

0	128	255
0	128	255


LUT Index	Red Value	Green Value	Blue Value
0	1	1	1
128	0	0	0
255	0.5	0.5	0.5



3

0	128	255
0	128	255

LUT Index	Red Value	Green Value	Blue Value
0	0.5	0.5	0.5
128	1	1	1
255	0	0	0



Multiple rows of a LUT can have the same RGB values so that the same colour can be represented by multiple different pixel values. Also, the LUT does not need to be in any order or pattern (it does not need to be in sequential steps). Each row can be arbitrary and independent of other rows in the table.

## 5.6 Formatting 8-bit into 14-bits representation

In the Mono++ and Colour++ video modes, 14-bits pixel values are generated by combining two 8-bit values (typical graphics card resolution). In Mono++ the two 8-bit values comes from two sub-pixels of the same pixel while in Colour++ mode the sub-pixels from two adjacent pixels are combined (see section above). Regardless of video mode, each has a 'most significant' value in one location and a 'least significant' value in another location. This use of



‘significant’ should not be interpreted to mean ‘important’, but simply indicates which bits the value relates to. Across two 8-bit locations, the MS and LS bits are the left and right 8 bits respectively. Therefore, to represent the 46485th condition as its respective MS and LS values for Bits#:

46484  
1011010110010100  
MS 10110101 10010100 LS  
MS 10110101 10010100 LS  
MS 181 148 LS

It should be noted however that two 8-bit values have 65,536 possible combinations (16-bit), but Bits# supports up to 14-bit resolution. This means that the actual colour or grey being specified will change only with intervals of 4 (i.e. 0, 4, 8, 12...) so that each 14-bit value can be represented by four different 16-bit values.

For example, in Mono++ mode, the grey level 65,280 (of the 16-bit range of 0-65,535) would be the same shade of grey as levels 65,281, 65,282 and 65,283. These four greys would be represented by [MS, LS] values of [255, 0], [255, 1], [255, 2] and [255, 3] respectively. However grey level 65,284 ([255, 4]) would display a grey one level lighter on the 14-bit range.

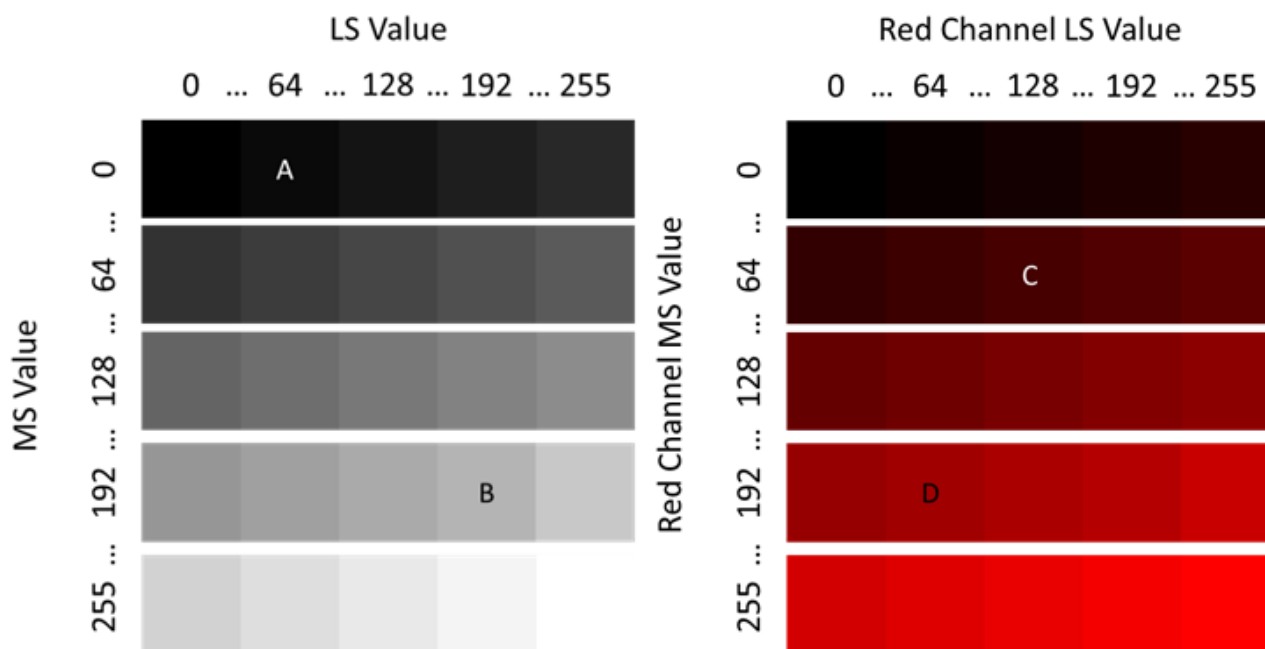
		LS Value							
		0	1	2	3	4	5	6	7
MS Value	255	65,280	65,281	65,282	65,283	65,284	65,285	65,286	65,287

As will be described in later sections, it is recommended that code continues to work in this 16-bit range (0 - 65,535) when specifying values, as it is simpler to format.

In order to calculate the MS and LS components of a 16-bit value, it is not necessary to go through the steps shown in the example above, of converting the ‘decimal’ number (a number - does not need to actually have any decimal places) to its binary equivalent, splitting the MS and LS components and converting them back to decimal numbers. Instead, it is possible to calculate the MS and LS using only decimal numbers.

The MS component is the number of times the 16-bit value can be divided by 256 without any remainder (i.e. rounded down to the nearest integer). The LS component is the remainder of this division. For example,  $46484 / 256 = 181.5781$ , therefore the MS component is 181. Similarly,  $46484 - (181 * 256) = 148$ , which is the LS component.

It may be helpful to think of the MS and LS components analogous to coordinates, with the MS component identifying a narrow range (256) of values and the LS component specifying the specific value within this range. The left and right figures below give crude examples illustrating this analogy, relating to Mono++ mode and Colour++ mode respectively.



Grey A (value 64 of the 16-bit range) would have [MS, LS] values of [0 64], and B (49344) values of [192 192]. If the green and blue colour channels were both set to 0, red C (16512) would have [MS, LS] values [64, 128] and red D (49216) would have values [192, 128]

## 5.7 T-Lock code

**An Important Note:** Many graphics cards transform the output data in a number of different ways. This can be statically such as gamma correction and/or variable by dithering the output in often undisclosed ways. Not only does this cause your stimulus to be unintentionally distorted but it also causes the T-Lock unlock code not to be recognised by Bits#. So even if you are not using the T-lock code it might be a good idea to check that it can be recognised to ensure you are presenting the stimulus you think you are. An unrecognised T-lock can be identified by a line of seemingly random pixel values as opposed to a blanked out line.

A T-Lock code is used by Bits# for some functions, such as initialising a digital trigger or loading a look up table (LUT) to the graphics card. A T-Lock is encoded as RGB pixel values in which the first 8 pixels (the unlock code) are set to a specific, arbitrary combination of values. The RGB values for a varying number of subsequent pixels on that row can then encode various information, depending on the type of T-Lock (see the main sections below for information on what values in different pixel locations mean).

When Bits# receives these values to 'display', it recognizes the 8-pixel unlock code and interprets subsequent pixel values as parameters rather than actual values to display. In addition, to prevent a distracting line of pixels of noisy, arbitrary colours from being displayed, Bits# blanks out that line, such as to black or the background colour, as specified by the user.

Examples of use:

- To initialise asynchronous digital trigger with visual stimuli
- To control a visual stimulus by updating a cLUT.
- To detect distortions of the graphics card output.

## 5.8 Digital Pulse Triggers

If a design requires very accurate timings, such as sub-millisecond accuracy, electronic pulse 'digital triggers' can be employed. These pulses allow for much more accurate timings, less affected by the current state of the computer.

These are triggered by a 'T-Lock' code (see the T-Lock section above). When the Bits# receives this code it blanks out that pixel line and uses the information contained in the subsequent pixels (data packet) to send out an electronic digital trigger pulse.

There are two trigger sockets on the Bits# tower. One (the 'Trigger Out' socket) has 1 pin while the other ('I/O Trigger' socket) has 10 output pins. Each of these pins can individually be set to 'ON' or 'OFF', allowing for a total of 2048 different combinations. These pulses could therefore be used to indicate the timing of an event, such as a stimulus onset or participant response. Different combinations of ON/OFF pins can even be used to give some information about the event, if required, such as which stimulus was displayed.

There are also further options for setting the parameters for the pulse (see the main Trigger section below for more details). For example, although it is typically sent at the start of a vertical screen refresh, it can be set to trigger at a different pixel row, such as the row the stimulus is to be drawn to, further improving the timestamp accuracy.

Examples of use:

- Accurate detection of visual stimulus timings.
- Synchronisation with other equipment.

## 5.9 Gamma Correction

The relationship between the value given to a monitor and the output it actually displays is not linear by default. For example, as the input voltage get higher, further increases in voltage produce larger changes in output intensity than the same voltage increase at lower voltages. Unlike CRT monitors, LCD monitors do not naturally do this, however most LCD monitors artificially create this relationship for compatibility reasons.

As it is simpler for most values to be calculated on the assumption of a linear relationship, a correction needs to be applied before values are inputted in order to achieve the intended output values.

There are several methods for correcting for this effect to give a functionally linear relationship. A crude method is to raise input values by a factor of 2.2, which is a typical 'rule of thumb' value for most monitors. However see the main Gamma Correction section below for further details on more sophisticated and reliable methods of creating a true linear relationship.

## 5.10 Temporal Dithering

In order to increase the range and depth of colours a screen can display, some graphics cards may use a technique known as temporal dithering. This means that if a display is requested to output an intensity it cannot naturally achieve, it can instead display the two closest values it is capable of in quick, alternating succession. To the observer, this will appear as an intensity being displayed that is somewhere between the two, and thus closer to the specified value.

### 5.10.1 Unintended Temporal Dithering

Although this can be useful for general computer use, it can cause significant problems for researchers wanting a high precision of control over low level, psychophysical factors. In addition, temporal dithering is likely to cause the T-Lock code to fail in Bits#. This could prevent digital pulses from triggering and prevent the intended cLUT from being loaded when using Bits++ video mode. It will also have negative effects the pixel encoding in Mono++ and Colour++ video modes.

There are a few ways to overcome this problem (see the Gamma Correction chapter for further details) and it is highly recommended that this is considered and addressed before using Bits#.

### 5.10.2 Intended Temporal Dithering

Most LCD monitors are capable of only 8-bit resolution when specifying colour levels. However Bits# can be use temporal dithering in a controlled, fully disclosed manner to increase this resolution up to 14-bits, depending on the duration of the presentation. It is important to note that because the Bits# itself is controlling the dithering of pixel levels, after

any possible T-Locks have been processed correctly, triggers and LUTs will not be affected by this use of dithering. See the LCD monitor chapter for information on the use of this technique.

## 5.11 Response Box

When collecting participant responses, many use a normal computer keyboard. However, the method the hardware employs for calculating the timings of such responses is not particularly accurate. If a key is pressed, that information is stored in the keyboard. The computer will check this keyboard buffer at regular intervals and extract this buffered information. However timing accuracy is generally of between 4-35ms resolution (depending on the operating system and type of keyboard). If millisecond precision timing is not important then this may not cause much concern, but for some uses may require greater precision.

A common alternative is a response box. There are various types but they generally have between 2 and 6 buttons, in various spatial layouts. Most importantly, these usually have event timings stored (timestamps) so that when the computer checks the buffer, accurate timings are returned.

For example, when using a response box with Bits#, timestamps are returned with 100 microsecond accuracy (1 second = 1,000 milliseconds = 1,000,000 microseconds).

**An Important Note:** *If using the wireless CRS CB6 Response Box, note that Bits#'s infra-red detector is on the 'front' of the tower (the smooth black side, opposite to the side with all of the sockets).*



# Chapter 6: Using Bits# with an LCD screen

## 6.1 Overview

There are a number of differences between a traditional CRT and LCD screens that make that latter less preferable for some vision research. Note that an LCD monitor can be constructed with a CCFL or LED backlight, but the fundamental display technology is the same. The CRT versus LCD screen debate is not fully covered here. Perhaps the limitation of an LCD screen that is most relevant to the use of Bits# is that they are only typically only capable of 8-bit output.

CRT monitors receive an analogue input and so the output values it can display are limited only by the input signal. This allows it to display the 14-bit values Bits# sends it. However the limitation of 8-bits in a typical LCD monitor means that if Bits# produces a 14-bit value to send, the 6 LS bits are ignored and only the 8 MS bits are sent.

Therefore, the Bits# processing of pixel values in mono++ mode and colour++ modes are still only able to produce 8-bit output values on an LCD screen. However, Bits# is able to employ temporal dithering to simulate 14-bit resolution per colour channel.

The temporal dithering feature is designed to enable LCD monitors with a native 8-bit greyscale (24 bit colour) response to have comparable greyscale or colour performance to an analogue CRT.

*An Important Note: Note that this type of temporal dithering, produced by Bits# itself, will not affect T-Lock codes.*

## 6.2 Bits# Temporal Dithering

### 6.2.1 Setup needed

When selecting an LCD to use with this mode it is important to select a monitor that has a panel with 8 bit glass. This might not be the case for entry level consumer panels based on TN technology and some high speed (120Hz panel drive) panels sold for 3D video games which are actually only 6 bits and internally use a temporal dithering (often termed FRC, frame rate control) to simulate a 24 bit response.

When using this mode it is also important to connect the monitor via DVI, don't use an analogue HD15 input.

### 6.2.2 How it works

The Bits# has 14 bit resolution for each pixel output to the monitor. For an analogue CRT this value is converted to a voltage with a 14 bit DAC (digital to analogue converter). However, for

an LCD monitor connected via DVI, only the most significant 8 bits of each of the three colour guns are sent to the monitor. The greyscale performance is therefore much reduced compared to a CRT because 6 of the output bits are never used.

To enhance the greyscale performance an averaging technique is employed. If the 8-bit value of a particular pixel has value “N”, then on a given video frame this may get sent to the monitor as either N or N+1. The aim is that after 64 frames ( $2^6$ ) have been sent, the average is accurate to 14 bit resolution not 8 bits; hence giving 16384 levels rather than just 256.

You cannot “see” the dithering. This is because the grey to grey response time of an LCD panel is typically far longer than the few ms black to white response times that are headlined (advertised) for a monitor (it is important to remember that the panel drive rate in Hz is not the same as the pixel response time, which might be much longer than one video frame). The LCD response is a low pass filter which effectively smoothes out the changes.

### 6.2.3 Limitations of the method

This is an averaging technique. Therefore a 1 frame stimulus presentation will not be more than 8 bit accurate. The resolution does however improve with the number of frames sent (see table below).

Number of frames pixel value displayed for	Time pixel value displayed for on a 60Hz monitor	Bit-resolution possible	Number of values possible
1	16.7ms	8-bit	256
$\geq 2$	$\geq 33.3$ ms	9-bit	512
$\geq 4$	$\geq 66.7$ ms	10-bit	1024
$\geq 8$	$\geq 133.3$ ms	11-bit	2048
$\geq 16$	$\geq 266.7$ ms	12-bit	4096
$\geq 32$	$\geq 533.3$ ms	13-bit	8192
$\geq 64$	$\geq 1066.7$ ms	14-bit	16384

Secondly the output is either N or N+1. You cannot send level 255 + 1 on the DVI link. This means that the codes from 16320 to 16383 are not distinguishable on the output (it will saturate).

Finally, the process is independent for each pixel. There is deliberately no coherence between different pixels of the same pixel level, or between different video frames.

These limitations mean that even with Bits#, it will not be possible to achieve a full 14-bit resolution for some stimuli.

For example, if wanting a stimulus contrast to gradually ramp up over the course of 1 second at stimulus onset, this would not be possible with a 14-bit value as the contrast would need to be kept at a single level during that second.



However, Bits# can still achieve a higher than 8-bit resolution if the level does not need to change on every frame. For example 15 intermediate levels could be chosen and gradually ramped up over the course of the second, each for 66.7ms and therefore able to achieve 10-bit resolution.

Similarly for a slowly drifting stimulus, the stimulus will remain in one position for multiple frames before moving and would therefore be able to achieve higher than 8-bit resolution.

A further note is that although the stimulus needs to be displayed for the number of frames stated to achieve the higher than 8-bit resolutions, these are not 'all or nothing' jumps and although displaying a stimulus for 7 frames will not quite reach 11-bit resolution, it will achieve higher than 10-bit resolution.

Care should also be taken when displaying stimuli for different periods as the same value will appear different to the participant depending on how many frames it is displayed for (and therefore what resolution it achieves). This applies to any stimuli that are specified with > 8 bit resolution and which are presented in one specific state for different numbers of frames, including stimuli moving at different speed or stimuli with different onset/offset ramps. If this is likely to cause a problem, then ensure that such stimuli are specified only in 8-bit values, as these will not be affected by temporal dithering and will appear consistent regardless of the number of frames each state is displayed for.

### 6.2.4 Calibrating a monitor when temporal dithering is enabled

Most colorimeters have a quite long integration time, so will "see" the full synthetic 14 bit resolution. If integration time is selectable with your colorimeter, set the integration time to be 1 second or more.

LCD monitors naturally have an S shaped response. The 2.2 gamma curve that they present is actually "faked" by choosing points along their S shaped curve. It is therefore small segments of the underlying LCD response that will be seen between the 256 points when the temporal dithering is used. To a first approximation the response can be considered as a 2.2 gamma curve with linear line segments between the 256 points. This must be considered when calculating the inverse gamma to load into the hardware gamma correction table available in Bits# mode.

### 6.2.5 Turning on Bits# Temporal Dithering

Temporal dithering can be turned on in one of two ways:

#### **CDC/serial port**

Temporal dithering can be turned on via the CDC interface, such as using Terminal/PuTTY or opening a serial-like port in a script (e.g. MATLAB or Python).

See the above chapter on how to connect to and communicate with Bits# via the CDC interface. The command to turn temporal dithering on is:

```
>> $TemporalDithering = [ON]
```

Similarly, the command to switch off temporal dithering is:

```
>> $TemporalDithering = [OFF]
```

*An Important Note: When turning changing the state of temporal dithering using the serial port, the state will be reset to the state defined in the config.xml file.*

## Config.xml File

Temporal Dithering can also be activated within the Config.xml file itself. To access the Config.xml file, put Bits# into USB\_massStorage device mode (see above chapter for how to do this). In the firmware folder will be the Config.xml file. Open this with a text editor. Part way down will be the following:

```
<Entry TemporalDithering="OFF" />
```

To switch temporal dithering on or off, change this to:

```
<Entry TemporalDithering="ON" />
```

Similarly, if the current status is already set to ON, change this back to OFF to disable temporal dithering.

# Chapter 7: Screen Calibration and Gamma Correction

## 7.1 Overview

### 7.1.1 CRT Screens

In CRT monitors, the relationship between the voltage input and the luminance displayed is not linear. Typically, as the input voltage gets higher, further increases in voltage lead to greater and greater output changes. This relationship is also sometimes called the gamma response because it is commonly modelled with a single variable parameter, the gamma parameter.

One suggested model for the gamma response of a CRT is:

$$\frac{L - k}{L_{max} - k} = \left( \frac{j - j_0}{j_{max} - j_0} \right)^\gamma, \text{ for } j \geq 0, \text{ else } 0$$

Adapted from Georgeson (2007), *Greyscale CRT gamma correction - a brief practical guide for psychophysics* (<http://www1.aston.ac.uk/lhs/staff/az-index/georgema/>)

*L* = output luminance

*j* = LUT output values or DAC input values (i.e. the values contained within the LUT itself, not the index of the LUT)

*j<sub>max</sub>* = maximum input value (typically 1, on the scale of 0 to 1)

*k* = theoretical minimum luminance according to model, however may not be physically possible if *j<sub>0</sub>* < 0

*j<sub>0</sub>* = DAC input required to produce minimum luminance (*k*). If *j<sub>0</sub>* < 0, *k* will not be possible as a negative DAC value cannot be given.

*L<sub>max</sub>* = maximum luminance possible by display (luminance at *j<sub>max</sub>*)

*γ* = Gamma

### 7.1.2 LCD Screens

The gamma response does not apply in the same way to modern LCD monitors, however they often arbitrarily create a similar nonlinear relationship to maintain compatibility. But due to the way such screens are driven, the above equation may not be as appropriate and instead a

simple linear interpolation may be preferred (i.e. a straight line between each of the 256-input values rather than fitting a smooth curve).

### 7.1.3 Non-linear versus linear relationship

Most users would find it simpler to specify stimuli on the assumption of a linear relationship (that the value they put in will linearly related to the output luminance on the monitor), and so with either type of monitor, a transformation should be applied to correct for the effects of gamma. There are multiple methods of doing this.

When using Bits#, there are two gamma correction tables that need to be updated, one in the computer hardware's own graphics card and the other in Bits# itself. It is recommended to make the table on the graphics card linear (no gamma correction) and use the table in Bits# to do the gamma correction. This is has two reasons. First, the gamma correction table in Bits# is of a higher precision than that of the computer graphics card. Second, a nonlinear gamma correction table on the graphics card makes it difficult to send the correct values for the encoding of the stimulus using Bits# video modes and for the Tlock code. It is therefore important first to nullify these effects by loading in a linear LUT (see the LUT section of the Introduction chapter for further explanation on what this means) so that only the gamma correction table of Bits# applies. Unfortunately this can be complicated to do manually, but Psychtoolbox includes several functions that can be used (see the "Nullifying the computer graphics card LUT" section below).

**An Important Note:** The effects of the computer's graphics card LUT can have serious effects on the functionality of the Bits#. In particular, the adjustments made to values may change those that specify a T-Lock unlock code, nullifying its intended effects.

## 7.2 Taking Screen Calibration Measurements

For the most accurate gamma correction, it is best to base corrections on measurements of the outputted values from the screen, given a range of input values. This can be done using a photometer or colorimeter, such as the Cambridge Research Systems ColourCal MKII. Typically a rectangle of a given input value, taking up most (but not necessarily all) of the screen is displayed and a measurement taken. The output level can then be measured and characteristics such as luminance and chromaticity calculated from this.

### 7.2.1 Display Considerations

The values displayed by a screen vary given a number of factors in addition to simply the input value given. Therefore these factors should be considered when taking screen measurements. How thorough a calibration should be will depend on time available and on what the screen is to be used for. It can often be time consuming, especially if taking measurements by hand. However, it is generally recommended that fewer measurements are taken per calibration and the screen calibrated regularly, than infrequent calibrations with many measurements. This

can be sped up further if using an automated device so that a single script can be used to display stimuli, take measurements, calculate gamma correction and apply this automatically.

### **Warming up**

Many screens may take some time (a few hours for example) to fully 'warm up', and their display may vary during this time before settling to a more consistent value. It is therefore recommended that for good control of the display, that the screen is turned on well in advance (at least an hour), before both calibration and being used for display.

### **Settling**

Also, when a screen changes to a new value, it can take some time for this new value to 'settle' to a consistent value. Therefore when making calibration measurements, one should leave the new display on the screen for a while (several seconds) before taking a measurement.

### **Greyscale vs Colour**

The display outputs of each colour gun are not uniform. For example the green colour gun can usually reach a greater intensity than the others. Also, combinations of outputs from each colour gun do not necessarily sum or average in a consistent way, but interact in a complex manner. Therefore when making measurements, it may be useful not only to measure outputs of greyscale values but also for each colour gun individually and perhaps even some combination of guns.

### **Previous Display**

The light displayed from a monitor also depends on the light that was previously displayed. For example, given the same input value, such as that for a mid-grey, the output may vary depending on whether the monitor was previously displaying a black screen than if it had previously been displaying a white screen. Therefore when taking measurements, it is recommended stimuli should perhaps be presented in a random order of presentation.

### **Brightness Settings**

The settings on the monitor may also have some effect on the gamma-correction. If the brightness is set too high, then there may be an offset, with the minimum brightness being above 0. Similarly if set too low the luminance will saturate at 0, with multiple input values producing a 0 luminance output. Ideally the luminance output should be 0 at input value 0, and increase exponentially with increasing input values.

## **Contrast Settings**

If the contrast settings are too high, then the luminance response may saturate at higher input values such that the plotted curve begins to flatten out towards the higher input values. The ideal settings should have a 0 luminance output at input value 0, and increase exponentially with increasing input values without "dropping off" or saturate at high values .

## **Pilot Data**

It may be useful to take some pilot measurements of luminance values for a small number of input values (< 10) to get some idea of the shape of the response curve before performing the full screen calibration. This would allow any issues to be identified and corrected, such as brightness or contrast levels set to high or low.

## **Input values of samples**

The most obvious values to sample are linear, evenly spaced input values ranging from minimum to maximum. However, depending on the monitor and the LUT being created, it may be advantageous to cluster some samples to a particular section of the curve, such as the low values, or the middle values.

## **The rest of the screen**

The output value for a given patch is also partly dependent on what else is being displayed on a screen. For example a mid-grey rectangle may vary depending on whether it is surrounded by a white background compared to if it were surrounded by a black background, even if the background itself is not visible to the measuring device.

## **Time**

A screen's display characteristics will also vary over time. This can occur for a number of reasons, such as natural drifting, bulb age or, especially in the case of CRT monitors, being knocked slightly or even air currents. Calibration should be done for the present characteristics of the monitor and should not be averaged with measurements of past calibrations. However it is recommended that calibration is done as often as possible.

### **7.2.2 Setting up the Calibration**

#### **Stimuli**

The recommended stimulus should be a uniform rectangle drawn at the centre of the screen and taking up approximately  $\frac{3}{4}$  of the screen. The input values of this rectangle can then be varied, taking account of some of the above considerations, and measurements of the output display taken for each input level used.

## Setting up the Device

This will depend on the device being used as there are many types. For example some may need to be attached to the screen using a band while others may be on a stand and need to be stood up against the screen.

It is important to check whether your device has a zero-calibration function that should be called before taking measurements. This will be done by blocking all light from the sensor (such as with one's hand) so that it can adjust all subsequent measurements to account for any offset.

Some devices have their own unique 'correction matrix', assigned by the manufacturer during calibration. This matrix will need to be multiplied by the measurements taken in order to convert them into meaningful values (XYZ values for example if using the ColorCal MKII). This should be checked and applied when taking measurements. Note that it may be stored in the device itself, but still need to be manually retrieved and applied. Check the documentation for your device.

## 7.3 Using measured luminance to create a Gamma Correction LUT

This is a key stage of an accurate screen calibration. Once enough luminance measurements have been taken, these need to be utilized to create a gamma correction LUT (inverse of the monitor's gamma response) that can be used to create a linear relationship between the values defined in the stimulus presentation software and the luminance output of the monitor.

### 7.3.1 Fitting a curve to an equation

The first stage that needs to be done is to fit a curve to the measured data point to minimize the sum of squares (error). There are multiple ways to do this.

The equation for the model of the gamma response of a CRT given in the overview section can be rearranged so that it will calculate the predicted luminance output for any given DAC input:

$$L = k + (L_{max} - k) * \left( \frac{\max(j - j_0, 0)}{j_{max} - j_0} \right)^{\gamma} \text{ if } > 0, \text{ else } 0$$

*Adapted from Georgeson (2007), Greyscale CRT gamma correction - a brief practical guide for psychophysics, (<http://www1.aston.ac.uk/lhs/staff/az-index/georgema/>)*

## Minimizing Error

Most data will not fit a perfect curve due to various sources of slight error. Therefore, the values of  $k$ ,  $j_0$ ,  $L_{max}$  and  $\gamma$  might instead need to be estimated to their 'best fit' so that the error is minimized.

This means that if those values are used in the equation and applied to the input values, the difference between the output values they would predict, and the output values actually measured, is minimal compared to if other parameter values are used.

This involves rearranging the equation slightly so that one side of the equation is empty, as this will equal the error. Therefore in the above equation for example, all of the values on the right hand side of the equation may be subtracted from  $L$ .

### 7.3.2 Interpolation

Once the best fit values of each of the above parameters have been calculated, they can be used to interpolate to other results. The above equation can be inverted to give:

$$j = j_0 + \left( \frac{L - k}{L_{max} - k} \right)^{1/\gamma} * (j_{max} - j_0)$$

*Adapted from Georgeson (2007), Greyscale CRT gamma correction - a brief practical guide for psychophysics, (<http://www1.aston.ac.uk/lhs/staff/az-index/georgema/>)*

This equation can be used with the above parameters to calculate the required DAC input to achieve a target output luminance.

The aim is to create a LUT where linearly increasing input values will produce similarly increasing luminance values. Therefore, an array of linearly spaced luminance values between minimum and maximum should be entered into the above equation. Enough values should be entered to produce enough corresponding input values to fill a LUT of the desired length. For example, if wishing to create a LUT to save to Bits# hardware (see below), a 13-bit LUT (8192 values) is required.

This will return an array of equal length, of the required DAC input values to save as a LUT to produce linear luminances and can then be applied using one of the methods described below.



## 7.4 Applying Gamma Correction

### 7.4.1 Simple Gamma Correction

Simple Gamma correction applies an arbitrary correction value (such as  $^{1/2.2}$ ) to all input values. This can either be done manually to stimuli values prior to being displayed to the screen, or universally using a Psychtoolbox command (if using MATLAB).

However this method is not based on calibration measurements and therefore is not likely to produce a highly accurate result. If good control of stimulus luminance is desired, the more thorough methods, based on calibration measurements, should be used.

### 7.4.2 Saving to the Bits# hardware

Bits# has 2 LUTs. One is the 256-entry LUT, which can be used for functions such as palette-based drawing. This is utilized by the video modes Bits++ and in Mono++. It also has a second 8192-entry LUT which is used for gamma correction. The main advantage of this is that it can be saved to the Bits# hardware and so will remain even after the Bits# is switched off. It can then be enabled permanently by editing the Config.xml file in the internal storage of Bits# (edit the entry: enableGammaCorrection="filename.txt") or temporarily (until device is turned off) by sending a command over the serial interface (See a list of possible CDC serial commands in the Appendix).

### 7.4.3 Using Psychtoolbox

An alternative to using the LUT saved to the Bits# hardware, if using MATLAB, Psychtoolbox includes a function that can allow a 12-bit LUT to be saved and applied on multiple successive calls. This is most useful if not using a Bits# but is made redundant by the Bits# hardware gamma table described above.

## 7.5 Nullifying the computer graphics card LUT (using Psychtoolbox)

To nullify the effects of the computer graphics card LUT, a linear 256\*3 LUT needs to be loaded to the computer's graphics card hardware. This can be done by loading a 256 row \* 3 column matrix using "Screen('LoadNormalizedGammaTable'...)". Each row should vary from 0 to 1 at 256 linearly spaced intervals:

```
>> linearLUT = repmat(linspace(0,1,256)', 1, 3);
>> Screen('LoadNormalizedGammaTable', windowHandle, linearLUT)
```

'windowHandle' is the handle assigned when the window was opened. See the relevant chapter for the Bits# mode being used for full explanation on how to do this.

It is important to note that as it is not possible to access this hardware directly and so the values entered may not be the values that get saved. Also, if this linear LUT required the computer to display a value it does not natively support, it may employ temporal dithering to compensate for this (see the “Temporal Dithering” section of the Introduction chapter for more details). The MATLAB function ‘TlockFixer.m’ can be used to address this. It will run multiple iterations, adjusting the input values until it appears the saved graphics card LUT is linear.

## 7.6 Using the ColorCAL MKII

How measurements are taken will vary between devices, however this section describes how measurements can be taken using the Cambridge Research Systems ColorCAL MKII Colorimeter.

ColorCAL is connected via a USB port which, like the Bits#, can be communicated with as though it were a serial port. See the ‘Setting Up’ chapter for more information and illustrations on how to establish which port ColorCAL is connected to and how to set up a connection using the PuTTY or terminal (depending on the operating system).

### Installing ColorCAL MKII

On Windows systems the ColorCAL serial-like port (CDC) need to be defined by an inf-file. You can use the same file as used when installing Bits#: `crsltd_usb_cdc_acm.inf`. It is stored on the Bits# device in the "Drivers" folder (and is available on our website). Install the ColorCAL using the same procedure as the Bits#.

#### 7.6.1 Communicating with ColorCAL MKII using MATLAB

Like the Bits#, communication is done as though writing to or reading from a text (.txt) file.

##### ***Creating a connection***

To create a connection with the serial port and assign it a handle, use:

```
>> s1 = serial(PortName);
```

s1 then becomes the handle used in all further communication with the port.

##### ***Opening the connection***

Before any information can be sent or received however, the connection then needs to be opened:

```
>> fopen(s1);
```

***Sending commands to ColorCAL MKII***

Commands can then be written to ColorCAL (see below for list of useful commands) using:

```
>> fprintf(s1, [Command 13]);
```

*An Important Note: The '13' represents the terminator character. 13 is a carriage return and should be included at the end of every command.*

***Reading data from ColorCAL MKII***

Some commands will return data. This is indicated by the size of the input buffer (values greater than 0 indicate data that can be read). The current status of the input buffer can be checked using:

```
>> s1.InputBuffer
```

```
ans = 0;
```

To read data from the InputBuffer, use:

```
>> data = fscanf(s1);
```

Data is returned as a string. The exact format of the string and the values will depend on the command given, however usually some further manipulation of this returned string is necessary in order to be able to extract and sort the useful data (see BitSharpGammaCorr.m for an example of how to do this).

***Closing the connection***

When you have finished communicating with the ColorCAL it is important to close the connection properly. Failure to do so may leave it 'stuck' open, preventing future attempts to connect to it. If this occurs, unplugging the ColorCAL will automatically close the connection and it should work as normal when plugged back in again (or type "delete(instrfindall)" in MATLAB to clear all ports).

To close the connection properly use:

```
>> fclose(s1);
```

Using fclose as above will still leave the connection active and can be reopened at any time. When no more communication with ColorCAL will be necessary (such as at the end of the script), to tidy up fully, the following commands should also be called:

```
>> delete(s1);
```

```
>> clear s1;
```

## 7.6.2 Useful Commands for ColorCAL MKII

### Zero Calibration - UZC

The first step that should be performed before using the ColorCAL to take measurements is to perform a zero calibration. All light should be blocked from the sensor. The ColorCAL will then take a reading and assign that to be 0 (no light). If it measured a non-zero value it may imply a zero error in the measurements (all measurements are shifted slightly up or down by the same amount) and will then correct all future measurements by this offset. If it measures a large non-zero value, it suggests that there was too much light and it will return an error message.

The command for this is UZC.

*An Important Note: The calibration begins as soon as this command is sent so ensure that all light is blocked from the ColorCAL sensor before sending it.*

The returned data will either be “OK00” if the calibration was successful or “ER11” if the calibration failed (because light was not properly blocked from the sensor).

This zero calibration is not permanent and will be cleared once the power is turned off. It is therefore necessary to perform this calibration each time the ColorCAL is to be used.

### ColorCAL MKII Correction Matrix – r0N

To increase the accuracy the ColorCal has been calibrated against known values at the site of manufacture. The correction values are stored inside the ColorCal and can be read using the command described below. Note that the measurements taken directly from ColorCal are not calibrated by these correction values. The values need to be read from the ColorCal and applied to each measurement taken. The correction values are stored in a 3-by-3 matrix, or Correction Matrix, and each row must be extracted one at a time using the command r0N where N is the row number (r01 for row 1, r02 for row 2 and r03 for row 3).

The returned data will be in the format:

`ans = 'OK00,XXXXX,YYYYY,ZZZZZ'`

XXXXX represents the value in the first column of that row (a 5 digit number).

YYYYY represents the value in the second column of that row (a 5 digit number)

ZZZZZ represents the value in the third column of that row (a 5 digit number).

This string will need to be formatted so that each value is converted from a string to a number and is then saved to the correction matrix in the appropriate location. See the MATLAB demo, BitSharpGammaCorr.m for an example of how this can be done.

A final consideration of this matrix is that the values returned are not the actual values to use as the correction matrix. The values returned here have been transformed and need to be converted back to their original values.

All values in the matrix have been multiplied by 10000. Numbers which were negative then have had an additional 50000 added to them. A value of 1.0631 would be represented as 10631 and a value of -0.0347 would be represented as 50347.

Therefore, once all 3 rows of the matrix have been retrieved and the returned strings analysed, with each value saved to the correct location, these transformations need to be reversed. First, any values greater than 50000 should have 50000 subtracted from them:

```
>> myCorrectionMatrix(myCorrectionMatrix > 50000) = 50000 - myCorrectionMatrix
(myCorrectionMatrix > 50000);
```

Once this is done, all values in the matrix should be divided by 10000:

```
>> myCorrectionMatrix = myCorrectionMatrix ./ 10000;
```

This matrix can be extracted at any time but all measurements must be multiplied by this matrix before they are used. It may be useful to extract the matrix before making measurements and then multiplying each measurement as it is taken, instead of trying to sort through them at the end.

### Making a measurement - MES

Once the ColorCAL has successfully performed a zero-calibration, measurements can be made. Position the ColorCAL up against the desired screen so that the sensor is roughly in the middle of the screen. Then, display your stimuli one at a time and when ready (taking into consideration the factors outlined above), take a reading.

The command for this is MES.

The data returned is a string in the form:

```
ans = ' OK00,XXX.XX,YYY.YY,ZZZ.ZZ'
```

XXX.XX, YYY.YY and ZZZ.ZZ represent the digits of the X', Y' and Z' values of the reading respectively. These values are not XYZ values as they are raw data and must first be multiplied by the correction matrix for that ColorCAL.

*An Important Note: To multiply the 2 matrices, the raw data from a measurement (X', Y', Z') need to be formatted as a vertical array. It is also important when multiplying that the correction matrix is on the left and the measurement values are on the right:*

Corrected  
ValuesCorrection  
Matrix(A)Measured  
Values

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} A(1,1) & A(1,2) & A(1,3) \\ A(2,1) & A(2,2) & A(2,3) \\ A(3,1) & A(3,2) & A(3,3) \end{bmatrix} \times \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

The corrected values then represent XYZ values, which can be converted to other value types (such as xyY) as required.

## 7.7 Calculating a Gamma Correction LUT using MATLAB

### 7.7.1 For a CRT Monitor

#### Using fminsearch

The MATLAB function `fminsearch` can take an equation with multiple parameters, some of which may be known while some might be those that need to be fitted.

For example, if wanting to fit a curve to the following green channel luminance output values, given the 8-bit input values:

```
myInput = [0 64 128 191 255]
```

```
myOutput = [1.1007 5.4513 16.3324 33.4818 56.3002]
```

To fit the gamma correction model to these points,  $k$ ,  $j_0$ ,  $L_{max}$  and  $\gamma$  would need to be estimated

#### Defining the equation

To use `fminsearch`, first define the equation as a function. This is done using the following format, including the `@` symbol at the start:

```
>> equationHandle = @(inputArguments), sum((equation) .^ 2);
```

As it would be possible to trend towards minus infinity by using negative numbers, the error value needs to be squared so any error will become a positive number and so that `fminsearch` will actually search for the error closest to 0 rather than the 'lowest' number. In addition, because using an array of  $x$  and  $y$  values simultaneously, and that it is the total error that

matters the errors are summed, And to make the largest errors more important to minimize the error values for each position is squared.

Also note that the equation needs to be rearranged so that one side of the equation is 'empty'. It is this side of the equation that will equal the error. So that for the above equation, if  $p$  represents an array of the unknown parameter values  $[k, j_0, L_{\max}$  and  $\gamma]$ , each of the values on the right side of the equation are now subtracted from  $L$  (i.e. have been moved to the left hand side of the equation):

```
>> equationHandle = @(p, j, L, jmax) sum((((L-p(1))./(p(3)-p(1)))-((max(j, p(2))-p(2))./(jmax-p(2))).^p(4)).^2);
```

### ***Defining the known input and output values***

Following this, any known values of any parameters are defined. For example in the current example, the values of  $x$  are known so:

```
>> myInput = [0 64 128 191 255];
>> myOutput = [1.1007 5.4513 16.3324 33.4818 56.3002];
```

### ***Applying fminsearch***

Finally, `fminsearch` is used with the following format:

```
>> bestFitValues = fminsearch(@(unknownParameterValues) ,
    equationHandle(inputArguments), [startingUnknownParameterValues]);
```

This will begin by applying the starting values given in the `arrayOfStartingValues` to each of the respective unknown parameters and calculate the error between the predicted  $y$  values and the observed  $y$  values. It will then perform multiple iterations, varying the starting values slightly, until it calculates the values for the unknown parameters that give the minimal error. It is therefore best to give a roughly sensible estimate of what the parameters might be.

For example, for the above equation, the following might be used:

```
>> bestFitValues = fminsearch(@(p) equationHandle(p, x, j, L, jmax), [0 min(j) max(L) 2.2]);
ans = [0.0866 -0.1299 56.4247 2.1206]
```

These are the best fit values for the parameters  $k, j_0, L_{\max}$  and  $\gamma$  respectively. The equation for calculating the luminance output for a given input value for the green channel of the current monitor could then be defined as:

$$L = 0.0866 + (56.4247 - 0.0866) * \left( \frac{\max(j - (-0.1299), 0)}{255 - (-0.1299)} \right)^{2.1206}$$

## Interpolating

### *Interpolating predicted luminance output from all DAC input values*

Whichever method is used to predict the equation of the line of best fit, this can then be used to predict other output values by interpolation. The above equation is for calculating L given any input value j and so could be used to predict the luminance output given every possible input value.

### *Interpolating required DAC input for linear luminances*

However for the purposes of gamma correction, the target luminance values are known (linear values from minimum to maximum) and it needs to be calculated what values need to be saved to the LUT so that the relationship between LUT index value and luminance output is linear.

*An Important Note: The intended output values are known (linearly spaced between Lmin and Lmax), what is yet to be calculated is what input values are needed to produce them.*

As described earlier in the chapter, the above equation can be rearranged to give an equation that will calculate the required LUT value to produce a given luminance output:

$$j = j0 + \left( \frac{L - k}{L_{max} - k} \right)^{\frac{1}{\gamma}} * (j_{max} - j0)$$

*Adapted from Georgeson (2007). Greyscale CRT gamma correction – a brief practical guide for psychophysics. (<http://www1.aston.ac.uk/lhs/staff/az-index/georgema/>)*

```
>> j = p(2) + ((L - p(1)) ./ (p(3) - p(1))) .^ (1/p(4)) .* (jmax - p(2))
```

In order to achieve a linear relationship, the luminance should increase linearly with each increasing LUT index value. Therefore, the array of L values to give the equation should range from 0 to Lmax with enough values to fill a LUT of the desired size (such as a 13-bit LUT to save to Bits# hardware):



```
>> targLums = linspace(0, Lmax, 2^13);
```

This array can then be entered into the above equation, with the values estimated for  $k$ ,  $j_0$ ,  $L_{\max}$  and  $\gamma$ . For example, continuing the previous example:

$$j = -0.1299 + \left( \frac{\text{targLums} - 0.0866}{56.4247 - 0.0866} \right)^{\frac{1}{2.1206}} * (1 - (-0.1299))$$

```
>> j = -0.1299 + ((targLums - 0.0866) ./ (56.4247 - 0.0866)) .^ (1/2.1206) .* (1 - (-0.1299))
```

The returned array of required input values can then be saved as a LUT (either 3 identical columns for greyscale gamma-correction or three independent columns for RGB gamma-correction).

### 7.7.2 For an LCD monitor

Due to the way LCD monitors are driven, it may be preferable to fit linear interpolations between each point rather than attempting to fit a smooth curve as with CRT monitors above.

Therefore, a polynomial should be fitted using the 'polyfit' function. A suggested value would be to attempt to fit a polynomial of the order of 8, although this will depend on the number of samples. For demonstration purposes only, as the current example only includes 5 values, it will only attempt to fit a polynomial of the order of 2, similar to the curve fitted in the CRT example.

For example, if wanting to fit a polynomial of the order of 2, to the following green channel luminance output values, given the 8-bit input values:

```
>> myInput = [0 64 128 191 255]
>> myOutput = [1.1007 5.4513 16.3324 33.4818 56.3002]
>> coefs = polyfit(myInput, myOutput, 2)

coefs = 0.0008 0.0229 1.0101
```

These returned values, assigned to the "coefs" variable, are the coefficients for the predicted equation for the line. There will always be one more value than the polynomial order (in the current example are three values because a 2nd order polynomial was fitted). These values relate to the equation as:

$$\text{predictedOutput} = 0.0008 * \text{input}^2 + 0.0229 * \text{input} + 1.0101$$

Using this equation, outputs for all of the possible inputs can be predicted by interpolation and saved to an array, each in the position corresponding to the respective input:

```
>> fullInputs = 0:255;
```

```
>> predictedOutputs = 0.0008 .* fullInputs .^ 2 + 0.0229 .* fullInputs + 1.0101;
```

With this interpolated array of output values predicted for each input value, a new LUT needs to be created that will allow each increasing index to produce a linear increase in luminance output. First, the linear outputs to be aimed for should be specified:

```
>> targetOutputs = linspace(myOutput(min), myOutput(max), 256);
```

Finally, for each of these target outputs, the predictedOutputs array can be searched to find the inputs that will produce the closest outputs above and below the target. Either the input value that will produce the closest output value can be used, or, if planning to use greater than 8-bit resolution, such as with Bits# temporal dithering, it can be calculated what proportion linearly between the two adjacent output values the target value is, and the corresponding input value could be saved (see BitSharpGammaCorr.m demo).

Once these values are saved to their appropriate positions within a LUT, this LUT can be applied using one of the methods below.

## 7.8 Saving a 13-bit LUT to Bits# Hardware

### 7.8.1 Overview

Bits# can store 2 LUTs. One is the 8-bit LUT used for palette-based graphics, such as by Bits++ mode or the mono++ mode overlay window. However, the other is a 13-bit LUT that can be used to apply gamma correction to the whole monitor. The LUT is created in a text file and saved to the Bits# hardware. This has the advantage that it only needs to be done once and can then easily be called at the start of a session and will remain active until Bits# is turned off. And if the LUT is created properly, interpolating from measurements using a photometer or colorimeter, then it will also be far more accurate than the crude application of a gamma of 2.2.

The text file (.txt) should only contain 3 columns of 8192 rows. The columns should be separated by a tab, and the rows by a carriage return. The file should not include any titles or headings.

### 7.8.2 Creating the LUT

#### Creating the LUT Matrix

The LUT should be 8192 rows and 3 columns (RGB), with values ranging between 0 (minimum colour gun intensity) and 1 (maximum colour gun intensity). For the best results, the LUT values should be interpolated from measurements using a colorimeter or photometer (see the 'Screen Calibration' section above for further details).

However to demonstrate what to do with this matrix once it is created, the current example will use a simple linear greyscale LUT ranging from black ([0 0 0]) to white ([1 1 1]). When this is applied, it should cause no noticeable difference to the display.

To create this example in MATLAB:

```
>> myLUT = repmat(linspace(0, 1, (2^13)), 1, 3);
```

## Writing the LUT Matrix to a Text (.txt) File using MATLAB

Once the correction LUT ('myLUT' in the current example) is saved as a Matrix, it can be written to a text file.

The first step to write to a text file is to name it, open it and assign it a handle:

```
>> fid = fopen('myFileName.txt', 'w');
```

'fid' becomes the variable assigned as the handle to the file and is used in any further communication with the file. 'myFileName.txt' is the name the file will be assigned. It is important to include the .txt to the end of the name. 'w' indicates that the file is being opened to write to.

*An Important Note: 'w' indicates the file should be opened to for writing to from scratch. If the file already exists, this will clear and overwrite any data that might already be contained within this file. It may therefore be worth including a check within the script that will check whether this filename already exists.*

Once the file is created, it can be written to. The values are written one at a time starting with the top left and moving down the columns before moving across 1. Therefore the above LUT needs to be transposed (using a "'").

In the following example, myLUT is the LUT matrix, '%' indicates the next value should be written, 'f' indicates the value is a decimal number, '8.6' specifies the value should be 8 characters long with 6 digits after the decimal point, '\t' represents a 'tab' and '\n' represents a new line. To write 3 values to a row, separated by tabs and then start a new row, the middle input argument should be '%8.6ft%8.6ft%8.6fn'. Therefore, to write the LUT matrix to a file:

```
>> fprintf(fid, '%8.6ft%8.6ft%8.6fn', myLUT');
```

Once the LUT is written to the file, the file should be closed properly to prevent future problems when trying to access it:

```
>> fclose(fid);
```

### 7.8.3 Saving the LUT to the Bits# (does not require MATLAB)

Once the LUT is written to a file, it can be saved in the Bits# hardware. First, put the Bits# back into #USB\_massStorage mode using the serial port commands (see the 'Setting Up' chapter for instructions on setting up and communicating with Bits# via a serial port).

Open the Bits# drive and find the folder called 'Gamma'. If it is not there, create it. Next, save the LUT text file within this folder. Be sure to take a note the name of the file (e.g. FileName.txt).

### 7.8.4 Applying the LUT

Disconnect any open serial port with Bits#, unplug Bits# and plug it back in so that it will be back in CDC mode. Reopen the Bits# serial port (such as in terminal, PuTTY or MATLAB, see the 'Setting Up' chapter for more information) and pass it the following command:

```
>> enableGammaCorrection = [FileName.txt]
```

The LUT does not update all values simultaneously and one may notice different areas of the screen being updated at different times over the course of a couple of seconds. However then Bits# has applied the gamma correction saved in the text file and will remain active until the Bits# is switched off. As the text file remains on the Bits# hardware, the LUT does not need to be recreated each time but can just be activated by the serial port communication, either manually, or perhaps at the start of a script.

***An Important Note:** Note that multiple such LUT files can be saved to the Bits# hardware in this way and called independently when needed. For example, if a greyscale gamma correction table has been created for use in mono++ mode and an RGB gamma correction table has been created for use in Bits++ mode or colour++ mode, these can both be saved using different file names, and then activated when the respective video mode is to be used*

# Chapter 8: Creating a Colour Look-up Table (cLUT)

## 8.1 Overview

In addition to the gamma correction tables described in the “Screen Calibration and Gamma Correction” chapter, Bits# can store an 8-bit (256 entry) cLUT to be used for palette based drawing (for example in mono++ mode and Bits++ mode). A cLUT is defined and then loaded using a T-Lock code (see the T-Lock code section of the Introduction chapter for further details).

This T-lock can also be used to change the Bits# video mode if it is set to auto++ mode.

*An Important Note: Only Mono++ and Bits++ modes use a CLUT as described in this section. If Bits# is only used in Colour++ mode, this section can be skipped.*

## 8.2 Creating a colour look-up table (cLUT) T-Lock

### 8.2.1 Creating a cLUT T-Lock matrix

Please read the T-Lock section of the Introduction chapter for further information on T-Locks.

#### Assigning Colour Values

Bits# has 14-bit accuracy (16,384 values) per colour channel. However each pixel position can only take an 8-bit value (0-255) per colour channel. Therefore, Bits# uses combinations of values across two pixel locations to specify the colour assigned to a position in the cLUT.

The left pixel represents the ‘most significant’ (MS) value and the right pixel represents the ‘least significant’ (LS) value. Please read the Bits# section in the Introduction chapter for a full explanation of how these values are calculated.

#### T-Lock unlock code

To indicate that the values in the matrix represent information for a cLUT and not simply RGB display data for those pixels, an arbitrary T-Lock unlock code is entered to the red, green and blue channels of the first 8 pixels:

Pixel Number	0	1	2	3	4	5	6	7
Red Channel	36	63	8	211	3	112	56	34
Green Channel	106	136	19	25	115	68	41	159
Blue Channel	133	163	138	46	164	9	49	208

Note that the pixel number count start from zero.

This code is arbitrary and does not contain any specific information about the cLUT itself, but alerts Bits# that the values in the subsequent pixels define a cLUT.

## Blanking Colour

So long as the T-Lock matrix starts on the far left pixel of a row, Bits# will blank out that entire pixel row to a colour specified by the user (for example, the colour of the background) Pixel locations 9 and 10 (pixel no. 8 and 9) specify the MS and LS components of this colour:

Pixel Number	8	9
Red Channel	Blanking colour Red MS bytes	Blanking colour Red LS bytes
Green Channel	Blanking colour Green MS bytes	Blanking colour Green LS bytes
Blue Channel	Blanking colour blue MS bytes	Blanking colour Blue LS bytes

## Video Mode

If the Bits# tower is set to autoPlusPlus mode (either via serial port communication or by changing the config.xml file, the video mode can be switched using the blue channel of Pixel location 12 (pixel no. 11) of the cLUT matrix (note that pixel no. 10 is unused and should be set to 0). If Bits# has already been set to a mode (e.g. monoPlusPlus), any values in these positions be ignored.

Pixel Number	10	11
Red Channel	0	0
Green Channel	0	0
Blue Channel	0	Set Video Mode

Normally "Set Video Mode" would be set to 0 for Bits++ mode, 2 for Colour++ mode or 15 for Mono++ mode. See next section "Creating the cLUT T-Lock Matrix using MATLAB" for further details.

## The cLUT

The remainder of the cLUT T-Lock matrix represents the values to be assigned to the 256 positions of the cLUT. As with the blanking colour values (see above), each colour is specified by the combination of pairs of adjacent pixels to achieve the 14-bit resolution of Bits#. Therefore, this portion takes 512 pixels ( $256 * 2$ ), starting with the values for cLUT position 0 and ending with the values for cLUT position 255.

**An Important Note:** *If using Microsoft Windows, the cLUT must be monotonic. This means that as the cLUT index increases, the colour values must only vary in one direction (i.e. always increasing or always decreasing), or not changing at all. This does not need to be a*

*linear relationship but the direction of change (increasing or decreasing) should not switch at any point.*

**An Important Note:** *If using mono++ mode, cLUT position 0 (pixel no. 12) will not be used and any values entered will be ignored (see the mono++ section for explanation). If wanting to use black ([0 0 0]), this should be set to at least position 1.*

Pixel Number	12	13	...	522	523
Red Channel	position 0 Red MS bytes	position 0 Red LS bytes	...	position 255 red MS bytes	position 255 red LS bytes
Green Channel	position 0 Green MS bytes	position 0 Green LS bytes	...	position 255 green MS bytes	position 255 green LS bytes
Blue Channel	position 0 Blue MS bytes	position 0 Blue LS bytes	...	position 255 blue MS bytes	position 255 blue MS bytes

### The rest of the line

The remainder of the pixel row will also be blanked out, but is unused by the cLUT T-Lock. One possible use for this space would be to encode triggering (see below).

It is recommended that a cLUT T-Lock is presented on every frame (even if not changing) to prevent distraction from the blanked row's onset and offset.

## 8.3 Creating the cLUT T-Lock Matrix using MATLAB

**An Important Note:** *The following MATLAB code can also be found in the demo function TlockGenCLUT.m*

For speed, it is recommended to predefine a matrix of 0s, of the correct dimensions. This is simple as the cLUT matrix is always the same size (1 \* 524 \* 3):

```
>> cLUTMatrix = zeros(1,524,3);
```

### 8.3.1 The Unlock Code

The 1\*8\*3 unlock code is arbitrary and always contains the same values in the same locations, no matter what the rest of the line specifies. Remember that the different channels represent different layers of the z matrix on the same horizontal row, not three different rows as it appears in the above tables. Therefore:

```
>> cLUTUnlockCode = cat(3, [36 63 8 211 3 112 56 34], [106 136 19 25 115 68 41 159], [133 163 138 46 164 9 49 208]);
```

```
>> cLUTMatrix(1, 1:8, 1:3) = cLUTUnlockCode;
```

### 8.3.2 Blanking Colour

The 9th and 10th pixel locations specify the MS and LS values respectively, of the colour to blank the line to. For example, to blank to a mid-grey (all three colour channels the same value, at 32,768 ( $65,535 / 2$ )), the MS and LS values can be calculated using the "floor" and "rem" functions:

```
>> cLUTBlankingMS = floor(32768/256);
```

```
>> cLUTBlankingLS = rem(32768, 256);
```

These can then be assigned to the three colour channels at the 9th and 10th pixel locations:

```
>> cLUTMatrix(1,9,1:3) = cLUTBlankingMS;
```

```
>> cLUTMatrix(1,10,1:3) = cLUTBlankingLS;
```

### 8.3.3 Unused pixel number 10

All three colour channels at pixel number 10 (the 11th pixel location) should be set to 0 as this position is unused:

```
cLUTMatrix(1,11,1:3) = 0;
```

### 8.3.4 Video Mode

The blue channel of pixel number 11 (the 12th pixel location) specifies the video mode. This is only used when the Bits# tower is set to autoPlusPlus mode, otherwise any values entered here will be ignored. If using autoPlusPlus mode, video mode can be switched by using this position.

Along with specifying the video mode, this value also specifies which colour channel Bits# should use as index when drawing pixels using the LUT. The default in Bits++ mode is to use each color channel to independently index its own channel in the LUT (called 'normal' below). In Mono++ mode you should always use the blue channel for the index. In Colour++ mode the index setting is ignored. If using the default index settings the video modes can be set with these values:

0 for Bits++ mode

2 for Colour++ mode

15 for Mono++ mode

If you want to use a single channel to index all colours you can calculate the value by the encoding of 4 binary bits (e.g. '1111'). The two LS bits (those on the far right) represent the video mode itself and different combinations relate to each of the possible modes:

'00' = Bits++ mode

'01' = not supported

'10' = colour++ mode

'11' = mono++ mode



The two MS bits (those on the far left) specify which colour channel to use as the index to the cLUT (normal is default; using each channel to separately index its own LUT channel):

'00' = normal  
 '01' = red channel  
 '10' = green channel  
 '11' = blue channel

**An Important Note:** If using mono++ mode, the index should be always set to the blue channel ('11').

*An Important Note: If Bits# is NOT in autoPlusPlus mode, this pixel will be ignored and the index set to the default ('normal' for Bits++ mode and blue channel for Mono++ mode).*

These can then be combined to give a 4-bit binary combination. This can then be converted to a decimal value using a converter, such as the MATLAB function below and it is this value that is entered into the blue channel of pixel number 11 of the cLUT.

For example, to use mono++ mode ('11') with the blue channel as the cLUT index ('11') would be:

```
>> videoMode = bin2dec('1111')
ans = 15
```

Alternatively, to set Bits# to colour++ mode ('10') and set the cLUT index channel to 'normal' ('00'):

```
>> videoMode = bin2dec('0010')
ans = 2
```

To write this value to the blue channel of pixel location 11 (the 12th pixel location), and to set the red and green channels to 0:

```
>> cLUTMatrix(1,12, 1:3) = [0 0 videoMode]
```

### 8.3.5 The cLUT values

Finally, the values specifying each of the 256 colours of the cLUT need to be specified. As previously outlined, colours are defined by the combinations of values of two adjacent pixels (representing the MS and LS values). The cLUT has 256 positions and so this portion of the Matrix will be 512 pixels long. Not every position needs to be assigned if the matrix was populated in advance with 0s, then all positions will represent black unless otherwise specified here.

To specify a cLUT requires a 2-dimensional 256 \* 3 matrix to be formed representing the red, green and blue channels for the 256 cLUT positions. Remember that if using MicrosoftWindows, these values must be monotonic (see above).

For example, to create a cLUT that ranges from black ([0 0 0]) to a mid-level yellow ([32768 32768 0]) in a linear ramp, the blue channel would need to be set to 0, and the value of the red and green channels would need to range from 0 to 32,768 in 256 steps

```
>> mycLUT(1, 1:256, 1) = linspace(0, 32768, 256);
>> mycLUT(1, 1:256, 2) = linspace(0, 32768, 256);
>> mycLUT(1, 1:256, 3) = 0;
```

To then calculate the relevant MS and LS values for each of the 256 positions of this cLUT, each value needs to be divided by 256 (MS), and any remainder assigned to the LS position:

```
>> mycLUTMS = floor(mycLUT ./ 256);
>> mycLUTLS = rem(mycLUT, 256);
```

Finally, to write these values to the main cLUTMatrix, each of these values should be written to alternate pixel positions starting with the 13th and 14th pixel for the MS and LS values respectively:

```
>> cLUTMatrix(1, 13:2:523, :) = mycLUTMS;
>> cLUTMatrix(1, 14:2:524, :) = mycLUTLS;
```

***An Important Note:** If using mono++ mode, a 0 in the blue channel of a pixel value will always indicate 'transparent' and therefore position 0 in the LUT is never used. If black is desired as the lowest position in the LUT, this should be set to 1, and therefore the cLUT will only contain 255 different colours.*

## 8.4 Using a cLUT T-Lock Matrix with MATLAB

Now that the cLUT T-Lock matrix is complete, it can be written to another image matrix ready to be displayed. It can be written to any line but as it will blank out, it is recommended it is written to a line where this blanking will not disrupt stimulus presentation (such as blanking out the very top pixel row). Also, if the blanked out row can be distinguished from the same row when not blanked (for example, a black blanked out line on a grey background), it is recommended a cLUT T-Lock is drawn to that line for every frame to prevent any distraction due to the onset/offset of the blanking.

This image (with the cLUT T-Lock) can then be drawn and displayed to the screen Bits# is attached to. For example, to attach it to a yellow rectangle,

```
>> myImage = ones(100, 524, 2) * 256;
>> myImage(:, :, 3) = 2;
>> myImage(1, :, :) = cLUTMatrix;
>> myImage = uint8(myImage);
```

```
>> image(myImage);
```

**An Important Note:** Note that if `image(imageMatrix)` returns an error stating values need to be between 0 and 1, it may be that it is representing the matrix values as 'doubles' and need first be converted into 8-bit values using:

```
>> imageMatrix = uint8(imageMatrix);
```

When this image is then displayed on the screen attached to Bits#, the cLUT should take effect. If the T-Lock is positioned such that it starts on the first (far left) pixel (any row), that line should also be blanked out.

However, it might be difficult to make sure the MATLAB figure is not scaled. If the image is scaled the cLUT will not be recognised. An easier way to avoid scaling might be to write the image to a BMP-file on the disk drive and display it with an image viewer. To write the image to a file you can use the following MATLAB statement:

```
>> imwrite(myImage,'myTlockCLUT.bmp','bmp');
```

If the Tlock still fails to blank out the line, it is like to be caused by a problem distortion of the output on the graphics card. This can be either a gamma correction or unintended temporal dithering (see "Temporal Dithering" section of "Introduction" and "Nullifying the computer graphics card LUT" section of the chapter "Screen Calibration and Gamma Correction").

## 8.5 Using the cLUT T-Lock Matrix with Psychtoolbox

### 8.5.1 Simple T-Lock – top pixel row, blanked to black

In Psychtoolbox it is not necessary to create the cLUT T-Lock matrix manually. Psychtoolbox provides the following function that will populate the matrix and assign it to the top pixel row automatically. All that needs to be created is the 256 row, 3 column matrix of the RGB values to be assigned to each cLUT index. However, when using this function, these values should range between 0 and 1. Therefore, to create the same linear cLUT as the above example, ranging between black and a mid-intensity yellow ([0.5 0.5 0]):

```
>> RedCLUTValues = linspace(0, 0.5, 256);
```

```
>> GreenCLUTValues = linspace(0, 0.5, 256);
```

```
>> BlueCLUTValues = zeros(1, 256);
```

```
>> yellowCLUT = [RedCLUTValues; GreenCLUTValues; BlueCLUTValues]';
```

After a window has been open for drawing (see below to the relevant Bits# mode section for a full explanation on opening windows and drawing to them), it can be populated and written to the top pixel line of the window by using the function:

```
>> Screen('LoadNormalizedGammaTable', windowHandle, yellowCLUT, 2);
```

The 4th input argument should be set to '2'. This indicates that the cLUT should be loaded to Bits#, not the computer graphics card.

### 8.5.2 Using a manually created T-Lock code for increased control

However this function lacks the flexibility of creating the matrix by hand. For example, it does not allow the blanking colour, the video mode or the pixel line to draw to to be specified. If using Bits# in any mode other than autoPlusPlus, and the top pixel line being blanked to black is acceptable for your purposes then this function should suffice. If however you need to create the matrix by hand, this can be drawn to the window after one has been opened:

```
>> cLUTtex = Screen('MakeTexture', windowHandle, cLUTMatrix);  
>> Screen('DrawTexture', windowHandle, cLUTtex, [], [0 y-1 523 y]);
```

Where 'windowHandle' is the handle of the window opened (see the relevant Bits# mode section below for a full explanation) and 'y' is the pixel row the T-Lock should be drawn to. On the next Screen('Flip', windowHandle), this will be drawn to the specified pixel row and should activate as a cLUT T-Lock, blanking out that line.

**An Important Note:** Manually drawing the Tlock will only work when Psychtoolbox has not enabled Mono++ output. This means that when you manually draw the Tlock you must also manually format the stimulus data into Mono++ structure and will not be able to let Psychtoolbox' format the data.

# Chapter 9: Mono++ Mode

## 9.1 Overview

### 9.1.1 Normal greyscale

If only luminance is required (not colour), setting all three colour channels to the same value (e.g. [128 128 128]) will produce grey. Lower values produce darker greys (with [0 0 0] being black) and higher values produce lighter greys (with [255 255 255] being white). This is how displays generally produce greyscale mode but is limited to only 256 shades (8 bits). Many uses require higher precision.

### 9.1.2 Mono++ mode greyscale

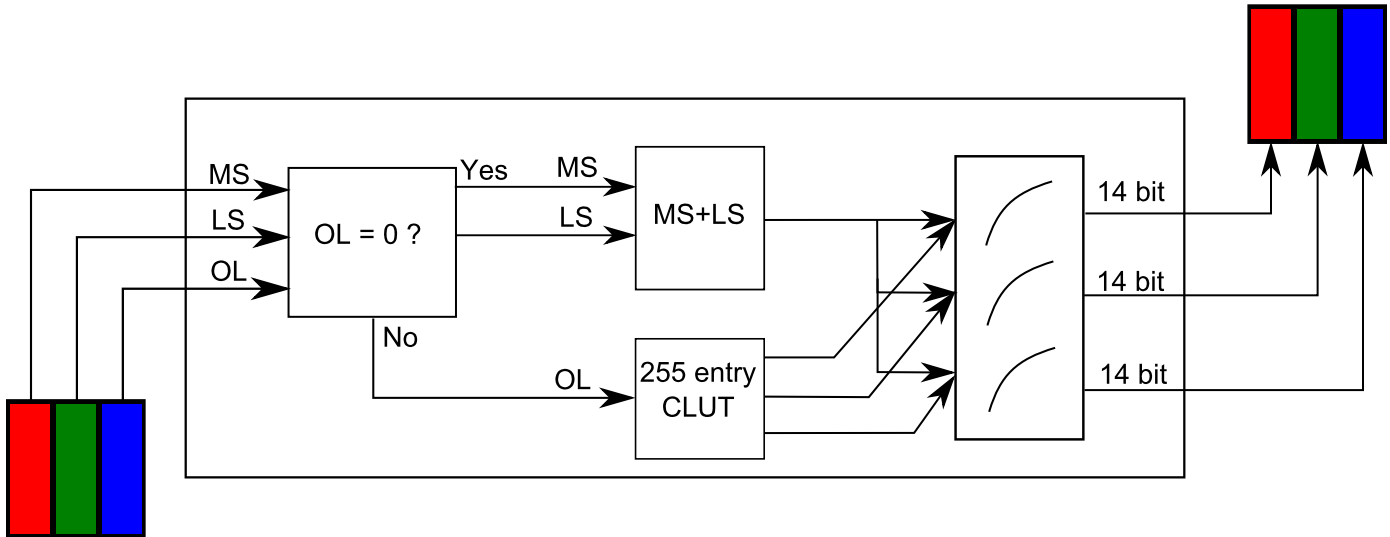
Mono++ mode does not interpret values in the red, green and blue channels as what intensity of red, green or blue respectively to display. Instead it takes combinations of values in the red and green channels to specify greys of different shade, even if these values differ (ordinarily producing a colour). There are 65,536 possible combinations of values (16 bit) of these two channels ( $256 * 256$ ). However Bits# is 14 bit and so different combinations therefore allow Bits# to specify 16,384 levels of grey (8 bits \* 6 bits;  $256 * 64$ ).

### 9.1.3 Mono++ mode overlay channel

The blue channel becomes the 'overlay' channel and can still specify 256 values (8 bits). Each of these values can still be chosen from Bits#'s 42-bit range (14-bits per colour channel), but only 256 of these can be assigned at one time. Values of 0 in the blue channel are 'transparent' and the pixel displays the shade of grey specified by the red and green channels. However the other 255 values can be specified by assigning a 'colour lookup table' (cLUT) and therefore can still display a limited range of colours, even in mono++ mode. This overlay channel is special as it is not cleared following a Screen('Flip', windowHandle) so stimuli only need to be drawn once (useful for stimuli such as fixation crosses).

The figure below outlines the basic operation for this mode. It shows how a pixel drawn by the graphics card (to the left) is changed in the Bits# (centre outlined box) into a high resolution pixel on the monitor (to the right).

After the value has been determined (either from the overlay or from combining the red and green sub-pixels) the signal is then gamma corrected (tall box with three curved lines) before sent to the monitor.



Bits# running in Mono++ video mode.

## 9.2 Using Mono++ mode with MATLAB

### 9.2.1 Initialising Mono++ mode

Before mono++ mode can be used, Bits# must first be set to mono++ mode. After Bits# is connected and a serial-like port open (see Communicating with Bits# using MATLAB), this can be done by giving it the command:

```
>> s1 = serial('Bits#PortNumber')
>> fopen(s1);
>> fprintf(s1, ['#monoPlusPlus' 13]);
>> fclose(s1);
```

'Bits#PortNumber' is the port number which Bits# is attached to.

### 9.2.2 Drawing in mono++ mode

#### Matrix

In mono++ mode, an image is represented by an  $x * y * 3$  matrix where  $x$  and  $y$  are the 2D dimensions of the image, and the three layers of the  $z$  dimension represent the red, green and blue channel values for each pixel.

## Red and Green Channels

In mono++ mode, the values in the red and green channels (layers 1 and 2 of the matrix) represent the MS and LS values respectively (see Introduction). Each position accepts a value between 0 and 255 (65536 possible combinations across channels, 16-bit) however Bits# has 14 bit support (16,384 levels) and so will convert any value given to it to the closest 14 bit value.

The value of the red channel (MS) is the number of times the chosen level of grey (between 0 and 66,563) can be divided by 256 completely. For example, for a light grey of 60,000 out of 66,564:

```
>> imageMatrix(x, y, 1) = floor(60000/256);
ans = 234
```

The value of the green channel (LS) is the remainder of the above division:

```
>> imageMatrix(x, y, 2) = rem(60000, 256);
ans = 96
```

If no overlay is desired, the third dimension should be set to zeros (otherwise see below for drawing an overlay image):

```
>> imageMatrix(:, :, 3) = zeros(x, y);
```

Make sure the data is in 8-bit values:

```
>> imageMatrix = uint8(imageMatrix);
```

This can then be drawn using:

```
>> image(imageMatrix);
```

The window will need to be moved so it is displayed on the screen in mono++ mode for the imageMatrix to be interpreted correctly.

## Overlay (Blue Channel)

To draw into the overlay, change the values of this third layer of the matrix to the corresponding index of the current cLUT (between 0 and 255). Setting the blue channel to 0 will cause it to be 'transparent', simply displaying the grey specified by the red and green channel values. However if the blue channel value is positive, it will display the corresponding colour from the current cLUT, regardless of the values specified in the red and green channels.

A method of demonstrating this is to draw the following square, such that the red and green channel values are all set to 255 (a [red, green] combination of [255, 255] will indicate white in the default cLUT). The bottom half of the blue channel is set to 0 (transparent) and the top half is set to 2 (almost black):

```
>> imageMatrix = zeros(100,100,3);
```

```
>> imageMatrix(:, :, 1) = 255;
>> imageMatrix(:, :, 2) = 255;
>> imageMatrix(1:50, :, 3) = 2;
>> imageMatrix = uint8(imageMatrix);
>> image(imageMatrix);
```

This should create a window filled with yellow. It might be possible to notice a subtle difference between the top half and bottom half due to the tiny amount of blue introduced to the top half, but this is unlikely to be obvious. However, if you display this window on the screen set to mono++ mode, the top half should be nearly black while the bottom half should be white (If using a single display configuration, change Bits++ modes between BitsPlusPlus mode and MonoPlusPlus mode to notice this difference).

This is because the bottom half blue channel values are set to 0 so allow the grey specified by the red and green channels (white) to be displayed. However the blue values for the top half are set to 2 (almost black) and so this value is displayed, regardless of the values in the red and green channels.

## 9.3 Using Mono++ mode with Psychtoolbox

### 9.3.1 Configuration

#### Initialising

To use some of the specialized Psychtoolbox functions, there are some further steps that need to be taken at the start of a script. To begin configuration:

```
>> PsychImaging('PrepareConfiguration');
```

#### Specifying precision of frame buffer

The first step is to specify the use of floating point 32 bit precision frame buffer. Bits# will not use all 32 bits but this is required for it to be able to utilize all 14 bits.

```
>> PsychImaging('AddTask', 'General', 'FloatPoint32Bit');
```

***An Important Note:** Depending on the graphics card used, this mode may disable alpha blending. If alpha blending is required, you may specify a different value but Bits# may only run at 10 or 11 bits. The Recommended alternative is 'FloatPoint32BitIfPossible' as this will revert to the optimum setting that still enables alpha blending, depending on the machine.*

#### Enable Mono++ mode

Finally, the Mono++ driver should be enabled so that:



```
>> PsychImaging('AddTask', 'General', 'EnableBits++Mono++Output');
```

**An Important Note:** Calling this at the start of a script does not change the Bits# to mono++ video mode but only tells Psychtoolbox how to format pixel values. Bits# needs to be set to Bits++ mode for stimuli to be presented correctly.

Alternatively, if an overlay window is desired:

```
>> PsychImaging('AddTask', 'General', 'EnableBits++Mono++OutputWithOverlay');
```

**An Important Note:** A fourth step that is not necessary for using Mono++ mode but is strongly recommended for accurate display of stimuli is to define the type of gamma correction to apply (see the gamma correction section for more information).

### 9.3.2 Opening a window for drawing

Once configuration is complete, the screen which is to be set to mono++ needs to be specified. A list of available screens can be returned using:

```
>> Screen('Screens')
```

In a single monitor setup the monitor should have the screen ID '0' whereas in a dual monitor setup (set in extended desktop mode) '0' would be both monitors, '1' the primary and '2' the secondary monitor. In the following the secondary monitor will be used as stimulus display and therefore we choose the monitor with the highest screen ID number:

```
>> whichScreen = max(Screen('Screens'));
```

Once the mono++ screen has been specified, a window can be open for drawing to. The third argument (0.5 in the below example) indicates colour and will set the entire screen to be the colour specified here (note that using PsychImaging, colour values range between 0 and 1, not 0 and 255). Therefore the current example would create blank window of a mid-grey.

```
>> windowHandle = PsychImaging('OpenWindow', whichScreen, 0.5);
```

If an overlay window is required, an additional command needs to be entered as this has a separate handle to the main window:

```
>> overlayWindowHandle = PsychImaging('GetOverlayWindow', windowHandle);
```

**An Important Note:** Some examples or documentation may use the functions:

```
>> BitsPlusPlus('OpenWindowMono++', screenID, ...)
```

Or:

```
>> BitsPlusPlus('OpenWindowMono++WithOverlay', screenID, ...)
```

However these are considered obsolete and exist for compatibility reasons. It is therefore recommended "PsychImaging(...)" is used.

## 9.4 Drawing in Mono++ mode with Psychtoolbox

### 9.4.1 Creating the Image Matrix

To draw an image to an opened window, instead of having to create the three dimensional matrix manually, the image matrix needs now to be only a 2-dimensional matrix populated with values between 0 (black) and 1 (white). Psychtoolbox will automatically convert these to their respective values of the red and green channels. For example, to create a square containing a sinusoidal grating, 200 pixels by 200 pixels:

```
>> sinGrating = sin((0:199)/4);
>> sinGrating = (sinGrating + 1)/2;
>> sinGrating2D = repmat(sinGrating, 200, 1);
```

### 9.4.2 Creating a texture

Once the image matrix has been created, it needs to be converted into a 'texture' before it can be drawn. This saves the matrix in memory, allowing for it to be drawn quickly without necessarily recreating the array on every screen refresh. Note it is important to include an output argument which will be the textureID as it is this, not your original matrix, that should be used to actually draw:

```
>> textureID = Screen('MakeTexture', windowHandle, sinGrating2D, [], [], 2);
```

***An Important Note:** Note the '2' as the 6th position during Screen('MakeTexture'... . This is the 'floatprecision' value and will determine the level of precision the texture is stored at. If left as its default value of 0, it will save the texture as 8 bits and expect values between 0 and 255 (therefore your image of values between 0 and 1 will appear black). A value of 1 specifies to store the texture at 16 bit precision and a 2 specifies 32 bits. It is recommended this be set to 2 for maximum accuracy.*

Several textures can be created but they take up memory. If there are not many textures to be created (stimuli), it may be worth creating all of the textures in advance and then only displaying them. However if there are a large number of textures (stimuli) to be created, or if error messages begin to warn that a large proportion of refreshes were missed, it may be preferable to draw the textures on the fly to free up some memory.

### 9.4.3 Drawing a texture and making it visible

Finally, textures can be drawn to the screen. Textures are not drawn directly to the visible screen but instead are drawn to a buffer screen. Textures will not be visible until these two screens are flipped. Multiple textures may be drawn before flipping the screen.

```
>> Screen('DrawTexture', windowHandle, textureIDa);
>> Screen('DrawTexture', windowHandle, textureIDb);
```

```
>> Screen('Flip', windowHandle);
```

***An Important Note:** By default, the textures will be drawn to the centre of the screen. To specify the location they should be drawn, include a four value array at the fifth input argument to Screen, indicating the edges of the image rectangle (starting with the left edge and following in a clockwise direction). Note that the index value is not the same as indexing a matrix of values. To draw just 1 pixel in the top left of the screen for example would not be [1 1 1 1] but [0 0 1 1], where the index values index the column separations rather than the columns themselves:*

```
>> LocationCoordinates = [leftEdge topEdge rightEdge bottomEdge];
```

Note that once the screens are flipped, the new buffer screen will be cleared and any textures will need to be withdrawn. If there is a simple texture that will be visible for all, or a large majority, of the presentation (such as a fixation cross), it may be quicker to draw the stimulus to the overlay window (see below).

#### 9.4.4 Overlay Window

Use the overlay window handle created with PsychImaging. The window is like a standard offscreen window except you use the index (LUT) when drawing to it. For a good example of how to use a overlay window in Mono++ mode see the script:

```
>> BitsPlusIdentityClutTest
```



# Chapter 10: COLOUR++ Mode

## 10.1 Overview

### 10.1.1 Typical colour display

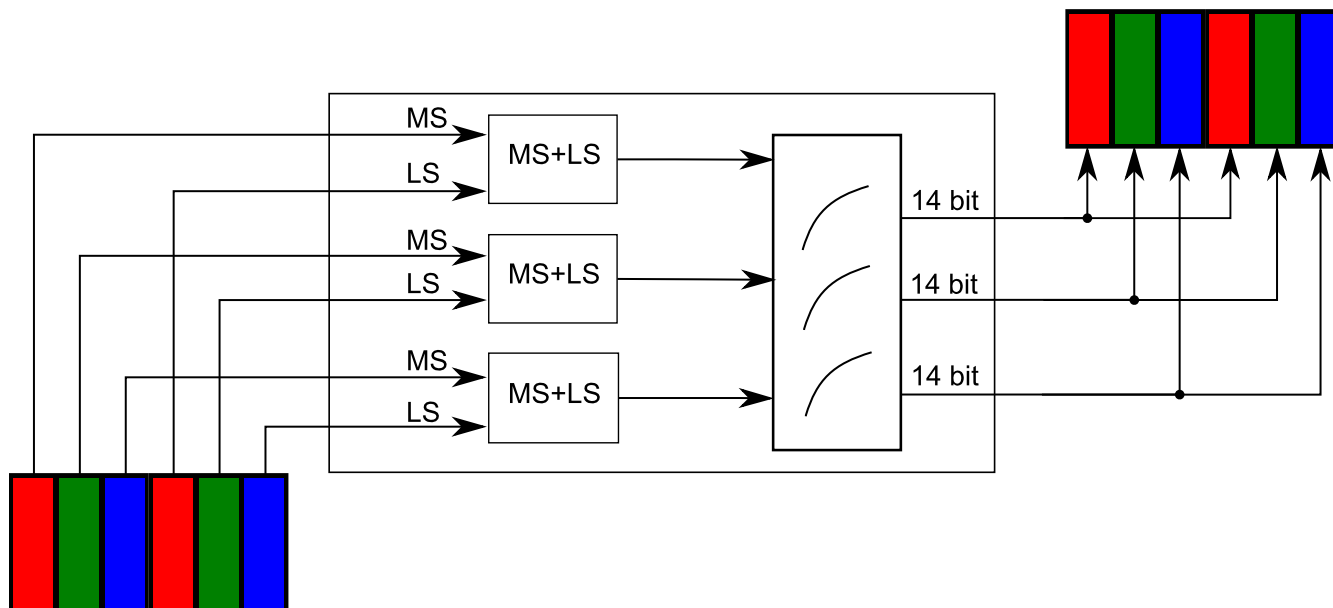
When displaying colour images, each pixel has 3 red, green and blue (RGB) values associated with it that determine the colour displayed. Each of these three colour channel values are typically '8-bit', so can be set to any of 256 different values ranging between 0 (no output from that colour gun) and 255 (full output from that colour gun). Therefore there are 16,777,216 different possible combinations of values across the three channels (24-bit), each combination specifying a different colour to be displayed.

### 10.1.2 Bits# Colour++ Mode

For some uses however, 8-bits per colour channel may not be sufficient. Colour++ mode uses combinations of values between pairs of pixels to allow for 14-bits (16,384 different values) per colour channel, and therefore 42-bits overall.

Colour values are no longer encoded to individual pixels, but across pairs of pixels. The most significant component is represented by the left pixel while the least significant component is represented by the right pixel. Although these two values will usually differ, both pixels will display the same colour specified by the combination of values.

The figure below outlines the basic operation for this mode. It shows how two pixels drawn by the graphics card (to the left) are changed in the Bits# (centre outlined box) into two high resolution pixels on the monitor (to the right). Each of the three sub-pixels are combined with the neighbouring pixel's sub-pixels (three boxes to the left inside the centre outlined box) which are then gamma corrected (tall box with three curved lines) before sent to the monitor.



Bits# running in Colour++ video mode.

It is important to note that this video mode trades off some spatial resolution for the increased colour resolution. The spatial resolution is changed in the horizontal direction so care must be taken when generating the stimulus to account for this.

If drawing simple geometric shapes, these can usually just be initially specified as half of their intended width so that their dimensions will be as intended once they have been converted. Similarly, such stimuli could be drawn with twice the vertical height as truly intended, so that the dimensions are preserved, and the screen set to a higher resolution to restore the originally intended size of the stimulus.

On a CRT there is another way to account for the spatial resolution change by using an artificial screen resolution so that it is twice the actual intended horizontal resolution. This will therefore double the horizontal refresh rate so that only one true pixel is displayed for each specified pair of pixels. However, care should be taken not to set parameters beyond the screen's capabilities as this may cause further display problems.

See 'Using Colour++ mode with Psychtoolbox' section below for further methods.

### 10.1.3 Formatting pixels for Colour++ mode

Before formatting an image for Colour++ mode, it is recommended it first be created as a normal  $x * y * 3$  matrix where  $x$  and  $y$  are the width and height of the image (in pixels) and 3 represents the 3 colour channels. These should be values in the 16-bit range (from 0 to 65,535), although note that some functions, such as those provided by Psychtoolbox allow for values to be specified between 0 and 1, not needing to be manually formatted.

Once an image has been specified as it would if each value were to correspond to a single pixel, each of the three colour values in the range of 0 to 65,535, these need to be split into their MS and LS components and each assigned to their respective pixel location.

The MS component of a value between 0 and 65,535 is number of times the value can be divided by 256, rounded down to the closest integer. The LS component is the remainder left over from such a division. These values are then assigned to pairs of horizontal pixels so that the MS value is assigned to the left pixel and the LS value is assigned to the right pixel. These values will be combined so that both pixels display the same colour, specified by both the MS and LS values.

## 10.2 Formatting Colour++ using MATLAB

### 10.2.1 Initialising Colour++ mode

First, Bits# should be set to Colour++ mode. If Bits# is set to a different video mode (such as mono++), any stimuli displayed to the screen will not be displayed properly as the other video modes interpret pixel values differently.

It might be preferable to include a few lines at the top of each script that change Bits# to colour++ mode, regardless of what mode it was in previously. This can be done via the serial port commands (see the 'Setting Up' chapter for further information) using:

```
>> s1 = serial('Bits#PortNumber')
>> fopen(s1);
>> fprintf(s1, ['#colourPlusPlus' 13]);
>> fclose(s1);
```

'Bits#PortNumber' is the port number which Bits# is attached to.

### 10.2.2 Displaying an image in colour++ mode

Once an image has been defined as a matrix of individual pixel values, each with three colour values ranging between 0 and 65,535, these values need to be split into their MS and LS components and then interleaved, each assigned to their respective position in each pair of pixels (MS on the left, LS on the right). If myImageMatrix is a matrix containing an image with the first dimension equal to the vertical resolution, the second dimension equal to the horizontal resolution and the third dimension equal to the three colours:

```
>> MScomponents = floor(myImageMatrix ./ 256);
>> LScomponents = rem(myImageMatrix, 256);
>> myFormattedMatrix(:, 1:2:size(MScomponents,2)*2-1, :) = MScomponents;
>> myFormattedMatrix(:, 2:2:size(LScomponents,2)*2, :) = LScomponents;
```

This new formatted imageMatrix can then be imaged as with any other image matrix:

```
>> image(myFormattedMatrix);
```

If then displayed on a screen, connected to Bits# in colour++ mode, each pair of pixels should be the same colour. This can be demonstrated using the script `colourPlusPlusSimpleDemo.m`. Once the figure is displayed, move it to the display screen formatted in colour++ mode and then adjust the figure width so that it is narrower, until the vertical bars become just a few thick bars or even the entire figure becomes one solid colour. As you then move the figure horizontally, the colour of these bars should swap between light and dark. This is because the pairs of pixels move between MS pixels and LS pixels. Finally, move the figure so that it is again displayed on a screen not configured in colour++ mode and the thick bars should now appear as many narrow bars, demonstrating that each pair of pixel contained different values, but that these were combined across pairs of pixels in colour++ mode.

## 10.3 Using Colour++ mode with Psychtoolbox

Instead of having to format image pixels manually, Psychtoolbox provides a range of functions that will format images automatically.

### 10.3.1 Configuration

#### Initialising

To use some of the specialized Psychtoolbox functions, there are some further steps that need to be taken at the start of a script. To begin configuration:

```
>> PsychImaging('PrepareConfiguration');
```

#### Specifying precision of frame buffer

The first step is to specify the use of floating point 32 bit precision framebuffer. Bits# will not use all 32 bits but this is required for it to be able to utilize all 14 bits.

```
>> PsychImaging('AddTask', 'General', 'FloatPoint32Bit');
```

*An Important Note: Depending on the graphics card used, this mode may disable alpha blending. If alpha blending is required, you may specify a different value but Bits# may only run at 10 or 11 bytes. The Recommended alternative is 'FloatPoint32BitIfPossible' as this will revert to the optimum setting that still enables alpha blending, depending on the machine.*

#### Enable Colour++ mode

Finally, the Colour++ driver should be enabled so that:

```
>> PsychImaging('AddTask', 'General', 'EnableBits++Color++Output', mode);
```



The 4th input argument 'mode' is compulsory and specifies the type of correction (if any) that should be applied to the displayed image in an attempt to correct for the horizontal 'stretching' effect described above.

Mode	Description	Example
0	<p>This mode will take the image as it is represented and split each pixel value across two pixels to represent the MS and LS components. This preserves the colours the image is specified in but will stretch the dimensions so that the image appears twice as wide as intended.</p> <p>The "Bits# Colour++" subsection of the above Overview includes some example methods of correcting for this.</p>	A fine grating of 1-pixel wide black and white stripes would appear as a fine grating of 2-pixel wide black and white stripes
1	<p>This method will take the image as represented but will ignore any pixel values in an odd-value column (i.e. columns 1,3,5,7...). The remaining even valued columns will then be split into their MS and LS components across the two pixels (including the odd-valued columns).</p> <p>This mode will preserve the dimensions of the image but it will lose half of its horizontal resolution.</p>	A fine grating of 1-pixel wide black and white stripes would appear as either a blank white or a blank black screen, depending on whether the white or black stripes were in the odd or even columns.
2	<p>This method will take the image as represented but will average the inputted values across each pair of pixels. It will then take this averaged value and split it into its MS and LS components across the two pixels.</p> <p>This mode will preserve the dimensions of the image, and all values will play some role in representing the image, however most of the displayed values will be different to either of the inputted values.</p>	A fine grating of 1-pixel wide black and white stripes would appear as a blank grey screen as the black and white columns would average to make a mid-level grey.

**An Important Note:** A fourth step that is not necessary for using colour++ mode but is strongly recommended for accurate display of stimuli is to define the type of gamma correction to apply (see the gamma correction section for more information).

*An Important Note: Calling this at the start of a script does not change the Bits# to colour++ video mode but only tells Psychtoolbox how to format pixel values. Bits# needs to be set to Bits++ mode for stimuli to be presented correctly.*

### 10.3.2 Opening a window for drawing

Once configuration is complete, the screen which is to be set to colour++ needs to be specified. A list of available screens can be returned using:

```
>> Screen('Screens')
```

The main computer monitor is usually 0 and with a dual-screen set up, the second screen is typically the maximum number in this list (2 if just two screens):

```
>> whichScreen = max(Screen('Screens'));
```

Once the colour++ screen has been specified, a window can be open for drawing to. The third argument (0.5 in the below example) indicates background colour and will set the entire screen to be the colour specified here. The current example would create blank window of a mid-grey.

```
>> windowHandle = PsychImaging('OpenWindow', whichScreen, 127);
```

## 10.4 Drawing in Colour++ mode with Psychtoolbox

### 10.4.1 Creating the Image Matrix

To draw an image to an opened window, instead of having to create the matrix with each value split into its MS and LS components, the image now simply needs to be defined with single pixel values for each pixel, ranging between 0 and 1, which will have 14-bit resolution. Psychtoolbox will automatically convert these to their respective MS and LS values and populate pairs of pixels with these, as determined by the 'mode' specified during initialization.

For example, to create an isoluminant mid-red square (100 pixels \* 100 pixels), it could be created as a rectangle half the desired width, in mode 0, so that it will be stretched and displayed with the intended dimensions:

```
>> mySquare = zeros(100, 50, 3);
```

```
>> mySquare(:, :, 1) = ones(100,50);
```

### 10.4.2 Creating a texture

Once the image matrix has been created, it needs to be converted into a 'texture' before it can be drawn. This saves the matrix in memory, allowing for it to be drawn quickly without necessarily recreating the array on every screen refresh. Note it is important to include an output argument which will be the textureID as it is this, not your original matrix, that should be used to actually draw:

```
>> textureID = Screen('MakeTexture', windowHandle, mySquare, [], [], 2);
```

An Important Note: Note the '2' as the 6th position during Screen('MakeTexture'... . This is the 'floatprecision' value and will determine the level of precision the texture is stored at. If left as its default value of 0, it will save the texture as 8 bits and expect values between 0 and 255 (therefore your image of values between 0 and 1 will appear black). A value of 1 specifies to store the texture at 16 bit precision and a 2 specifies 32 bits. It is recommended this be set to 2 for maximum accuracy.

Several textures can be created but they take up memory. If there are not many textures to be created (stimuli), it may be worth creating all of the textures in advance and then only displaying them. However if there are a large number of textures (stimuli) to be created, or if error messages begin to warn that a large proportion of refreshes were missed, it may be preferable to draw the textures on the fly to free up some memory.

### 10.4.3 Drawing a texture and making it visible

Finally, textures can be drawn to the screen. Textures are not drawn directly to the visible screen but instead are drawn to a buffer screen. Textures will not be visible until these two screens are flipped:

```
>> Screen('DrawTexture', windowHandle, textureID, [], [], location);
```

```
>> Screen('Flip', windowHandle);
```

*An Important Note: By default, the textures will be drawn to the centre of the screen. To specify the location they should be drawn, include a four value array at the fifth input argument ('[, location]' in the above example), indicating the edges of the image rectangle (starting with the left edge and following in a clockwise direction). Note that the index value is not the same as indexing a matrix of values. To draw just 1 pixel in the top left of the screen for example would not be [1 1 1 1] but [0 0 1 1], where the index values index the column separations rather than the columns themselves:*

```
>> LocationCoordinates = [leftEdge topEdge rightEdge bottomEdge];
```

Also note that multiple textures may be drawn before flipping the screen.

```
>> Screen('DrawTexture', windowHandle, textureIDa);
```

```
>> Screen('DrawTexture', windowHandle, textureIDb);
```

```
>> Screen('Flip', windowHandle);
```



# Chapter 11: Bits++ Mode

## 11.1 Overview

### 11.1.1 Palette-based animation

If drawing complex or large stimuli, it can take some time to redraw the entire stimulus on successive frames. This redrawing may not be necessary if the stimuli are stationary but if creating moving stimuli, such as a drifting grating, then there is a risk of the stimulus drawing taking too long and results in dropped frames. An alternative to this is palette-based animation.

Palette-based animation assigns each pixel a single value between 0 and 255, each of which index a row in a LUT. The values within this LUT can be specified in the 14-bit resolution, however only 256 different values can be specified and displayed on the screen simultaneously.

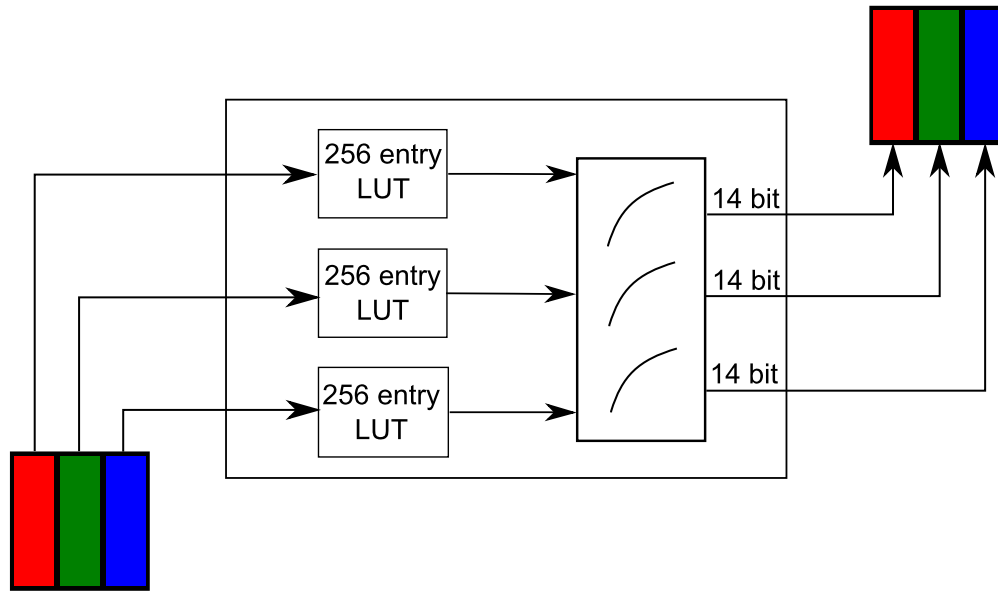
The advantage of palette-based animation is that in order to create a moving stimulus, it is not necessary to redraw the stimulus and its pixel values. Instead, the LUT values can be changed and most of the time, changing 256 rows of a LUT will be much faster than redrawing a stimulus.

Similarly, if a complex stimulus is drawn, such as a lot of lines, dots or geometric shapes, and one wants to use the same stimulus but change the colours, it would be difficult to redraw the stimulus, but one could instead simply change the LUT values.

For example, to draw a drifting sin-wave grating, the 1-dimensional values of the grating can be assigned to a cLUT and on each successive frame, shifted along (with values at one end being moved to the other end). The pixel values themselves will remain constant, however as the values associated with that index will change, it will appear as though the pixel values are changing.

The figure below outlines the basic operation for this mode. It shows how a pixel drawn by the graphics card (to the left) is changed in the Bits# (centre outlined box) into a high resolution pixel on the monitor (to the right).

Each of the three sub pixels are used to select a value in a 256 entry LUT (three boxes to the left inside the centre outlined box) which then is gamma corrected (tall box with three curved lines) before sent to the monitor.



Bits# running in Bits++ video mode.

### 11.1.2 Bits++ vs Mono++

Bits++ does not do anything that the overlay window of Mono++ can't do. However, if using the overlay window of Mono++, cLUT index 0 always represents 'transparent' and therefore only 255 other colours can be specified. Bits++ allows the user to specify 256 different colours.

## 11.2 Formatting Bits++ using MATLAB

### 11.2.1 Initialising Bits++ mode

First, Bits# should be set to Bits++ mode. If Bits# is set to a different video mode (such as mono++), any stimuli displayed to the screen will not be displayed properly as the other video modes interpret pixel values differently.

It might be preferable to include a few lines at the top of each script that change Bits# to bits++ mode, regardless of what mode it was in previously. This can be done via the serial port commands (see the 'Setting Up' chapter for further information) using:

```
>> s1 = serial('Bits#PortNumber')
>> fopen(s1);
>> fprintf(s1, ['#BitsPlusPlus' 13]);
>> fclose(s1);
```

'Bits#PortNumber' is the port number which Bits# is attached to.

### 11.2.2 Pixel Values

There is little that needs to be done with regard to formatting the actual pixel values drawn to the screen. Each pixel should be set to a value between 0 and 255. All three colour channels should be set to the same value. Multiple pixel values can be used to represent the same colour simultaneously (just assign the same RGB value combination to multiple rows of the LUT).

Therefore to draw a stimulus in Bits++ mode, all that is needed is an  $x*y*3$  matrix where  $x$  is the stimulus width,  $y$  is the stimulus height and the 3 represents the RGB layers (which should all be the same value). These numbers should be between 1 and 256.

If drawing a stimulus such as a grating, the pixel values do not need to vary in a sinusoidal manner, a linear ramp would suffice if the LUT values are set to a sinusoidal pattern. However if preferred, the pixel values can be assigned in the desired pattern and the LUT values set in a linear fashion. To draw a  $256*100*3$  matrix with a linear ramp from 1 to 256 along the  $x$  dimension:

```
>> imageMatrix(:, :, 1) = repmat(0:255, 100, 1);
```

```
>> imageMatrix(:, :, 2) = repmat(0:255, 100, 1);
```

```
>> imageMatrix(:, :, 3) = repmat(0:255, 100, 1);
```

If drawing multiple stimuli (such as dots) and want to vary their colour, each stimulus could be drawn with a different value. Some consideration should be made to the LUT when creating the stimuli. For example, should one want stimuli to vary between 25 different colours, one could set 10 rows of the LUT for each colour (0-9, 10:19, etc.). If one wants the stimuli all to change colour simultaneously, then stimuli should be assigned the pixel values in the same respective position of these 10 LUT values (for example, the first value of each colour, 0, 10, 20, 30) so that as the LUT is cycled, all stimuli LUT indexes will cross the colour boundary and get different RGB values at the same time. However, if the stimuli should change colours at different times, like a twinkling effect, then pixel values should be assigned to different LUT positions (e.g. 0, 4, 12, 18, 21) so that as the LUT is cycled, values will cross the boundary between colours at different times.

*An Important Note: Regardless of what the stimuli being drawn is, it is often worth considering any colours that need to be kept constant, such as a fixation cross or background colour, as these will need to be assigned LUT indices that will be kept constant, and so should not be included in your calculations of how many LUT values can be used for stimulus drawing.*

### 11.2.3 LUT values

#### Creating a LUT matrix

The LUT should be a matrix with 3 columns and 256 rows. Each row represents a LUT index and each column represents the Red, Green and Blue channels respectively. Each RGB value can be specified as pixel values would normally be specified. For example, black is [0 0 0], white is [1 1 1] and full intensity red is [1 0 0]. Using Bits#, each cLUT index can be specified to 14-bit resolution. Some examples are below.

To specify a simple 8-bit linear greyscale LUT:

```
>> myLinearLUT = repmat(linspace(0,1,256)', 1, 3);
```

To create a linear 8-bit red cLUT (varying from black to full intensity red):

```
>> myRedCLUT = [linspace(0,1,256)' zeros(256,2)];
```

To create a cLUT populated by random colours (14-bit resolution) on every row:

```
>> myRandomCLUT = randi((2^14),256,3)/(2^14);
```

#### Modifying a LUT matrix

It is often helpful to manually change specific rows to arbitrary values. For example, if using the random colour cLUT above, but wanting to present stimuli on a mid-grey background, it might be useful to assign the first row (for example) to represent the background colour:

```
>> myRandomCLUT(1, :) = [0.5 0.5 0.5];
```

#### Cycling through a LUT matrix (palette-based animation)

If presenting a dynamic stimulus, such as a drifting grating, or geometric shapes that change colour over time, it is not necessary to redraw the pixel values to the screen on each frame, as in other modes. Instead, it is possible to simply change the values within the LUT so that the row a pixel value indexes will contain different values to display.

This can be done a number of ways, and not all rows must be changed each time. However perhaps the most common use is to 'cycle' the values, shifting them up or down 1 row at a time, with the top/bottom row being moved to the other end of the LUT. For example, to shift the values in the linear greyscale LUT up one row (and the top row being moved to the bottom):

```
>> myLinearLUT = [myLinearLUT(2:end,:); myLinearLUT(1,:)];
```

This could then be placed within a 'for loop' or a 'while loop' and will cycle through the values on successive iterations.



## Keeping some values constant while cycling through a LUT matrix

However, if wanting to keep one or more rows a constant colour, such as a fixation cross or background colour, remember not to change the relevant row of the LUT.

For example, with the random colour cLUT created above, with the first row representing the mid-grey background colour, if all values were cycled like in the previous example, the background would also cycle through all the random colours. In order to shift the colours up one row, while keeping the background colour constant:

```
>> myRandomCLUT(2:end, :) = [myRandomCLUT(3:end, :); myRandomCLUT(2, :)];
```

This example does not change the values in the first row, only those of the second row and below. It should be noted that it is much easier to keep rows constant if they are at the very top, or very bottom, rather than at an arbitrary point in the middle. As colours can be assigned to any row, and mid-grey does not need to be assigned to the middle LUT row, it is advised values to be kept constant are assigned to the top, or bottom rows.

## Applying the LUT

Creating a LUT matrix will not automatically update the values represented by each pixel value. Similarly, simply cycling through a LUT matrix will not actually update the currently uploaded LUT. Each new iteration of the LUT will need to be uploaded or applied in order for it to take effect.

Without using Psychtoolbox, there is no way of uploading this created LUT to Bits# in its current format (as a 256 row, 3 column matrix). However, it can be encoded to a T-Lock as described in the “Creating a Colour Look-up Table (cLUT)” chapter above. Therefore, to create a moving stimulus in Bits++, a T-Lock should be encoded to each video frame updating the LUT values for the following frame.

Alternatively, Psychtoolbox provides some functions that can be used to upload a created LUT without having to manually populate a T-Lock line.

## 11.3 Using Bits++ mode with Psychtoolbox

### 11.3.1 Configuration

#### Initialising

To use some of the specialized Psychtoolbox functions, there are some further steps that need to be taken at the start of a script. To begin configuration:

```
>> PsychImaging('PrepareConfiguration');
```

## Specifying precision of framebuffer

In the other 2 video modes, the framebuffer precision is specified to 32-bit precision (allowing a final resolution of 14-bits). However as the pixel values in Bits++ mode are 8-bits, this step should be skipped here in case it may lead to rounding errors.

### Enable Bits++ mode

To use Bits++ mode, it is necessary to specify that Psychtoolbox should format the pixel values for Bits++ mode:

```
>> PsychImaging('AddTask', 'General', 'EnableBits++Bits++Output');
```

*An Important Note: Calling this at the start of a script does not change the Bits# to Bits++ video mode but only tells Psychtoolbox how to format pixel values. If Bits# is set to a different video mode (such as mono++), any stimuli displayed to the screen will not be displayed properly as the other video modes interpret pixel values differently. It should be ensured Bits# is in Bits++ mode before displaying stimuli.*

### Enable cLUT based mapping

As pixel values are themselves not values to display, but indexes of a cLUT, this should be specified during the configuration:

```
>> PsychImaging('AddTask', 'General', 'EnableCLUTMapping');
```

### Gamma Correction

A fourth step that is not necessary for using Bits++ mode but is strongly recommended for accurate display of stimuli is to define the type of gamma correction to apply (see the gamma correction section for more information).

#### 11.3.2 Opening a window for drawing

Once configuration is complete, the screen which is to be set to Bits++ mode needs to be specified. A list of available screens can be returned using:

```
>> Screen('Screens');
```

The main computer monitor is usually 0 and with a dual-screen set up, the second screen is typically the maximum number in this list (2 if just two screens):

```
>> whichScreen = max(Screen('Screens'));
```

Once the Bits++ screen has been specified, a window can be opened for drawing to. The third argument (0.5 in the below example) indicates colour and will set the entire screen to be the colour specified here (note that using PsychImaging, colour values range between 0 and 1, not 0 and 255). Therefore the current example would create blank window of a mid-grey:

```
>> windowHandle = PsychImaging('OpenWindow', whichScreen, 0.5);
```

## 11.4 Drawing in Bits++ mode with Psychtoolbox

### 11.4.1 Uploading the LUT

Before attempting to draw stimuli in Bits++ mode, it is important to ensure the desired LUT has first been uploaded. If a stimulus is displayed to the screen before a LUT has been uploaded, it will be displayed with whatever values are saved in the current LUT and may therefore not appear as intended.

Once you have created the 256 rows \* 3 columns LUT matrix (in the following example, assigned to the variable name `myLinearLUT`), this can be uploaded to Bits# using the function:

```
>> Screen('LoadNormalizedGammaTable', windowHandle, myLinearLUT, 2);
```

The '2' as the 4th input argument indicates that this is a LUT to be uploaded to Bits#, not a gamma correction table.

If cycling through LUT values, this function needs to be called on every iteration for the changes to take effect.

### 11.4.2 Creating the Image Matrix

To draw an image to an opened window, instead of having to create the matrix with 3 RGB levels, the image now simply needs to be defined with single pixel values for each pixel, ranging between 0 and 255. Psychtoolbox will automatically expand these values across all 3 channels. For example:

```
>> imageMatrix = repmat(1:256, 100, 1);
```

### 11.4.3 Creating a texture

Once the image matrix has been created, it needs to be converted into a 'texture' before it can be drawn. This saves the matrix in memory, allowing for it to be drawn quickly without necessarily recreating the matrix on every screen refresh. Note it is important to include an output argument which will be the textureID as it is this, not your original matrix, that should be used to actually draw:

```
>> textureID = Screen('MakeTexture', windowHandle, imageMatrix);
```

Several textures can be created but they take up memory. If there are not many textures to be created (stimuli), it may be worth creating all of the textures in advance and then only displaying them. However if there are a large number of textures (stimuli) to be created, or if error messages begin to warn that a large proportion of frames were dropped, it may be preferable to draw the textures 'on the fly' to free up some memory.

#### 11.4.4 Drawing a texture and making it visible

Finally, textures can be drawn to the screen. Textures are not drawn directly to the visible screen but instead are drawn to a buffer screen. Textures will not be visible until these two screens are flipped:

```
>> Screen('DrawTexture', windowHandle, textureID, [], location);  
>> Screen('Flip', windowHandle);
```

*An Important Note: By default, the textures will be drawn to the centre of the screen. To specify the location they should be drawn, include a four value array at the fifth input argument ('[, location]' in the above example), indicating the edges of the image rectangle (starting with the left edge and following in a clockwise direction). Note that the index value is not the same as indexing a matrix of values. To draw just 1 pixel in the top left of the screen for example would not be [1 1 1 1] but [0 0 1 1], where the index values index the column separations rather than the columns themselves:*

```
>> LocationCoordinates = [leftEdge topEdge rightEdge bottomEdge];
```

Also note that multiple textures may be drawn before flipping the screen.

```
>> Screen('DrawTexture', windowHandle, textureIDa);  
>> Screen('DrawTexture', windowHandle, textureIDb);  
>> Screen('Flip', windowHandle);
```

# Chapter 12: Communication and Synchronisation with external equipment

## 12.1 Overview

Bits# can generate video line synchronous, TTL-compatible voltages. These are useful if you want to know the accurate timing of an event such as the onset or offset of a stimulus, and you want to send this information to another system for control and data synchronisation. The voltages are created using a standard digital binary high/low electrical signalling scheme: between 0 V and 0.4 V is "low" and between 2.6 V and 5 V is "high". Bits# has 11 digital output channels – one on the Trig Out BNC connector and ten on the Digital I/O DB25 connector (see Appendix for the specification of the pins). The status of each digital output line is controlled using the Bits# Data Packet. This works in a similar way to the Bits# T-Lock LUT loading mechanism and consists of specially encoded RGB pixels that are transmitted to Bits# in the DVI video stream. The video line which is used to transmit the Data Packet is blanked and set to RGB=[0,0,0] by Bits# so the encoded pixels on that line will not be directly visible on the stimulus monitor.

An example of a data packet is provided in the Appendix which contains an example of the RGB pixel values to encode a Data Packet matrix to achieve a simple, 1 ms trigger pulse on Bits# Digital Output line 0, with additional explanations of the values that should be put in each pixel location. It will be useful to refer to this example while reading the rest of this section.

Note that although the tables in the following pages show the RGB pixel values in two dimensions, the red, green and blue channel values should extend along the z axis, not the y axis.

## 12.2 Important Considerations

The information contained within the data packet will not take effect until the frame after it has been received by Bits# (note that this may not necessarily be the frame it was specified to be if that frame was dropped or missed). Therefore if displaying the stimuli A, B, C, D on successive video frames, and the Data Packet information is created in the same buffer as stimulus A, the Digital Output lines will not activate until the next video frame, when stimulus B is displayed.

Therefore, some thought should be given as to when the Data Packet is transmitted. Perhaps the most significant concern is the risk of a dropped video frame: if a video frame is not drawn in time it may miss its intended vertical refresh, not drawn until the one after that, and the existing stimulus (including the Data Packet) will remain on the output for an additional video frame.

### 12.2.1 Time between two Data Packet events

If your priority is to know the time between the onsets of two different stimuli, then the Data Packets can be drawn on the same video frame as each stimulus and although the triggers themselves will be delayed by 1 frame, this will be the same for both and therefore the time difference between the events will still be the same.

### 12.2.2 Time between an event encoded by the Data Packet and an event that occurs on the Bits# Digital Input

If you want to know the timing of an external event with respect to the stimulus, such as a button press from a response box, not encoded to a T-Lock and therefore without this 1 frame delay, and to compare it to the onset of a stimulus, there are two options. The T-Lock could be drawn on the frame before the stimulus is to be displayed so that the trigger will activate on the next frame, which will be the frame the stimulus is actually first presented for. However, there is a risk that if this frame is dropped and therefore the stimulus delayed by a further frame, the trigger will still activate, before the stimulus has actually been displayed.

### 12.2.3 Trigger to activate another event synchronized to stimulus onset

If a trigger pulse is required to activate another event, such as another piece of equipment, and synchronization is important, then drawing a T-Lock to the same frame as the stimulus is displayed means that the second event will not activate until 1 frame (or 2 frames if the next frame is dropped and therefore delayed) after the stimulus onset, which could be a significant delay if intending for synchronized onsets. Therefore it may be preferable in this situation to draw the T-Lock to the frame before the stimulus so that the event activated by the trigger will occur synchronous with stimulus onset. There is still the risk of a dropped frame, and the trigger activating before the stimulus is truly displayed, however steps can be taken to reduce this probability.

## Double Triggers

It should be noted that if the frame after a T-Lock is dropped or delayed by 1 frame, then the T-Lock will remain on the screen for an additional frame and will have the same effect as if drawn for the first time. Therefore, two triggers will activate, one on the frame after the first T-Lock and the 2nd on the frame after unintentionally delayed T-Lock. This may be good or bad depending on the effects of the trigger. If the activation of another event twice in quick succession is a concern, then this should be considered, but the two triggers could also be used in data analysis to detect dropped frames and adjust measurements to account for this.

## 12.3 The T-Lock Code

### 12.3.1 T-Lock unlock code

To indicate that a trigger pulse is wanted for that frame, a T-Lock is used. This is a series of arbitrary values in the red, green and blue channels for eight consecutive pixels. This lock itself does not contain any specific information other than indicating that a trigger is required, and it is the values of the red, green and blue channels of the following pixels that specify the parameters of the trigger.

Pixel Location	0	1	2	3	4	5	6	7
Red Channel	69	40	19	119	52	233	41	183
Green Channel	33	230	190	84	12	108	201	124
Blue Channel	56	208	102	207	192	172	80	221

### 12.3.2 Length of T-Lock

The value in the blue channel of the 9th pixel relates to the length of the data packet and therefore indicates how many of the subsequent pixels contain data relevant to the trigger.

Pixel Location	8
Red Channel	0
Green Channel	0
Blue Channel	(Length of data packet) - 1

### 12.3.3 Memory Addresses

The red channel for all subsequent pixels indicates its associated 'address' in the memory and it is therefore useful to discuss them in terms of their memory address rather than pixel location. However it should be noted that each memory address takes up two pixel locations. One will contain information while the other will be set to 0 in all three channels.

### 12.3.4 Stereo Goggle Control

The blue channel of memory address 1 is dedicated to control the stereo goggles and which eye(s) they are currently allowing light to.

Pixel Location	9	10
Red Channel	1 (Memory address)	0
Green Channel	0	0
Blue Channel	Goggle Control	0

Of the 8 bits being sent from the graphics card on that channel, bit no. 5 and 6 are used to control the goggles. The other bits are ignored. They each control one of the pins of the Stereo Goggle port and it is the status of these pins that controls which shutter eyes are open or closed:

Left Eye Status	Right Eye Status	Bit 5	Bit 6	Bits	Blue Channel Value
Open	Closed	0	0	00000000	0
Closed	Closed	0	1	00100000	32
Closed	Open	1	0	01000000	64
Open	Open	1	1	01100000	96

Controlling the stereo goggles using a T-Lock is highly recommended as it helps to prevent the goggles from becoming out of phase of the display due to dropped frames. If the goggles are simply set to switch at a synchronous frequency to the display but a frame is dropped, then the goggles will become out of phase and the wrong image will be presented to the wrong eye.

As the T-Lock takes effect on the next frame, it is still possible that if a frame is dropped, the goggles will switch eyes but the image will not change. This means that the wrong image will be displayed to the wrong eye for 1 frame. However, as the display will not be updated, the same T-Lock will also still be presented, ensuring that on the next flip when the display is updated, the goggles will remain in the state they were intended to. Although this incorrect frame is a concern if it happens too often, at the high frequencies being used, it may not be noticeable and may not cause significant problems for some purposes.

If the goggle status is not controlled in this way to be tied to the status of the display and instead simply switching eyes arbitrarily but in phase with the display switch, then a delay in the display switch would not delay the goggles and so they would become out of phase, with the wrong images being presented to the wrong eyes, until another frame is dropped.

### 12.3.5 Analogue DAC Output

Bits# has two analogue output ports which are able to produce an analogue signal between -5 and +5 Volts. Each of the two output channels are encoded with its own memory address (2 and 3) in 16 bit resolution with its 8 MS and 8 LS bits in the green and blue channels respectively. The signal level is encoded so that values between 0 and 5 Volts are between 0 and 32767 (MS=0,LS=0 to MS=127,LS=255). Values between -5 and -0.00015 Volts are between 32768 and 65535 (MS=128,LS=0 to MS=255,LS=255). Once activated, the pulse will continue until changed or reset.

Pixel Location	11	12	13	14
----------------	----	----	----	----



Red Channel	2 (memory address)	0	3 (memory address)	0
Green Channel	DAC 1 MS Value	0	DAC 2 MS Value	0
Blue Channel	DAC 1 LS Value	0	DAC 2 LS Value	0

### 12.3.6 Command

Address 6 sets the 'command', usually set to 0, telling the hardware to output the data on the Digital Output ports. If wanting to reset a response box timer, this should be set to 12. Doing so will mean it is the only thing this T-Lock does and any other information will be ignored.

Therefore it may be necessary to encode 2 T-Locks if both a response box reset and another of the T-Lock functions are required on the same frame.

Pixel Location	15	16
Red Channel	6 (memory address)	0
Green Channel	Command	0
Blue Channel	2	0

### 12.3.7 Mask

Address 7 defines the mask in its green and blue channels. The mask specifies which of the 11 pins the trigger pulse applies to. This can be particularly useful if using multiple devices outputting triggers as it may be undesirable for each trigger to overwrite the status of all of the pins, potentially overwriting a command from another device.

Pixel Location	17	18
Red Channel	7 (memory address)	0
Green Channel	Mask for DOUT 8-10	0
Blue Channel	Mask for DOUT 0-7	0

### 12.3.8 Data

Addresses 8 onwards define the data in their green and blue channels. This indicates which pins should be set to 'high' and which should be set to 'low'. If there are multiple possible stimuli being presented, a trigger indicating the onset of a stimulus may be of little use without knowing which stimulus it was. Therefore different combinations of the pins can be assigned to different conditions.

Each value in the 'data' and therefore each memory address relates to 100 microsecond duration. Therefore to activate a pulse for 1 ms, 10 memory addresses will need to be set to high.

Pixel Location	19	20	21	22	23	...	216	217
Red Channel (address)	8 (0-100 microseconds)	0	9 (100-200 microseconds)	0	10 (200-300 microseconds)	...	0	107 (9000-10000 microseconds)
Green Channel	Data for DOUT 8-10	0	Data for DOUT 8-10	0	Data for DOUT 8-10	...	0	Data for DOUT 8-10
Blue Channel	Data for DOUT 0-7	0	Data for DOUT 0-7	0	Data for DOUT 0-7	...	0	Data for DOUT 0-7

## 12.4 Creating a trigger pulse using MATLAB

### 12.4.1 Creating the T-Lock matrix

#### Calculating total length

It may be useful first to calculate the required length of the trigger matrix (this is not the same as the length of the data packet). The T-Lock code will always require 8 pixels, and the stereo goggle control, Analogue DAC output, command, and mask data will require two pixels each, giving a total of at least 18. The remainder of the data packet is the data for each of the 100 microsecond packets for that frame. Even if the duration of the pulse is only to be for part, not all, of the frame, enough data points still need to be specified to fill the entire frame. We can calculate how many such packets are needed by establishing the frame rate:

```
>> frameRate = Screen('FrameRate', Screen );
>> NumberPackets = round((1/frameRate)/(1/1000000 * 100));
```

Therefore the total length of the trigger matrix is  $18 + (\text{NumberPackets}) * 2$ . Given the large number of data points that will be set to 0, it is recommended you initially create a matrix of the correct size filled with 0s. For example, if the frame rate of the screen is 100Hz:

```
>> frameRate = 100;
>> NumberPackets = round(10000/frameRate)
ans = 100
>> triggerMatrix = zeros(1, ((2 * NumberPackets) + 18), 3);
```

#### The T-Lock code

Once the trigger matrix is created, the first 8 pixels can be populated with the T-lock code:

```
>> BitsPlusPlusDataPacketUnlockCode = cat(3, [69 40 19 119 52 233 41 183], [33 230 190 84  
12 108 201 124], [56 208 102 207 192 172 80 221]);  
>> triggerMatrix(1, 1:8, :) = BitsPlusPlusDataPacketUnlockCode;
```

### Setting the Stereo Goggle Status

The stereo goggle status is 1 so this is what the red channel should be set to. The green channel is unused so should be set to 0. The blue channel contains the value that is to specify the command to the goggles. The 4 numbers and the status they represent are given in the table above (0, 16, 32, 48). Therefore, to close the left eye but open the right eye would be 32 ('100000'):

```
>> triggerMatrix(1, 10, :) = cat(3, 1, 0, 32);
```

Alternatively, if controlling the stereo goggles using the Digital I/O Port instead of the Stereo Video Goggle port, then this memory address will have no effect and instead the status of the 5th and 6th bits of the data packet itself will control the goggle status. See below for further information on how to set the status of each bit (pin) in the data packet.

### Setting the Analogue DAC Outputs

There are 2 analogue DAC outputs. Analogue Output Ports 1 and 2 are represented by memory addresses 2 and 3 respectively. The red channel represents the memory address (2 or 3), the green channel for each location represents the MS values of each output and the blue channel for each location represents the LS values of each output.

Analogue DAC outputs are 16-bit numbers (0 to 65535), representing voltages between -5 and 5. The values between 0 and 32767 represents voltages between 0 and 5 Volts, while values between 32768 and 65535 represents voltages between -5 and ~0 Volts. To calculate the value needed for a desired voltage:

if "VOLTS" is between 0 and 5:

```
>> dacOut = round(65534*(VOLTS/10));
```

else if "VOLTS" is between -5 and 0

```
>> dacOut = round(65535*(10+VOLTS)/10);
```

To separate the value in MS and LS for the two channels, simply:

```
>> dacMS = floor(dacOut/256);
```

```
>> dacLS = rem(dacOut,256);
```

The values can then be filled in the trigger matrix:

```
>> triggerMatrix(1, 12, :) = cat(3, 2, dacMS, dacLS);
```

```
>> triggerMatrix(1, 14, :) = cat(3, 3, dacMS, dacLS);
```

*An Important Note: Notice that columns 11 and 13 are unused and so all three colour channels remain set to 0.*

### Creating the T-Lock matrix – Specifying total length of data packet

The blue value of the 9th pixel indicates the length of the data packet. The red and green channels should be set to 0 and the blue channel set to the length of the data packet, minus 1. This refers to the number of memory addresses to be assigned, not the pixel length and so should include the stereo goggle control, analogue DAC output, command, mask and data packet.

```
>> triggerMatrix(1, 9, :) = cat(3, 0, 0, (NumberPackets + 5 - 1));
```

### Setting the Command Parameter

The green channel of the pixel assigned to memory address 6 specifies the 'command'. This is set to 0, telling the hardware to output the data on the Digital Output ports. However, if using this trigger to reset a response box timer, this value should be set to 12, but this will then be all the trigger does and all other values will be ignored (see ResponseBox section for more details). The blue channel should be set to 2:

```
>> triggerMatrix(1, 16, :) = cat(3, 6, 0, 2); % or
```

```
>> triggerMatrix(1, 16, :) = cat(3, 6, 12, 2);
```

### Setting the Mask

The green and blue channels of the pixel assigned to memory address 7 defines the mask. As previously mentioned, the mask is useful as it allows the user to specify specific pins the trigger should apply to, not necessarily overwriting all of them. The 11 pins are represented by two binary strings. The 10 pins are represented in the first 10 positions and the 'Trigger Out' pin is represented at position 16. Therefore the first 8 pins of the I/O trigger port are represented in the blue channel (e.g. '11111111'), while the last two of the I/O trigger port pins and the 'Trigger Out' port pin are represented in the green channel (e.g. '10000011'). It does not matter what values are entered to positions 11-15, they are ignored. If a pin is set to 1, its status (high or low) will be overwritten by whatever the trigger data specifies for that pin.

However if a pin is set to 0, then regardless of what the current trigger data may specify, the pin will maintain its current activity (i.e. remain high if already high or remain low if already low).

For example, the following would only overwrite the activity of pins 1 and 10, leaving the others unaffected by the current trigger:

```
MaskAGreenChannel = '00000010'
```

```
MaskABlueChannel = 00000001'
```

These binary strings are not written to the trigger matrix directly though, but instead their decimal equivalent value is. The function 'bin2dec' is particularly useful for calculating the required values. For example, for the above mask, to only affect pins 1 and 10:

```
>> bin2dec('00000010')
```

```
ans = 2
```

```
>> bin2dec('00000001')
```

```
ans = 1
```

*An Important Note: If only sending triggers from the 'Trigger Out' pin, the value of the green channel will always be 64 ('10000000') and the blue channel will always be 0 ('00000000').*

If the 'Trigger Out' port is not connected or will not be used at all, then the additional numbers in the green channel are unnecessary and can instead be thought of as just 2 bits, such as '11' for the last two I/O pins (therefore a maximum value of 3, for both pins).

Therefore, the values written to the RGB channels of the pixel to be assigned to the mask would be:

```
>> triggerMatrix(1, 18, :) = cat(3, 7, 2, 1);
```

*An Important Note: Note that the pixel value 18. Nothing was written to pixel value 17 as that was the blank*

## Setting the Data Packet

The remaining pixels and memory addresses represent the data the trigger should contain. Like the mask, this is represented in the green and blue channels as two binary strings with each position representing one of the pins (DOUT 0-9 are represented by the lower 10 bits, while DOUT10 (Trigger Out) is represented by the 16th bit).

A 1 indicates a pin should be set to 'high' while a 0 indicates a pin should be set to 'low'. For example, green and blue values of '00000000' and '00000001' would set only the first pin to high and all other pins to low. This may be the simplest form of trigger and may suffice for tasks such as indicating the onset of a stimulus when there is only one stimulus being presented. However if there are many possible stimuli for example, or many possible conditions (such as

stimuli presented in different locations), a trigger only indicating something was on the screen, without indicating what was on the screen, may be of little use.

Therefore it may be useful to assign pin 1 to indicate whether a stimulus was present on screen and different patterns of combinations of the other 10 pins to indicate what that stimulus was. For example:

Stimulus A = '00000000' '00000011'

Stimulus B = '00000000' '00000101'

Stimulus C = '00000000' '00000111'

Stimulus D = '00000000' '00001001'

As with the masks, the binary strings themselves are not assigned to the trigger matrix, but their decimal equivalent. For example, to indicate stimulus D was presented:

```
>> bin2dec('00001001')
ans = 9
>> triggerMatrix(1, 20, :) = cat(3, 8, 0, 9);
```

**An Important Note:** If an 'on' pin 1 should indicate the presence of a stimulus, with combinations of the other pins indicating which stimulus, only use values that are (multiples of 2)-1 (i.e. 1, 3, 5, 7, 9 and so on) as these will create a binary string with a '1' as the far right value (pin 1):

```
>> dec2bin(1)
ans = 00000001
>> dec2bin(2)
ans = '00000010'
>> dec2bin(3)
ans = '00000011'
>> dec2bin(5)
ans = '00000101'
>> dec2bin(7)
ans = '00000111'
```

Each data pixel represents the activity of one 100 microsecond packet. Therefore the data values should be replicated to enough memory addresses for the desired duration of the pulse. For example, for the above pulse, indicating Stimulus D, to trigger for 1ms, 10 consecutive memory addresses would need to be set:

```
>> triggerMatrix(1, 20:2:38, :) = cat(3, [8:17], [zeros(1, 10)], [ones(1,10)*9]);
```

Finally, the remainder of the memory addresses need to be set (plus 7 because the first packet's memory address started at 8, not 1), even though the data values will all be 0, so that enough 100 microsecond packets are defined to fill the entire frame:

```
>> triggerMatrix(1, 40:2:length(triggerMatrix), 1) = 18:NumberPackets + 7;
```

Once this matrix is completed, it could be written to a row in another image matrix and drawn:

```
>> otherImage(2, :, :) = triggerMatrix;
```

***An Important Note:** A T-lock will blank out the line of pixels it is written to (to prevent the other RGB values from producing odd looking colours). If this onset/offset of a black line at the top of the screen may be a problem, there are ways of fixing it. For example, it may be useful at the start of the experiment to draw a black rectangle to the overlay window for the rows at the top of the screen that may be blanked out. The overlay window does not get cleared after each Screen('Flip', windowHandle) so does not need to be redrawn each time. When a T-lock code is detected, the line will still blank out but as it will already be black, this change will no longer be noticeable.*

```
>> image(otherImage);
```

When this image is then displayed on the screen set to Mono++, a pulse should trigger (evidenced by the blacking out of the pixel row the T-lock was 'displayed' on). However this may not work without Psychtoolbox (see below) as there are a number of transformations to pixel values done by the computer that bypass MATLAB and are difficult to control without the dedicated Psychtoolbox functions.

## 12.5 Creating a Trigger Pulse using Psychtoolbox

If Psychtoolbox is used, it is not necessary to create the t-lock matrix manually as functions are provided that will automatically create such a matrix given some input parameters:

```
>> BitsPlusPlus('DIOCommand', window, repetitions, mask, data, command [, xpos, ypos]);
```

Window is the handle of the display window. Repetitions is the number of Screen('Flip', windowHandle) it should continue to trigger for (setting this to -1 will set the pulse to continually trigger until reset).

### DAC output control

There are no input parameters for the analogue DAC output function or stereo goggle control. This is a newer feature and Psychtoolbox has not yet implemented support for them. In this case the T-lock Matrix must be populated as shown above and drawn manually (i.e. the Screen function) instead of using the BitsPlusPlus function.

### Stereo Goggle Control

Psychtoolbox does also not have an explicit option for specifying the stereo goggle control. However they can still be driven using the BitsPlusPlus function if they are plugged into the 'Digital I/O port'. If this port is required by another piece of equipment though then the stereo

goggles will need to be controlled by populating the T-Lock Matrix manually like the DAC output control.

## Setting the Mask and Data

Data is an array of decimal values representing 16-bit binary strings, indicating the status of each pin. It is important to note that using this function, data must be a 248 element vector, even though the number of 100 microsecond packets will be less than this for most screens. It is therefore important still to calculate the number of such packets in one frame and not to exceed this, instead setting any unused slots to 0. Remember the "Trigger out" is on bit 16 so to set it high you should provide the decimal 32768:.

```
>> bin2dec('1000000000000000')
```

## Setting the Location

xpos and ypos indicate the position to draw the T-lock. This function assigns it to the third pixel row by default, but it is recommended it be drawn to the second pixel row to minimize the amount of screen lost.

Example: 1ms trigger

Therefore, to create a trigger that will apply to the first 8 pins ('11111111', 255), but only cause pin 1 to pulse ('00000001', 1), for 1ms for the next Screen('Flip', windowHandle) only:

```
>> myData = ones(1,10);
```

```
>> BitsPlusPlus('DIOCommand', windowHandle, 1, 255, myData, 0, 1, 2);
```

*An Important Note: Note that as the T-lock will not be drawn until after the next Screen('Flip', windowHandle), the trigger should be set up the frame before the event is due to appear on screen.*

## Clearing the trigger pulse matrix

To prevent a continuous pulse-train, particularly when repetitions is set to -1, Psychtoolbox contains a function that will clear the trigger matrix and so prevent further pulses being triggered:

```
>> BitsPlusPlus('DIOCommandReset', windowHandle);
```

Alternatively, this can be cleared by calling another Trigger that sets all pins back to 'low' (0):

```
>> myResetData = zeros(1, 248);
```

```
BitsPlusPlus('DIOCommand', windowHandle, 1, 255, myResetData, 0, 1, 2);
```



# Chapter 13: Stereo Display

## 13.1 Overview

### 13.1.1 Alternate Lines (Interleaved) Stereo

To display interleaved stereo images, each image is presented on alternate lines so that, for example, the left eye image is displayed on pixel lines 0,2,4,6,8, etc. while the right eye image is displayed on pixel lines 1,3,6,7,9, etc. This may be alternate rows or alternate columns.

When displayed on an appropriate screen (such as the Cambridge Research Systems 3D BOLD Screen), the light from each of these alternating lines will have a different polarization which, when viewed with the appropriate glasses, will allow only light of different polarizations to enter each eye. Interleaved stereo is the type of stereo display generally used in cinemas and some recent home televisions.

### **Spatial and Temporal Resolution**

This method of stereo has the disadvantage that the screen loses half its horizontal or vertical resolution, depending on whether it is alternate rows or columns (e.g. 1080 rows overall becomes 540 vertical pixels per image). However it does have the advantage that both images are displayed to both eyes simultaneously and therefore without the potential temporal effects that may be a concern if using alternate frame stereo presentation (see below).

### **Dropped Frames**

Interleaved stereo also is less affected by dropped frames than alternate frames stereo. If presenting a static image, dropped frames have no effect (unless additional information was encoded to that frame such as a trigger or cLUT T-Lock). If presenting a moving image or a movie, it may be noticed as a brief jitter, but no worse than if using alternate frames stereo or ordinary 2D displays, and the correct images will still be presented to the correct eye throughout.

### **MRI**

Interleaved stereo is also preferable for use in an MRI scanner as the passive glasses are inherently MRI compatible. Active shutter goggles required for alternate frames stereo (see below) may cause some issues as they contain circuitry that could interfere, or be interfered by the magnetic field.

## **Ghosting**

Passive polarized glasses however will often have some light leakage. This is where some of the polarized light of one image manages to leak through the filter for the other eye. If displaying 3D images, this can cause 'ghosting' (where parts of the image may look similar as without the glasses when able to see both images simultaneously). This is often subtle but may be of a concern if sharp contours are important, and if using high contrast stimuli (such as a white stimulus on a black background), when it is most noticeable. Similarly if presenting different images such as for interocular suppression, some of each image may be faintly visible to the wrong eyes.

### **13.1.2 Alternate Frames Stereo**

Alternate frames stereo display displays two different images on alternating frames. This does not require a specialized screen however it does require active shutter goggles. Active shutter goggles will selectively block the light to one eye at a time, usually at a high refresh rate. This should be synchronized to the display of the images so that one image is always displayed when the glasses allow light to one eye, and the other image is displayed when the glasses allow light to the other eye.

## **Spatial and Temporal Resolution**

Alternate frames stereo has the advantage that each image can be displayed at full vertical resolution. An image 1080 pixels high can be displayed as 1080 pixels high, unlike interleaved stereo which would halve the vertical resolution (if using alternate rows) to 540. However, this means that at any given moment in time, only 1 image is being presented to the participant and to only 1 eye. Although this is usually at very high frequencies, depending on the limits of the equipment (such as screen refresh rate) and the aims of the study, the temporal effects of this may make it poorly suited to some uses.

## **Dropped Frames**

Also the consequences of dropped frames are of far more concern. If the goggles are not synched correctly, a dropped frame may cause them to become out of phase with the display so that the wrong image is displayed to the wrong eye. This would prevent any 3D effect. Even if the goggles are able to cope with dropped frames by delaying their own switch between eyes, this will prolong the time only a single image is presented. If this were to happen to frequently, it may still lessen the 3D effect. Similarly if presenting different stimuli such as for interocular suppression, it will swap the images visible to each eye.

## **MRI**

The active shutter goggles required for alternate frames stereo (see below) are also incompatible with MRI research as most are not fMRI safe and the circuitry required to drive

them means they would be likely to interfere with, or be interfered by the magnetic field.

## **Ghosting**

The active shutter goggles are more efficient at blocking light than the passive polarized glasses used with interleaved stereo display, reducing ghosting. If presenting different stimuli such as for interocular suppression, this is when the wrong image may be slightly visible to the wrong eye. If presenting 3D stimuli, this is when parts of the image look similar as they would without the glasses, when able to see both images at once. However ghosting may still occur if the goggles are not properly synchronous with the display. If the display switches which image it is displaying before the goggles switch which eye they allow light to, then the wrong image will be visible to the wrong eye for a very brief period of time, but as this may occur on many or all frames, it may become apparent.

### **13.1.3 Others**

It should be noted that other methods of 3D display also exist:

## **Mirrors/Prisms**

It is possible to display two images on two separate screens and to use positioned mirrors or prisms to reflect the light from each of these images separately into each eye. This method can combine the advantages of the above interleaved and alternate frames methods. Both images are visible to the participant simultaneously, therefore eliminating the potential temporal concerns, but as they are presented on different screens, they can also be presented at their full resolution with no loss of vertical resolution such as when displaying both images interleaved on the same screen. However it is important that both screen's presentations are synchronized properly.

## **Video/Slide Goggles**

These are different to the active shutter goggles used in alternate frames stereo presentation. These goggles may contain small video screens, or slides, which can therefore present different images to either eye. However there are often a number of concerns that make these less preferable to other display methods. If using slides, it limits presentation to only static images. If using video, the screens may lack resolution and may be difficult to calibrate properly. Both types may also have limited field of vision.

## **Anaglyph**

Anaglyph is perhaps the stereo display people are most familiar with, although it is rarely used since the advent of more advanced methods. It uses the glasses in which one eye has a red filter and the other has a green filter (although note some glasses use different pairs of colours) and the image contains two images occupying the same spatial location on the screen, each

only visible through one of the filters. However, these are often less effective than other methods and have a number of negative characteristics such as poor colour definition and higher chance of ‘ghosting’.

## 13.2 Specifying Stereo Image Matrices with MATLAB

### 13.2.1 Creating the Image Matrices

The first step of how to display interleaved stereo images with MATLAB depends on what format the images are. Ultimately the aim is to get two matrices of pixel values that can be drawn to the screen as with any other stimulus.

If presenting simple stimuli that are possible to create artificially in MATLAB, such as gratings or geometric shapes, then they will likely already be in a matrix format and the rest of this section can be skipped.

However, often, images may be from a database, or taken using a 3D camera and will be saved as an image file. There are various image files and different functions may be required for specialised file types (see the MATLAB documentation for more information on how to read image data), however this section will describe the typical functions and their uses.

### 13.2.2 Importing Image Files

#### Reading an image file into MATLAB

Stereo images can be saved both in the same file or in two separate files. Separate files are more convenient as if they are in the same file, the relevant pixels for each image will need to be identified and saved as separate matrices.

The image files should be saved to the current directory but can then easily be read in using:

```
>> imageMatrix = imread('myStereoImage.jpg');
```

#### Converting the data type

However, this may save the image matrix as a data type such as uint8, which are less easily manipulated. In particular, if using Psychtoolbox, pixel values should vary between 0 and 1, however uint8 will round our values to the nearest 8-bit integer, causing our image to be either black or white pixels.

One can check the data type of a given variable using:

```
>> whos
```

To convert the data type to ‘double’, a data type that can be easily manipulated, use:

```
>> imageMatrix = double(imageMatrix);
```

## Converting value range

Finally, if they do not already, it needs to be ensured that these values are in the correct range for the video mode being used. If using Bits++ mode, or if not using Psychtoolbox, these will typically need to be integers that vary between 0 and 255. For example, if the values currently vary between 0 and 1, they need to be multiplied by 255 and rounded to the nearest integer:

```
>> imageMatrix = round(imageMatrix * 255);
```

***An Important Note:** Leaving the matrix in this format will only work if using Bits++ mode. If using Mono++ mode or Colour++ mode, these values will then need to be formatted in the appropriate manner (see the mono++ or colour++ chapters for further information).*

If using the Psychtoolbox drawing functions in either mono++ mode or colour++ mode, these will need to be values that vary between 0 and 1. For example, if the pixel values range from 0 to 255 by default, these will need to be converted:

```
>> imageMatrix = imageMatrix / 255;
```

## Separating Images saved in the same image file

If images are saved in separate files, this subsection can be skipped. However some files will have both images saved in the same file (such as if intended to be viewed using mirrors or prisms, see above). How to separate them will depend on whether they are horizontally or vertically adjacent. Either way, the aim is to calculate the midpoint that separates the two images so that the pixels either side of this divide can be saved as independent matrices.

The code below should work if the images are separated by a 1 pixel dividing line or no dividing line and are completely adjacent. However further manipulation may be required if the images are in a different format, for example if they were separated by a dividing line several pixels thick.

If the two images are horizontally adjacent, they can be separated using:

```
>> imageMatrixLeft = imageMatrix(:, 1:floor(length(imageMatrix(1,:)/2), :);
>> imageMatrixRight = imageMatrix(:, ceil(length(imageMatrix(1,:)/2):end, :);
```

If the two images are vertically adjacent, the same code can be used but with the first two inputs arguments swapped around:

```
>> imageMatrixLeft = imageMatrix(1:floor(length(imageMatrix(1,:)/2), :, :);
>> imageMatrixRight = imageMatrix(ceil(length(imageMatrix(1,:)/2):end, :, :);
```

Once both images are represented by separate matrices of the same size with integers between 0 and 255, they are ready to be formatted so they will be displayed to different eyes.

## 13.3 Displaying Alternate Line (interleaved) Stereo Images using MATLAB

If using Psychtoolbox, it is not possible to manually interleave the two images and the matrices can be used in their separate form. See the next section for details on how to do this.

Displaying stimuli this way has some advantages, in particular if displaying a dynamic stimulus (rather than a stationary, unchanging stimulus) as the matrices do not need to be recombined on each frame.

### 13.3.1 Interleaving the Images without Psychtoolbox

If displaying on a monitor that uses alternate pixel rows or columns with different polarizations to achieve stereoscopic display, the two matrices need to be combined into one matrix. Only half of rows, or columns, will be included from each image, so the final matrix will have the same dimensions as either of the separate matrices. The odd rows/columns of the final matrix will be the odd rows/columns of one image, while the even rows/columns will be the even rows/columns of the other image. For example, If the screen uses alternate pixel rows, this can be done using:

```
>> imageMatrixHeight = length(imageMatrixLeft(:,1));
>> finalImageMatrix(1:2:imageMatrixHeight-1, :, :) ...
= imageMatrixLeft(1:2:imageMatrixHeight-1, :, :);
>> finalImageMatrix(2:2:imageMatrixHeight, :, :) ...
= imageMatrixRight(2:2:imageMatrixHeight, :, :);
```

Alternatively if the screen uses alternate pixel columns, this can be done using:

```
>> imageMatrixWidth = length(imageMatrixLeft(1, :));
>> finalImageMatrix(:, 1:2:imageMatrixHeight-1, :) ...
= imageMatrixLeft(:, 1:2:imageMatrixHeight-1, :);
>> finalImageMatrix(:, 2:2:imageMatrixHeight, :) ...
= imageMatrixRight(:, 2:2:imageMatrixHeight, :);
```

**An Important Note:** Note that different screens may have different settings regarding whether odd or even rows/columns should represent the image for the left eye or for the right eye.

This final matrix can be displayed using:

```
>> image(finalImageMatrix)
```

The window that is created can then be displayed on the 3D screen. However this is not a particularly robust method of display and, for example, if the window resizes, then pixel

rows/columns may be deleted or added, causing the others to become out of phase and preventing the image from being displayed properly. It is therefore recommended that a more precise method is used to display this matrix, such as the CRS toolbox or Psychtoolbox.

## 13.4 Displaying Interleaved Stereo Images using Psychtoolbox

If the image matrices have been combined manually to form a single, interleaved matrix, as described in the previous section, this matrix is ready to be drawn to the screen and displayed like any other stimulus. However some video modes such as mono++ or colour++ mode may require some further formatting so see the relevant chapter for the video mode being used for full information on how to draw and display stimuli in that mode.

If the two image matrices are still separate, Psychtoolbox provides functions that will automatically interleave them. This may be advantageous, particularly if presenting one or more dynamic stimuli in which one image changes, as it saves needing to manually interleave the matrices each time.

### 13.4.1 Configuration

In addition to the usual PsychImaging configuration commands that should be called at the start of a script (see the relevant chapters for the video mode being used and the gamma correction chapter), in order to use Psychtoolbox to automatically interleave two images, there is an additional command that needs to be called. The exact command depends on whether the images should be interleaved on alternate rows or columns. To initialise the configuration phase:

```
>> PsychImaging('PrepareConfiguration');
```

If using interleaved rows:

```
>> PsychImaging('AddTask', 'General', 'InterleavedLineStereo', starttright);
```

Or if using interleaved columns:

```
>> PsychImaging('AddTask', 'General', 'InterleavedColumnStereo', starttright);
```

Whichever version being used, the 'starttright' argument can be set to either 0 (default) or 1 and specifies which rows/columns the left or right images are drawn to. If set to 0, even columns/rows will display data from the left buffer image and odd columns/rows will display data from the right buffer image. If set to 1, the opposite pattern is true.

### 13.4.2 Left and Right buffers

Psychtoolbox allows the user to use two separate buffers for drawing the two images. Within each buffer, the images can be drawn in the same way as usually drawing stimuli with Psychtoolbox (see the relevant chapter for the video mode being used), and means that the

two images do not need to be interleaved each time a change is made. This is also particularly useful if drawing multiple stimuli to the screen that may not easily be represented as a complete matrix. For example, if using the Psychtoolbox drawing functions to draw circles or diagonal lines, which are not easily saved to a matrix.

When ready to draw stimuli, select which buffer the textures are to be drawn to using the following command. `bufferIndex` can be set to either 0 (left image buffer) or 1 (right image buffer):

```
>> Screen('SelectStereoDrawBuffer', windowHandle, bufferIndex);
```

Therefore, once one image has been drawn, if another image is to be displayed to the other eye, the command should be repeated but for the other buffer.

Once all images are drawn, they can be displayed to the screen as usual:

```
>> Screen('Flip', windowHandle);
```

It does not matter which buffer is currently active when calling the 'Flip' command, it will display both buffers to the screen simultaneously.

***An Important Note:** The 'DrawTexture' function has an optional input argument for specifying the draw position. The default is to draw it in the centre of the screen with the matrices' native dimensions. However note that depending on the dimensions, the odd/even rows/columns of the matrix may displayed on the wrong rows/columns of the screen (therefore presenting the wrong image to the wrong eyes). It may be preferable to specify the drawing rectangle. However if doing so, take care to preserve the image dimensions as, if it is resized, rows/columns may be added or deleted and the others may become out of phase so that the wrong image is presented to the wrong eye, or allowing both images to be visible to both eyes.*

## 13.5 Displaying Alternate Frame Stereo Images using MATLAB

Alternate frames stereo presentation requires two different images to be drawn to alternate frames at a very high frequency. MATLAB does not have any native functions that would allow this change to be done. Therefore to display stereo images using alternate frames, a toolbox such as Psychtoolbox is required.



## 13.6 Displaying Alternate Frame Stereo Images using Psychtoolbox

### 13.6.1 Drawing the Images

To use alternate frame stereo display, the two images need to be drawn to alternating frames. This does not require any additional configuration or functions than drawing a normal image so see the chapter for the relevant video mode being used for full information on drawing in that mode.

As the alternating frames will be at a very high frequency, quick drawing is a key consideration. Therefore it is recommended that where possible, the textures are created in advance. These should then be drawn on alternate iterations of a loop (e.g. matrix A on odd iterations and matrix B on even iterations). For example:

```
>> myTexA = Screen('MakeTexture', windowHandle, imageMatrixA);
>> myTexB = Screen('MakeTexture', windowHandle, imageMatrixB);
>> for i = 1:flips
>> if rem(i, 2) ~= 0;
>> Screen('DrawTexture', windowHandle, myTexA);
>> else
>> Screen('DrawTexture', windowHandle, myTexB);
>> end
>> Screen('Flip', windowHandle);
>> end
```

### 13.6.2 Dropped Frames

Psychtoolbox prints an estimate of the number of dropped frames. As described in the overview section, dropped frames are particularly problematic for alternate frame presentation, particularly if they happen regularly. Psychtoolbox provides several suggestions in their documentation for methods of improving synchronization if large numbers of frames are missing their refresh deadline. Also, read the following section on goggle control for some ways of stopping dropped frames from causing the goggles to become completely out of phase with the stimuli, so that the wrong image is not presented to the wrong eye, but merely a short delay before the images and eyes are switched.

### 13.6.3 Goggle Control

As described in the manual, alternate frame stereo display requires the use of active shutter goggles that selectively block light to one eye or the other, in synchronisation with the two images being flipped between frames on the screen.

It is therefore important to ensure the goggles are properly synchronised with the image display, particularly in case of a dropped frame. This can be included as part of the Digital Trigger T-Lock (see the chapter Communication and Synchronisation with external equipment). This should be included on every frame to ensure the best synchronisation of the goggle status with the display status.

# Chapter 14: Analogue Data

## 14.1 Overview

In addition to being able to input and output digital signals, Bits# is able to input up to six analogue signals and output up to two analogue signals, all at 16-bit resolution. These are transmitted through the 'Analogue In' or 'Analogue Out' ports.

## 14.2 Outputting Analogue Signals

The two analogue output ports can output analogue signals between -5V and +5V with 16-bit resolution. The signals are programmed using a horizontal line of video in the video feed from the graphics card, also called the T-lock code. This allows one analogue signal level to be specified per video frame. For information about how to program the T-lock code see the chapter: Communication and Synchronisation with external equipment.

## 14.3 Receiving Analogue Signals

### 14.3.1 Initiating Data Acquisition

Analogue data is received via the serial port interface. Data acquisition is initiated by passing the command:

```
>>$Start
```

### 14.3.2 Collecting the Data

#### Using PuTTY (Windows)

PuTTY has the option of capturing data to a text file. Set the file name and logging settings under the category "Logging". Data will then be captured as it is received until it is stopped. Data is received as a string.

#### Using MATLAB

To read in a sample in MATLAB create a handle to the serial port using the "serial" command and open it using the "fopen" command. You can now repeatedly read in a sample by using the "fscan" command. The data is received as a single string (one sample) with values separated by semicolons

### 14.3.3 Stopping Data Acquisition

Data acquisition can be stopped by giving the command:

```
>>$Stop
```

#### 14.3.4 Sorting the Data

Each value within the string represents a different value and is separated by semicolons. The Analogue-In signals are the values between the 20th semicolon and the 26th (last) semicolon. Each value represents a separate analogue input port.

#### Sorting the Data using MATLAB

If a sample (text string) is saved in myData, the relevant data from the Analogue In port can be extracted with the MATLAB function "textscan":

```
>>C = textscan(myData,['%s %u %f %u %u %u %u %u %u %u %u %u %u %u %u %u %u %u %u %f %f %f %f %f %f','Delimiter',';']);  
>>myAnalogueData = cell2mat(C(21:26));
```

# Chapter 15: Response Box

## 15.1 Overview

Many uses of visual presentation require the participant or subject to make a response. This can be made in a wide variety of ways, such as a key presses, mouse clicks or even eye movements.

A typical computer keyboard would suffice for many uses but it has some weaknesses that make it ill-suited for some purposes. One reason might be the complex layout which could be a problem for some participants especially if the experiment is conducted in the dark or if the task requires to participant to focus on the monitor rather than the keyboard. But perhaps the most significant of these is its poor performance for accurate timing. The way the computer samples responses and reports their timestamps, there is often a relatively large error, with some keyboards having a non-deterministic delay of up to 35 ms.

A common alternative is a response box. This usually offers a more streamlined and ergonomic option compared to a typical keyboard and when connected to the Bits# it will be able to utilise its internal clock to achieve sub-millisecond accuracy.

The Bits# offers both wireless and wired input options for response boxes. An infra-red receiver is placed on the front of the device which is compatible the our hand held wireless response boxes. For further options, the back panel has number of digital input pins for connecting with third party response boxes (TTL-compatible).

## 15.2 Setup

For compatibility reasons the bits# emulates the protocol of the RTBox<sup>1</sup>. This means that there are seven virtual events available: four buttons: "Btn1", "Btn2", "Btn3" and "Btn4" and three auxiliary events: "Light", "Pulse" and "Trigger". None of these are physical buttons on the Bits# but merely calling names that define the available events in the protocol and are the names used by the host computer when getting response box events from Bits#.

The physical events connected to Bits# (11 wired or 6 wireless buttons) can then be "mapped" to the seven virtual events in any order defined by the user. This can be done in two ways by either configuring it in the Config.xml file located in the memory of the Bits# (recommended) or by the sending the appropriate command over the virtual serial port. For information about setting it over the serial port refer to the list of CDC commands in the Appendix.

---

<sup>1</sup><http://docs.psychtoolbox.org/PsychRTBox>  
<http://www.ncbi.nlm.nih.gov/pubmed/20160301>

The Config.xml file can be access by setting Bits# to USB-mass storage device mode. This is done by sending the command: "\$USB\_massStorage" to Bits# over the serial port (see Bits# Installation for instructions). In the mass storage device drive locate Config.xml file in the folder named "firmware". Towards the bottom of the file, there will be 8 lines that read:

```
#<Entry Btn1="Din0" />
#<Entry Btn2="Din1" />
#<Entry Btn3="Din2" />
#<Entry Btn4="Din3" />
<Entry Btn1="IRButtonA" />
<Entry Btn2="IRButtonB" />
<Entry Btn3="IRButtonC" />
<Entry Btn4="IRButtonD" />
```

Four of these lines will be preceded with a hash (#) symbol, indicating they are 'commented out', or inactive. If they where active they would specify that the first four pins on the D-sub connector The other four are active and specify that the physical buttons A through D on the wireless response box are mapped to the virtual buttons 1 though 4,

The Bits# has six wireless inputs named IRButtonX, where X is a letter from A though F corresponding the six buttons on the wireless response box.

There are 2 response button values when using a response box with Bits#. The first is one of 6 possible 'bits', which correspond to 6 digital input pins. Which button maps to which pin depends on the response box model. Each of these pins is also then mapped to one of 4 button response values (1, 2, 3 or 4). The mapping between the physical button (and its corresponding pin) and the button response value can be customized.

It is important to check in advance which buttons are mapped to which pins, and to remap them if necessary.

### 15.2.1 Mapping Pins to Button Response Values using the Bits#

The RTBox protocol Bits# can currently support up to 4 different button response values. Multiple buttons (and their corresponding pins) can be mapped to the same button response value. To set which pin should be mapped to which response value, Bits# needs to be put back into USB\_massStorage mode (see the Installation chapter for full details on how this can be done), and the 'config.xml' file, saved within the 'Firmware' folder, to be opened in an editable way, such as with a text editor such as Notepad.

Towards the bottom of the file, there will be 8 lines that read:

```

<Entry Btn1="Din0" />
<Entry Btn2="Din1" />
<Entry Btn3="Din2" />
<Entry Btn4="Din3" />
<Entry Btn1="IRButtonA" />
<Entry Btn2="IRButtonB" />
<Entry Btn3="IRButtonC" />
<Entry Btn4="IRButtonD" />

```

**An Important Note:** Only 4 “Btn” lines should be active at a time – one ‘Entry Btn1’ line, one ‘Entry Btn2’ line, one ‘Entry Btn3’ line and one ‘Entry Btn4’ line. The top 4 lines represent a digital response box while the bottom 4 lines represent an infrared response box. The inappropriate set of lines should be ‘commented out’ by preceding them with a hash symbol. For example, if using a digital response box connected using a wire:  
 <EntryBtn2="IRButtonB" />  
 would become:  
 #<EntryBtn2="IRButtonB" />

It is not simply specifying between infrared or wired response box that needs to be done here. As mentioned in the ‘Number of Buttons’ subsection above, the pins mapped to each of the 4 response values may need to be changed, depending on the box used and the button layout.

For example, the Cedrus RB-530 has 5 buttons with 1 in the centre and 4 above, below, left and right of centre. As previously mentioned, of the 6 response box pins, it is the 2nd (‘Din1’) that does not have a corresponding physical button, not the 6th. Therefore, Din1 should not be used as no button is mapped to that pin.

If wanting to make the far left button (‘Din2’ on the Cedrus RB-530 model) map to response value 1 (‘Entry Btn1’), and the far right button (‘Din4’ on the Cedrus RB-530 model) map to response value 2 (‘Entry Btn2’), the relative ‘Din’ values should be changed accordingly so the corresponding button is mapped to the intended response.

Once finished, close any serial-port connections with Bits#, turn it off and then back on again to return it to CDC mode, again allowing for communication via a serial-like port.

**An Important Note:** Remember if using a Mac, opening the Config.xml file with a text editor will change the file type to .txt. The file therefore needs to be renamed and converted back to Config.xml before turning off the Bits#.

### 15.2.2 Accurate Timings

Many response boxes will have the option of returning a timestamp in the form it is saved internally, or to be calibrated with the host computer and returned in the host computer form. Although there are various ways of increasing the accuracy of such a calibration, it is often best to measure reaction time as the difference between two events. For example, a digital trigger could be sent at the onset of a stimulus and compared with the timestamp of a button press.

## 15.3 Using a response box with MATLAB

### 15.3.1 Using a response box with Psychtoolbox

Psychtoolbox includes the 'PsychRTBox' function, which makes interfacing with the response box far simpler. Only the most important functions are described here however there are many, more advanced, functions and additional arguments. See the Psychtoolbox documentation for further possibilities if needed.

### 15.3.2 Connecting to the Response Box

To connect to the response box, use the function:

```
>> handle = PsychRTBox('Open', [, RespBoxPortName]);
```

The RespBoxPortName parameter may be optional if only one response box is connected, it will be detected automatically. However if more than one response box is connected, the target one needs to be specified here. This will typically be the same port name as Bits# is connected to.

The variable name used for the output argument value will be the handle used for all subsequent communication with the response box. However if only one response box is connected, this variable can also be omitted from all further commands.

**An Important Note:** If using a Mac, PsychRTBox function may not work. The port name that Bits# is connected to, as used in other functions (e.g. /dev/tty.usbmodemfd131) will not work with the PsychRTBox command. Instead, the 'cu' port needs to be used. This is the same as the usual port name except that the 'tty' is replaced with 'cu'. The above example would therefore need to be:

```
>> handle = PsychRTBox('Open', /dev/cu.usbmodemfd131');
```

However, even if only one response box is connected, if this port-name parameter is left blank, the response box will not be found automatically. This is because the PsychRTBox script searches for files starting with '/dev/cu.usbserial', which the Bits# port name does not. There are two ways of overcoming this. The simplest is to specify the port name on each call



*of the function. However, it is also possible to edit the Psychtoolbox code to include files starting with '/dev/cu.usbmodem' in its automatic search. NB. If any other parts are altered incorrectly, the function may fail to function properly in other ways.*

*To open the PsychRTBox script, use:*

```
>> edit PsychRTBox
```

*Line 2784 should read:*

```
candidateports=dir('/dev/cu.usbserial*');
```

*To expand this search to include the port Bits# is connected to, change this line to:*

```
candidateports=[dir('/dev/cu.usbserial*'); dir('/dev/cu.usbmodem*')];
```

*Be careful to check spelling and character case. However, once changed and saved, PsychRTBox should find the response box port (Bits# port) automatically if only one is connected and the parameter is left blank:*

```
>> handle = PsychRTBox('Open');
```

### 15.3.3 Calibrating host clock and box clock

Due to a number of factors, two clocks will not run at exactly the same time and will gradually drift apart over time. Therefore, if wanting to use host clock timings instead of box clock timestamps, it is important to calibrate the two clocks using:

```
>> PsychRTBox('ClockRatio', handle);
```

This will calculate the speed difference between the two clocks and correct all timestamps by this factor, allowing for the timing accuracy required of reaction time experiments.

### 15.3.4 Monitoring for response events

The response box does not always automatically monitor for responses and even once a connection has been opened, it can be set to be dormant if needed. In order to activate the response box to monitor and store response events, use:

```
>> PsychRTBox('Start', handle);
```

To clear any currently stored events but then continue to monitor for further events, such as after an interval period, or at the start of a trial to clear any stale data from the previous trial, use:

```
>> PsychRTBox('Clear', handle);
```

In order to stop recording response events, use:

```
>> PsychRTBox('Stop', handle);
```

### 15.3.5 Setting which response types to monitor

By default, the only response events that will be logged are button presses. However some other events, such as a button release, can also be monitored if needed. To enable an additional event type to be recorded, use:

```
>> PsychRTBox('Enable', handle, eventType);
```

Similarly, to disable the monitoring of an event type, including a button press:

```
>> PsychRTBox('Disable', handle, eventType);
```

**An Important Note:** The possible event types to enable or disable are:

*'press' = push-button press*

*'release' = push-button release*

*'pulse'*

*'light'*

*'tr'*

*'all' = all events*

### 15.3.6 Retrieving recorded events

To read a logged event, the following command is used:

```
>> [time, event, boxTime] = PsychRTBox('GetSecs', handle);
```

If an event has occurred since the last call of this function, three arrays of values will be returned (If no new events have been returned, these arrays will be empty):

*time* = timestamps of the events, in host clock time.

*event* = event identity. For example, event will be 1 if button 1 was pressed.

*boxtime* = same timestamp of event but expressed in box clock time.

**An Important Note:** The default event identities are:

*'1' = Button 1 pressed '1up' = Button 1 released*

*'2' = Button 2 pressed '2up' = Button 2 released*

*'3' = Button 3 pressed '3up' = Button 3 released*

*'4' = Button 4 pressed '4up' = Button 4 released*

*'pulse'*

*'light'*

*'tr'*

### 15.3.7 Getting accurate measures of event times

Box timestamps can be converted to host clock time stamps at the end of an experiment by using:

```
>> [GetSecs, Stddev] = PsychRTBox('BoxsecsToGetsecs' [, handle], boxTimes);
```

The Stddev returns an error measure. This is an accurate way of mapping timestamps however only call it once at the very end of the function, when all of the data has been collected.

Perhaps the best way of measuring time is to measure the difference in box timestamps between two specified events. For example, the difference in box between a trigger indicating stimulus onset, and a response button being pressed.



# Chapter 16: Appendix

## 16.1 Using a Custom .edid File

The Extended Display Identification Data (EDID) is a 128-byte data structure with details about the display. It is provided by most modern monitors and transferred to the graphics card by the DVI or VGA cable connection. If however you are using another connection method or if you are using a display that is not compliant with EDID it is possible to setup the Bits# to emulate an EDID.

When you connect the Bits# to an EDID display with a DVI or VGA cable a copy of the EDID (named 'MON\_EDID.edid') will be written to the Firmware folder in the internal storage. By default this file will be passed on to the host computers graphics card but you can also specify your own. To do this you will need to change a line of code in the config.xml (in the Firmware folder) and add the file to the EDID folder:

In config.xml change the line:

```
<Entry TARGET_EDID="AUTO" />
```

to

```
<Entry TARGET_EDID="filename.edid" />
```

Where filename.edid is the name of the file you want Bits# to send to your graphics card. Place this file in the EDID folder on the Bits# internal storage.

For your convenience a number of EDIDs from common CRT screens are already located in this folder.

For further information on the EDID standard please consult the web sites:

<http://www.vesa.org>

[http://en.wikipedia.org/wiki/Extended\\_display\\_identification\\_data](http://en.wikipedia.org/wiki/Extended_display_identification_data)

## 16.2 CDC protocol for Bits#

Generically commands follow the syntax of either:

```
>>$<command>
```

Or if a parameter is updated:

```
>>$<command>=[<command data>]
```

The 'dollar' character before is to distinguish the command from a single character RTBox command (see next section). The commands must be followed by a carriage return (use ASCII code '13' in MATLAB). Returned results follow this format:

```
>>$<data type>;<value>;<value>;
```

Command	Arguments	Description
\$Help	(none)	Returns a list of all commands.
\$Start	(none)	Resets the sample number and starts acquiring analogue and digital input data. Returns: #sample;<sample#>;<timestamp>;<trigger-in bit value>;<digital input bits 0 .. 9>;<IR response bits 1 .. 6>;<ADC 1 .. 6>;
\$Stop	(none)	Stops data acquisition. No return.
\$ProductType	(none)	Returns the string "Bits_Sharp", or "Bits_Sharp_Basic" if it lacks the analogue board. Return example: #ProductType;Bits_Sharp;
\$SerialNumber	(none)	Returns the 8 character serial number. Returns: #SerialNumber;12345678;
\$FirmwareDate	(none)	Returns the firmware compilation date. Returns: #FirmwareDate;DD/MM/YYYY hr:min;
\$USB_massStorage	(none)	Enables Bits# as a mass storage device and disables the CDC interface. To revert back to CDC interface make sure Bits# is set to boot in CDC mode (see Installation) before power cycling Bits#. No return.
\$setMonitorType	=[<file-name.edid>]	Sets the EDID file that Bits# sends to the host computer. The EDID file must be located in the EDID folder of the internal storage. Returns the updated status, or if no argument, the current status. Default is 'AUTO'. Return example: \$setMonitorType=AUTO
\$autoPlusPlus	(none)	Lets Tlock control the video mode. No return.
\$monoPlusPlus	(none)	Switch to mono++ mode. No return.
\$colourPlusPlus	(none)	Switch to colour++ mode. No return.
\$colorPlusPlus	(none)	Switch to colour++ mode. No return.
\$BitsPlusPlus	(none)	Switch to bits++ mode. No return.
\$statusScreen	(none)	Force display of status screen when video is present. Revert with a video command e.g.

		\$monoPlusPlus. No return.
\$TemporalDithering	=[<ON   OFF>]	Enable or disable temporal dithering. Returns the updated status, or if no argument, the current status. Return Example: \$TemporalDithering = ON
\$enableGammaCorrection	=[<filename.txt>]	Sets the gamma correction look-up-table (LUT) to use. The file must be located in the Gamma folder in the Bits# internal storage. Factory default is a linear LUT (no correction) named 13bitLinearLUT.txt. Return Example: \$enableGammaCorrection;13bitLinearLUT.txt;
\$Beep	=[<freq.>,<duration>]	The Beep command activates a piezo sounder inside the Bits#. This can be used to provide feedback on e.g. button pushes. Valid arguments are in the range 10-20,000 Hz for frequency (freq.) and 0.0001-6.5 seconds for duration. NB. An extreme frequency or very short duration can be inaudible! Example use: \$Beep=[1000,2] . No return.
\$GetVideoLine	=[<line#>,<n>]	Returns n pixels starting from the line specified. This command is useful for debugging distortion in graphics card output, which can cause the Tlock commands to fail. NB. Status screen mode will be enabled when using this function. Example use: \$GetVideoLine=[6,265] Returns a string beginning with the command name followed by the RGB values of each pixel: #GetVideoLine;<R1>;<G1>;<B1>;<R2>;<G2>;-<B2>; ... <Rn>;<Gn>;<Bn>;
\$btn<ID>	=[<event source>]	Set the event source for one of the four buttons. <ID> can be 1,2,3 or 4. <event source> can be one of the 17 input event sources (see list below). No return.
\$light	=[<event source>]	Set the event source for the light. <event source> can be one of the 17 input event sources (see list below). No return.
\$pulse	=[<event source>]	Set the event source for the pulse. <event source> can be one of the 17 input event sources (see list below). No return.

\$trigger	=[<event source>]	Set the event source for the trigger. <event source> can be one of the 17 input event sources (see list below). No return.
\$VideoFrameRate	(none)	Returns the video frame rate in Hz. Return example: \$VideoFrameRate;60.02;
\$VideoPixelClock	(none)	Returns the video pixel clock in MHz. Return example: \$VideoPixelClock;108.0020;

### 16.2.1 List of event sources:

Din0  
 Din1  
 Din2  
 Din3  
 Din4  
 Din5  
 Din6  
 Din7  
 Din8  
 Din9  
 TrigIn  
 IRbuttonA  
 IRbuttonB  
 IRbuttonC  
 IRbuttonD  
 IRbuttonE  
 IRbuttonF

### 16.2.2 List of RTBox commands

Unlike the regular CDC commands, the RTBox commands should be send without a carriage return. When an event is enabled a 7-byte string will be sent for each event occurrence: the event ID followed by a 6-byte time stamp. The event IDs are: 49,51,53,55 for buttons 1 to 4 presses, 50, 52, 54, 56 for buttons 1 to 4 releases and 48, 57, 97 for light, pulse and TR events, respectively.

#### Command Description

?	Returns 2 bytes: the command and the current button state where the 5th to 8th bits corresponds to the state of button 1 to 4 (1 for pressed, 0 for unpressed).
A/a	Enable (A) or disable (a) buttons and TR/light/pulse. Returns the command.



D/d	Enable (D) and disable (d) button-down event (presses). Returns the command.
F/f	Enable (F) and disable (f) TR event. Returns the command.
O/o	Enable (O) and disable (o) Light event. Returns the command.
P/p	Enable (P) and disable (p) Pulse event. Returns the command.
U/u	Enable (U) and disable (u) button-up events (releases). Returns the command.
X/x	Enable (X) and disable (x) Advanced mode. Returns the command when disable. Returns device info when enable : USTCRTBOX,921600,v4.0
Y	Returns 7 bytes: the command and a 6-bytes time stamp of the current time of the device clock.

### 16.3 Single-Link DVI frame rates and resolutions

The Single link DVI standard is limited to a pixel clock frequency of 165 MHz. The pixel clock is defined by the display resolution and the frame rate plus an additional overhead. The amount of overhead depends on the timing standard being used. Typical timing standards include: GTF, DMT, CVT and CVT-RB. Below is shown a few examples of pixel formats and frame rates which are below the 165 MHz pixel clock limit (CVT timing standard):

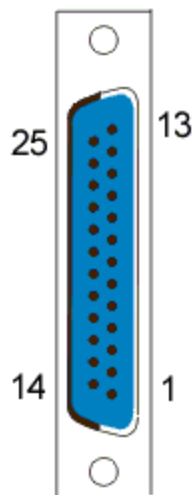
Horizontal resolution (pixels)	Vertical Resolution (pixels)	Frame Rate (Hz)	Pixel Clock (MHz)
640	480	200	95
800	600	200	149
1024	768	130	152
1152	864	100	143
1176	664	130	150
1280	720	110	147
1280	768	100	142
1280	800	100	147
1280	900	90	149
1280	1024	80	150
1360	768	100	150
1600	900	75	152

1600	1024	65	149
1600	1200	60*	130*
1680	1050	60	147
1768	992	60	145
1920	1080	60*	139*

\*using CVT-RB (Reduced Blanking). This an option on certain monitors to allow more active video throughput.

### 16.4 Digital I/O Connector Pinout

The digital I/O connector is a standard 25-pin female D-sub connector (DB-25F) and has the following pinout:



Pin No	Signal
1	DOUT 0
2	DOUT 1
3	DOUT 2
4	DOUT 3
5	DOUT 4
6	DOUT 5
7	DOUT 6
8	DOUT 7

9	DOUT 8
10	N/C
11	DOUT 9
12	DGND
13	DGND
14	DIN 0
15	DIN 1
16	DIN 2
17	DIN 3
18	DIN 4
19	DIN 5
20	DIN 6
21	DIN 7
22	+5V 200mA
23	+5V 200mA
24	DIN 8
25	DIN 9

